

Полностью обновлено с учетом
версии Java SE 8 (JDK 8)

Java 8

Полное руководство

Девятое издание



Исчерпывающее и незаменимое учебное пособие по написанию, компилированию и выполнению современных программ на Java

Герберт Шилдт

Oracle
Press™

ORACLE

Oracle Press™

**Полное
руководство**

Java 8
Девятое издание

ORACLE

Oracle Press™

**The
Complete
Reference**

Java
Ninth Edition

Herbert Schildt

**Mc
Graw
Hill**
Education

New York ♦ Chicago ♦ San Francisco ♦ Athens ♦ London ♦ Madrid
Mexico City ♦ Milan ♦ New Delhi ♦ Singapore ♦ Sydney ♦ Toronto

ORACLE

Oracle Press™

**Полное
руководство**

Java 8
Девятое издание

Герберт Шилдт



Издательский дом "Вильямс"
Москва ♦ Санкт-Петербург ♦ Киев
2015

ББК 32.973.26-018.2.75

Ш57

УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *И.В. Берштейна*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Шилдт, Герберт.

Ш57 Java 8. Полное руководство, 9-е изд. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2015. — 1376 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1918-2 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства McGraw-Hill Education.

Authorized translation from the English language edition published by McGraw-Hill Education, Copyright © 2014.

All rights reserved. Except as permitted under the Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of Publisher, with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

Oracle and Java are registered trademarks of Oracle Corporation and/or its affiliates.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2015

Научно-популярное издание

Герберт Шилдт

Java 8. Полное руководство

9-е издание

Литературный редактор	<i>И.А. Попова</i>
Верстка	<i>О.В. Мишутина</i>
Художественный редактор	<i>Е.П. Дынный</i>
Корректор	<i>Л.А. Гордиенко</i>

Подписано в печать 14.01.2015. Формат 70х100/16

Гарнитура Times. Усл. печ. л. 86,0. Уч.-изд. л. 79,2

Тираж 1000 экз. Заказ № 267

Отпечатано способом ролевой струйной печати

в ОАО «Первая Образцовая типография»

Филиал «Чеховский Печатный Двор»

142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1918-2 (рус.)

ISBN 978-0-07-180855-2 (англ.)

© Издательский дом "Вильямс", 2015

© McGraw-Hill Education, 2014

Оглавление

Предисловие	28
Часть I. Язык Java	31
Глава 1. История и развитие языка Java	33
Глава 2. Краткий обзор Java	53
Глава 3. Типы данных, переменные и массивы	75
Глава 4. Операции	103
Глава 5. Управляющие операторы	125
Глава 6. Введение в классы	155
Глава 7. Подробное рассмотрение классов и методов	177
Глава 8. Наследование	209
Глава 9. Пакеты и интерфейсы	235
Глава 10. Обработка исключений	263
Глава 11. Многопоточное программирование	285
Глава 12. Перечисления, автоупаковка и аннотации (метаданные)	317
Глава 13. Ввод-вывод, апплеты и прочие вопросы	355
Глава 14. Обобщения	395
Глава 15. Лямбда-выражения	437
Часть II. Библиотека Java	467
Глава 16. Обработка символьных строк	469
Глава 17. Пакет <code>java.lang</code>	497
Глава 18. Пакет <code>java.util</code> , часть I. <code>Collections Framework</code>	563
Глава 19. Пакет <code>java.util</code> , часть II. Прочие служебные классы	651
Глава 20. Пакет <code>java.io</code> для ввода-вывода	717
Глава 21. Система ввода-вывода NIO	769
Глава 22. Работа в сети	811
Глава 23. Класс <code>Applet</code>	833
Глава 24. Обработка событий	855
Глава 25. Введение в библиотеку AWT: работа с окнами, графикой и текстом	885
Глава 26. Применение элементов управления, диспетчеров компоновки и меню из библиотеки AWT	923
Глава 27. Изображения	975
Глава 28. Утилиты параллелизма	1005
Глава 29. Поточковый API	1061
Глава 30. Регулярные выражения и другие пакеты	1089
Часть III. Введение в программирование ГПИ средствами Swing	1119
Глава 31. Введение в библиотеку Swing	1121
Глава 32. Исследование библиотеки Swing	1143
Глава 33. Введение в меню Swing	1175
Часть IV. Введение в программирование ГПИ средствами JavaFX	1209
Глава 34. Введение в JavaFX	1211
Глава 35. Элементы управления JavaFX	1233
Глава 36. Введение в меню JavaFX	1281
Часть V. Применение Java	1309
Глава 37. Компоненты <code>Java Beans</code>	1311
Глава 38. Введение в сервлеты	1323
Приложение	1347
Применение документирующих комментариев в Java	1347
Предметный указатель	1355

Содержание

Предисловие	28
Часть I. Язык Java	31
Глава 1. История и развитие языка Java	33
Происхождение Java	33
Зарождение современного программирования: язык C	34
Следующий этап: язык C++	35
Предпосылки к созданию Java	37
Создание языка Java	37
Связь с языком C#	39
Каким образом язык Java изменил Интернет	40
Апплеты на Java	40
Безопасность	41
Переносимость	41
Чудо Java: байт-код	41
Сервлеты: серверные программы на Java	43
Отличительные особенности Java	43
Простота	44
Объектная ориентированность	44
Надежность	44
Многопоточность	45
Архитектурная нейтральность	45
Интерпретируемость и высокая производительность	46
Распределенность	46
Динамичность	46
Эволюция языка Java	46
Версия Java SE 8	50
Культура нововведений	51
Глава 2. Краткий обзор Java	53
Объектно-ориентированное программирование	53
Две методики	53
Абстракция	54
Три принципа ООП	54

Первый пример простой программы	60
Ввод кода программы	61
Компиляция программы	61
Подробный анализ первого примера программы	62
Второй пример короткой программы	65
Два управляющих оператора	66
Условный оператор if	67
Оператор цикла for	68
Использование блоков кода	69
Вопросы лексики	70
Пробелы	71
Идентификаторы	71
Литералы	71
Комментарии	72
Разделители	72
Ключевые слова Java	72
Библиотеки классов Java	73
Глава 3. Типы данных, переменные и массивы	75
Java – строго типизированный язык	75
Примитивные типы	75
Целые числа	76
Тип byte	77
Тип short	77
Тип int	77
Тип long	78
Числа с плавающей точкой	78
Тип float	79
Тип double	79
Символы	80
Логические значения	81
Подробнее о литералах	82
Целочисленные литералы	82
Литералы с плавающей точкой	84
Логические литералы	85
Символьные литералы	85
Строковые литералы	86
Переменные	86
Объявление переменной	86
Динамическая инициализация	87
Область и срок действия переменных	87
Преобразование и приведение типов	90
Автоматическое преобразование типов в Java	90
Приведение несовместимых типов	91
Автоматическое продвижение типов в выражениях	92
Правила продвижения типов	93
Массивы	94
Одномерные массивы	94
Многомерные массивы	97
Альтернативный синтаксис объявления массивов	101
Введение в символьные строки	101
Замечание по поводу указателей для программирующих на C/C++	102

Глава 4 . Операции	103
Арифметические операции	103
Основные арифметические операции	104
Операция деления по модулю	105
Составные арифметические операции с присваиванием	105
Операции инкремента и декремента	106
Поразрядные операции	108
Поразрядные логические операции	109
Сдвиг влево	111
Сдвиг вправо	113
Беззнаковый сдвиг вправо	114
Поразрядные составные операции с присваиванием	116
Операции отношения	117
Логические операции	118
Укороченные логические операции	120
Операция присваивания	120
Тернарная операция ?	121
Предшествование операций	122
Применение круглых скобок	123
Глава 5. Управляющие операторы	125
Операторы выбора	125
Условный оператор if	125
Оператор switch	128
Операторы цикла	133
Цикл while	133
Цикл do-while	135
Цикл for	138
Вложенные циклы	147
Операторы перехода	147
Применение оператора break	148
Применение оператора continue	151
Оператор return	153
Глава 6. Введение в классы	155
Основы классов	155
Общая форма класса	155
Простой класс	156
Объявление объектов	159
Подробное рассмотрение оператора new	160
Присваивание переменным ссылок на объекты	161
Введение в методы	162
Ввод метода в класс Box	163
Возврат значений	164
Ввод метода, принимающего параметры	166
Конструкторы	168
Параметризованные конструкторы	170
Ключевое слово this	171
Соккрытие переменной экземпляра	172
Сборка "мусора"	172
Метод finalize()	173
Класс Stack	174

Глава 7. Подробное рассмотрение классов и методов	177
Перегрузка методов	177
Перегрузка конструкторов	180
Применение объектов в качестве параметров	182
Подробное рассмотрение особенностей передачи аргументов	184
Возврат объектов	186
Рекурсия	187
Введение в управление доступом	189
Ключевое слово <code>static</code>	193
Ключевое слово <code>final</code>	194
Еще раз о массивах	195
Вложенные и внутренние классы	197
Краткий обзор класса <code>String</code>	200
Применение аргументов командной строки	202
Аргументы переменной длины	203
Перегрузка методов с аргументами переменной длины	206
Аргументы переменной длины и неоднозначность	207
Глава 8. Наследование	209
Основы наследования	209
Доступ к членам класса и наследование	211
Более практический пример	212
Переменная суперкласса может ссылаться на объект подкласса	213
Ключевое слово <code>super</code>	214
Вызов конструкторов суперкласса с помощью ключевого слова <code>super</code>	215
Другое применение ключевого слова <code>super</code>	218
Создание многоуровневой иерархии	219
Порядок вызова конструкторов	222
Переопределение методов	223
Динамическая диспетчеризация методов	225
Назначение переопределенных методов	226
Применение переопределения методов	227
Применение абстрактных классов	228
Ключевое слово <code>final</code> в сочетании с наследованием	231
Предотвращение переопределения с помощью ключевого слова <code>final</code>	231
Предотвращения наследования с помощью ключевого слова <code>final</code>	232
Класс <code>Object</code>	233
Глава 9. Пакеты и интерфейсы	235
Пакеты	235
Определение пакета	236
Поиск пакетов и переменная окружения <code>CLASSPATH</code>	236
Краткий пример пакета	237
Защита доступа	238
Пример защиты доступа	239
Импорт пакетов	242
Интерфейсы	244

Объявление интерфейса	245
Реализация интерфейсов	246
Вложенные интерфейсы	249
Применение интерфейсов	250
Переменные в интерфейсах	253
Расширение интерфейсов	255
Методы по умолчанию	255
Основы применения методов по умолчанию	257
Более практический пример	258
Вопросы множественного наследования	259
Применение статических методов в интерфейсе	260
Заключительные соображения по поводу пакетов и интерфейсов	261
Глава 10. Обработка исключений	263
Основы обработки исключений	263
Типы исключений	264
Необрабатываемые исключения	265
Использование блоков операторов <code>try</code> и <code>catch</code>	266
Вывод описания исключения	268
Применение нескольких операторов <code>catch</code>	268
Вложенные операторы <code>try</code>	270
Оператор <code>throw</code>	272
Оператор <code>throws</code>	273
Оператор <code>finally</code>	275
Встроенные в Java исключения	276
Создание собственных подклассов исключений	278
Цепочки исключений	280
Недавно введенные средства для обработки исключений	282
Применение исключений	284
Глава 11. Многопоточное программирование	285
Модель потоков исполнения в Java	286
Приоритеты потоков	287
Синхронизация	288
Обмен сообщениями	289
Класс <code>Thread</code> и интерфейс <code>Runnable</code>	289
Главный поток исполнения	290
Создание потока исполнения	291
Реализация интерфейса <code>Runnable</code>	292
Расширение класса <code>Thread</code>	294
Выбор способа создания потоков исполнения	295
Создание многих потоков исполнения	295
Применение методов <code>isAlive()</code> и <code>join()</code>	297
Приоритеты потоков исполнения	299
Синхронизация	300
Применение синхронизированных методов	301
Оператор <code>synchronized</code>	303
Взаимодействие потоков исполнения	304
Взаимная блокировка	309

Приостановка, возобновление и остановка потоков исполнения	311
Получение состояния потока исполнения	313
Применение многопоточности	315
Глава 12. Перечисления, автоупаковка и аннотации (метаданные)	317
Перечисления	317
Основные положения о перечислениях	317
Методы <code>values()</code> и <code>valueOf()</code>	319
Перечисления в Java относятся к типам классов	321
Перечисления наследуются от класса <code>Enum</code>	323
Еще один пример перечисления	325
Оболочки типов	326
Класс <code>Character</code>	327
Класс <code>Boolean</code>	327
Оболочки числовых типов	327
Автоупаковка	328
Автоупаковка и методы	329
Автоупаковка и автораспаковка в выражениях	330
Автоупаковка и распаковка значений из классов <code>Boolean</code> и <code>Character</code>	332
Автоупаковка и автораспаковка помогают предотвратить ошибки	333
Предупреждение	333
Аннотации (метаданные)	334
Основы аннотирования программ	334
Правила удержания аннотаций	335
Получение аннотаций во время выполнения с помощью рефлексии	336
Второй пример применения рефлексии	338
Получение всех аннотаций	339
Интерфейс <code>AnnotatedElement</code>	341
Использование значений по умолчанию	341
Аннотации-маркеры	343
Одночленные аннотации	343
Встроенные аннотации	345
Типовые аннотации	347
Повторяющиеся аннотации	352
Некоторые ограничения на аннотации	353
Глава 13. Ввод-вывод, апплеты и прочие вопросы	355
Основы ввода-вывода	355
Потоки ввода-вывода	356
Потоки ввода-вывода байтов и символов	356
Предопределенные потоки ввода-вывода	359
Чтение данных, вводимых с консоли	359
Чтение символов	360
Чтение символьных строк	361
Запись данных, выводимых на консоль	362
Класс <code>PrintWriter</code>	363
Чтение и запись данных в файлы	364
Автоматическое закрытие файла	370
Основы создания апплетов	374

Модификаторы доступа <code>transient</code> и <code>volatile</code>	377
Применение оператора <code>instanceof</code>	378
Модификатор доступа <code>strictfp</code>	380
Платформенно-ориентированные методы	381
Трудности, связанные с платформенно-ориентированными методами	384
Применение ключевого слова <code>assert</code>	385
Параметры включения и отключения режима проверки утверждений	388
Статический импорт	388
Вызов перегружаемых конструкторов по ссылке <code>this()</code>	391
Компактные профили Java API	393
Глава 14. Обобщения	395
Что такое обобщения	396
Простой пример обобщения	396
Обобщения действуют только со ссылочными типами	400
Обобщенные типы различаются по аргументам типа	400
Каким образом обобщения повышают типовую безопасность	401
Обобщенный класс с двумя параметрами типа	403
Общая форма обобщенного класса	404
Ограниченные типы	405
Применение метасимвольных аргументов	407
Ограниченные метасимвольные аргументы	410
Создание обобщенного метода	414
Обобщенные конструкторы	417
Обобщенные интерфейсы	417
Базовые типы и унаследованный код	420
Иерархии обобщенных классов	422
Применение обобщенного суперкласса	422
Обобщенный подкласс	424
Сравнение типов в обобщенной иерархии во время выполнения	425
Приведение типов	428
Переопределение методов в обобщенном классе	428
Выведение типов и обобщения	429
Стирание	431
Мостовые методы	431
Ошибки неоднозначности	433
Некоторые ограничения, присущие обобщениям	434
Получить экземпляр по параметру типа нельзя	434
Ограничения на статические члены	435
Ограничения на обобщенные массивы	435
Ограничения на обобщенные исключения	436
Глава 15. Лямбда-выражения	437
Введение в лямбда-выражения	438
Основные положения о лямбда-выражениях	438
Функциональные интерфейсы	439
Некоторые примеры лямбда-выражений	440
Блочные лямбда-выражения	444
Обобщенные функциональные интерфейсы	446
Передача лямбда-выражений в качестве аргументов	447
Лямбда-выражения и исключения	450

Лямбда-выражения и захват переменных	451
Ссылки на методы	453
Ссылки на статические методы	453
Ссылки на методы экземпляра	454
Ссылки на обобщенные методы	458
Ссылки на конструкторы	460
Предопределенные функциональные интерфейсы	465
Часть II. Библиотека Java	467
Глава 16. Обработка символьных строк	469
Конструкторы символьных строк	470
Длина символьной строки	472
Специальные строковые операции	472
Строковые литералы	472
Сцепление строк	473
Сцепление символьных строк с другими типами данных	473
Преобразование символьных строк и метод <code>toString()</code>	474
Извлечение символов	475
Метод <code>charAt()</code>	475
Метод <code>getChars()</code>	476
Метод <code>getBytes()</code>	476
Метод <code>toCharArray()</code>	477
Сравнение символьных строк	477
Методы <code>equals()</code> и <code>equalsIgnoreCase()</code>	477
Метод <code>regionMatches()</code>	478
Методы <code>startsWith()</code> и <code>endsWith()</code>	478
Метод <code>equals()</code> в сравнении с операцией <code>==</code>	479
Метод <code>compareTo()</code>	480
Поиск в символьных строках	481
Видоизменение символьных строк	482
Метод <code>substring()</code>	483
Метод <code>concat()</code>	484
Метод <code>replace()</code>	484
Метод <code>trim()</code>	484
Преобразование данных методом <code>valueOf()</code>	485
Изменение регистра символов в строке	486
Соединение символьных строк	487
Дополнительные методы из класса <code>String</code>	487
Класс <code>StringBuffer</code>	489
Конструкторы класса <code>StringBuffer</code>	489
Методы <code>length()</code> и <code>capacity()</code>	490
Метод <code>ensureCapacity()</code>	490
Метод <code>setLength()</code>	491
Методы <code>charAt()</code> и <code>setCharAt()</code>	491
Метод <code>getChars()</code>	492
Метод <code>append()</code>	492
Метод <code>insert()</code>	493
Метод <code>reverse()</code>	493
Методы <code>delete()</code> и <code>deleteCharAt()</code>	494

Метод <code>replace()</code>	494
Метод <code>substring()</code>	495
Дополнительные методы из класса <code>StringBuffer</code>	495
Класс <code>StringBuilder</code>	496
Глава 17. Пакет <code>java.lang</code>	497
Оболочки примитивных типов	498
Класс <code>Number</code>	498
Классы <code>Double</code> и <code>Float</code>	498
Методы <code>isInfinite()</code> и <code>isNaN()</code>	503
Классы <code>Byte</code> , <code>Short</code> , <code>Integer</code> и <code>Long</code>	503
Класс <code>Character</code>	514
Дополнения класса <code>Character</code> для поддержки кодовых точек в Юникоде	517
Класс <code>Boolean</code>	519
Класс <code>Void</code>	520
Класс <code>Process</code>	520
Класс <code>Runtime</code>	521
Управление памятью	523
Выполнение других программ	524
Класс <code>ProcessBuilder</code>	525
Класс <code>System</code>	528
Измерение времени выполнения программы	
методом <code>currentTimeMills()</code>	530
Применение метода <code>arraycopy()</code>	531
Свойства окружения	532
Класс <code>Object</code>	532
Применение метода <code>clone()</code> и интерфейса <code>Cloneable</code>	533
Класс <code>Class</code>	535
Класс <code>ClassLoader</code>	538
Класс <code>Math</code>	539
Тригонометрические функции	539
Экспоненциальные функции	540
Функции округления	540
Прочие методы из класса <code>Math</code>	542
Класс <code>StrictMath</code>	544
Класс <code>Compiler</code>	545
Классы <code>Thread</code> , <code>ThreadGroup</code> и интерфейс <code>Runnable</code>	545
Интерфейс <code>Runnable</code>	545
Класс <code>Thread</code>	545
Класс <code>ThreadGroup</code>	548
Классы <code>ThreadLocal</code> и <code>InheritableThreadLocal</code>	553
Класс <code>Package</code>	553
Класс <code>RuntimePermission</code>	555
Класс <code>Throwable</code>	555
Класс <code>SecurityManager</code>	555
Класс <code>StackTraceElement</code>	555
Класс <code>Enum</code>	556
Класс <code>ClassValue</code>	557
Интерфейс <code>CharSequence</code>	558
Интерфейс <code>Comparable</code>	558
Интерфейс <code>Appendable</code>	559

Интерфейс Iterable	559
Интерфейс Readable	560
Интерфейс AutoCloseable	560
Интерфейс Thread.UncaughtExceptionHandler	560
Подпакеты из пакета в java.lang	561
Пакет java.lang.annotation	561
Пакет java.lang.instrument	561
— Пакет java.lang.invoke	561
Пакет java.lang.management	561
Пакет java.lang.ref	562
Пакет java.lang.reflect	562
Глава 18. Пакет java.util, часть I. Collections Framework	563
Краткий обзор коллекций	564
Изменения каркаса коллекций в версии JDK 5	566
Обобщения коренным образом изменили каркас коллекций	566
В средствах автоматической упаковки используются	
примитивные типы данных	567
Цикл for в стиле for each	567
Интерфейсы коллекций	567
Интерфейс Collection	568
Интерфейс List	571
Интерфейс Set	573
Интерфейс SortedSet	573
Интерфейс NavigableSet	574
Интерфейс Queue	576
Интерфейс Dequeue	577
Классы коллекций	579
Класс ArrayList	580
Класс LinkedList	584
Класс HashSet	585
Класс LinkedHashSet	587
Класс TreeSet	587
Класс PriorityQueue	588
Класс ArrayDeque	589
Класс EnumSet	590
Доступ к коллекциям через итератор	591
Применение интерфейса Iterator	593
Цикл for в стиле for each как альтернатива итераторам	595
Итераторы-разделители	596
Сохранение объектов пользовательских классов в коллекциях	599
Интерфейс RandomAccess	600
Обращение с отображениями	601
Интерфейсы отображений	601
Классы отображений	609
Компараторы	614
Применение компараторов	617
Алгоритмы коллекций	622
Массивы	629
Унаследованные классы и интерфейсы	634

Интерфейс Enumeration	635
Класс Vector	635
Класс Stack	639
Класс Dictionary	641
Класс Properties	645
Применение методов store() и load()	649
Заключительные соображения по поводу коллекций	650
Глава 19. Пакет java.util, часть II. Прочие служебные классы	651
Класс StringTokenizer	651
Класс BitSet	653
Классы Optional, OptionalDouble, OptionalInt и OptionalLong	656
Класс Date	659
Класс Calendar	661
Класс GregorianCalendar	664
Класс TimeZone	666
Класс SimpleTimeZone	667
Класс Locale	668
Класс Random	670
Класс Observable	672
Интерфейс Observer	673
Пример наблюдения за объектами	674
Классы Timer и TimerTask	676
Класс Currency	678
Класс Formatter	680
Конструкторы класса Formatter	680
Методы из класса Formatter	681
Основы форматирования	682
Форматирование строк и символов	684
Форматирование чисел	684
Форматирование времени и даты	685
Спецификаторы формата %n и %%	687
Указание минимальной ширины поля	687
Указание точности	689
Применение признаков формата	689
Выравнивание выводимых данных	690
Признаки пробела, +, 0 и (691
Признак запятой	692
Признак	692
Прописные формы спецификаторов формата	692
Применение индекса аргумента	693
Закрывание объекта типа Formatter	694
Аналог функции printf() в Java	695
Класс Scanner	695
Конструкторы класса Scanner	695
Некоторые примеры применения класса Scanner	700
Установка разделителей	704
Прочие средства класса Scanner	705
Классы ResourceBundle, ListResourceBundle и PropertyResourceBundle	706

Прочие служебные классы и интерфейсы	710
Пакеты java.util.concurrent, java.util.concurrent.atomic, java.util.concurrent.locks	712
Пакет java.util.function	712
Пакет java.util.jar	716
Пакет java.util.logging	716
Пакет java.util.prefs	716
Пакет java.util.regex	716
Пакет java.util.spi	716
Пакет java.util.stream	716
Пакет java.util.zip	716
Глава 20. Пакет java.io для ввода-вывода	717
Классы и интерфейсы ввода-вывода	718
Класс File	718
Каталоги	722
Применение интерфейса FilenameFilter	723
Альтернативный метод listFiles()	724
Создание каталогов	724
Интерфейсы AutoCloseable, Closeable и Flushable	724
Исключения ввода-вывода	725
Два способа закрытия потока ввода-вывода	726
Классы потоков ввода-вывода	727
Потоки ввода-вывода байтов	728
Класс InputStream	728
Класс OutputStream	729
Класс FileInputStream	730
Класс FileOutputStream	732
Класс ByteArrayInputStream	734
Класс ByteArrayOutputStream	736
Фильтруемые потоки ввода-вывода байтов	737
Буферизованные потоки ввода-вывода байтов	737
Потоки ввода-вывода символов	748
Класс Reader	749
Класс Writer	750
Класс FileReader	751
Класс FileWriter	751
Класс CharArrayReader	752
Класс CharArrayWriter	754
Класс BufferedReader	755
Класс BufferedWriter	756
Класс PushbackReader	757
Класс PrintWriter	758
Класс Console	759
Сериализация	761
Интерфейс Serializable	762
Интерфейс Externalizable	762
Интерфейс ObjectOutput	763
Класс ObjectOutputStream	763
Интерфейс ObjectInput	764

Класс <code>ObjectInputStream</code>	765
Пример сериализации	767
Преимущества потоков ввода-вывода	768
Глава 21. Система ввода-вывода NIO	769
Классы системы ввода-вывода NIO	769
Основные положения о системе ввода-вывода NIO	770
Буфера	770
Каналы	772
Наборы символов и селекторы	774
Усовершенствования в системе NIO, начиная с версии JDK 7	774
Интерфейс <code>Path</code>	774
Класс <code>Files</code>	776
Класс <code>Paths</code>	779
Интерфейсы атрибутов файлов	780
Классы <code>FileSystem</code> , <code>FileSystems</code> и <code>FileStore</code>	782
Применение системы ввода-вывода NIO	782
Применение системы NIO для канального ввода-вывода	783
Применение системы NIO для потокового ввода-вывода	793
Применение системы ввода-вывода NIO для операций в файловой системе	796
Примеры организации канального ввода-вывода до версии JDK 7	804
Чтение из файла до версии JDK 7	804
Запись в файл версии до JDK 7	807
Глава 22. Работа в сети	811
Основы работы в сети	811
Сетевые классы и интерфейсы	813
Класс <code>InetAddress</code>	813
Фабричные методы	814
Методы экземпляра	815
Классы <code>Inet4Address</code> и <code>Inet6Address</code>	815
Клиентские сокеты по протоколу TCP/IP	816
Класс <code>URL</code>	819
Класс <code>URLConnection</code>	821
Класс <code>HttpURLConnection</code>	824
Класс <code>URI</code>	826
Cookie-файлы	826
Серверные сокеты по протоколу TCP/IP	827
Дейтаграммы	827
Класс <code>DatagramSocket</code>	828
Класс <code>DatagramPacket</code>	829
Пример обработки дейтаграмм	830
Глава 23. Класс <code>Applet</code>	833
Два типа апплетов	833
Основы разработки апплетов	834
Класс <code>Applet</code>	835
Структура апплетов	837
Скелет апплета	838
Инициализация и прекращение работы апплета	839

Переопределение метода update ()	840
Простые методы воспроизведения апплетов	841
Запрос на повторное воспроизведение	843
Простой апплет с баннером	844
Применение строки состояния	846
HTML-дескриптор APPLET	847
Передача параметров апплетам	848
Усовершенствование апплета, воспроизводящего баннер	850
Методы getDocumentBase () и getCodeBase ()	851
Интерфейс AppletContext и метод showDocument ()	852
Интерфейс AudioClip	854
Интерфейс AppletStub	854
Консольный вывод	854
Глава 24. Обработка событий	855
Два механизма обработки событий	855
Модель делегирования событий	856
События	856
Источники событий	857
Приемники событий	857
Классы событий	858
Класс ActionEvent	859
Класс AdjustmentEvent	860
Класс ComponentEvent	861
Класс ContainerEvent	861
Класс FocusEvent	862
Класс InputEvent	863
Класс ItemEvent	864
Класс KeyEvent	864
Класс MouseEvent	865
Класс MouseWheelEvent	867
Класс TextEvent	868
Класс WindowEvent	868
Источники событий	870
Интерфейсы приемников событий	870
Интерфейс ActionListener	871
Интерфейс AdjustmentListener	871
Интерфейс ComponentListener	872
Интерфейс ContainerListener	872
Интерфейс FocusListener	872
Интерфейс ItemListener	872
Интерфейс KeyListener	872
Интерфейс MouseListener	873
Интерфейс MouseMotionListener	873
Интерфейс MouseWheelListener	873
Интерфейс TextListener	873
Интерфейс WindowFocusListener	873
Интерфейс WindowListener	873
Применение модели делегирования событий	874
Обработка событий от мыши	874
Обработка событий от клавиатуры	877

Классы адаптеров	880
Внутренние классы	882
Анонимные внутренние классы	883
Глава 25. Введение в библиотеку AWT: работа с окнами, графикой и текстом	885
Классы библиотеки AWT	886
Основные положения об окнах	888
Класс Component	889
Класс Container	889
Класс Panel	889
Класс Window	890
Класс Frame	890
Класс Canvas	890
Работа с обрамляющими окнами	890
Установка размеров окна	891
Соккрытие и отображение окна	891
Установка заголовка окна	891
Заккрытие обрамляющего окна	891
Создание обрамляющего окна в аплете, построенном на основе библиотеки AWT	892
Обработка событий в обрамляющем окне	894
Создание оконной прикладной программы	898
Отображение информации в окне	899
Поддержка графики	900
Рисование линий	900
Рисование прямоугольников	900
Рисование эллипсов и окружностей	901
Рисование дуг	901
Рисование многоугольников	901
Демонстрация методов рисования	902
Изменение размеров графики	902
Работа с цветом	904
Методы из класса Color	905
Установка режима рисования	907
Работа со шрифтами	909
Определение доступных шрифтов	910
Создание и выбор шрифта	911
Получение сведений о шрифте	913
Управление форматированием выводимого текста	914
Отображение многострочного текста	915
Центровка текста	917
Выравнивание многострочного текста	918
Глава 26. Применение элементов управления, диспетчеров компоновки и меню из библиотеки AWT	923
Основные положения об элементах управления	924
Ввод и удаление элементов управления	924
Реагирование на элементы управления	925
Исключение типа HeadlessException	925

Метки	925
Экранные кнопки	926
Обработка событий от кнопок	927
Флажки	930
Обработка событий от флажков	931
Кнопки-переключатели	932
Элементы управления выбором	934
Обработка событий от раскрывающихся списков	935
Использование списков	936
Обработка событий от списков	938
Управление полосами прокрутки	939
Обработка событий от полос прокрутки	941
Текстовые поля	943
Обработка событий в текстовых полях	944
Текстовые области	945
Диспетчеры компоновки	947
Класс <code>FlowLayout</code>	948
Класс <code>BorderLayout</code>	950
Вставки	952
Класс <code>GridLayout</code>	953
Класс <code>CardLayout</code>	954
Класс <code>GridBagLayout</code>	957
Меню и строки меню	962
Диалоговые окна	968
Диалоговые окна выбора файлов	972
О переопределении метода <code>paint()</code>	973
Глава 27. Изображения	975
Форматы файлов	975
Основы работы с изображениями: создание, загрузка и отображение	976
Создание объекта класса <code>Image</code>	976
Загрузка изображения	977
Воспроизведение изображения	977
Интерфейс <code>ImageObserver</code>	979
Двойная буферизация	980
Класс <code>MediaTracker</code>	983
Интерфейс <code>ImageProducer</code>	986
Класс <code>MemoryImageSource</code>	986
Интерфейс <code>ImageConsumer</code>	988
Класс <code>PixelGrabber</code>	988
Класс <code>ImageFilter</code>	991
Класс <code>CropImageFilter</code>	991
Фильтр класса <code>RGBImageFilter</code>	992
Дополнительные классы для формирования изображений	1004
Глава 28. Утилиты параллелизма	1005
Пакеты параллельного API	1006
Пакет <code>java.util.concurrent.atomic</code>	1008
Пакет <code>java.util.concurrent.locks</code>	1008
Применение объектов синхронизации	1008

Класс Semaphore	1008
Класс CountdownLatch	1014
Класс CyclicBarrier	1016
Класс Exchanger	1018
Класс Phaser	1020
Применение исполнителя	1028
Простой пример исполнителя	1029
Применение интерфейсов Callable и Future	1031
Перечисление TimeUnit	1034
Параллельные коллекции	1035
Блокировки	1036
Атомарные операции	1039
Параллельное программирование средствами Fork/Join Framework	1040
Основные классы Fork/Join Framework	1041
Стратегия "разделяй и властвуй"	1045
Первый простой пример вилочного соединения	1047
Влияние уровня параллелизма	1049
Пример применения класса RecursiveTask<V>	1053
Асинхронное выполнение задач	1055
Отмена задачи	1056
Определение состояния завершения задачи	1056
Перезапуск задачи	1056
Предмет дальнейшего изучения	1057
Рекомендации относительно вилочного соединения	1059
Утилиты параллелизма в сравнении с традиционным подходом к многозадачности в Java	1059
Глава 29. Поточковый API	1061
Основные положения о потоках данных	1061
Потоковые интерфейсы	1062
Получение потока данных	1065
Простой пример потока данных	1066
Операции сведения	1070
Параллельные потоки данных	1073
Отображение	1075
Накопление	1079
Итераторы и потоки данных	1083
Применение итератора в потоке данных	1083
Применение итератора-разделителя	1085
Дальнейшее изучение потокового API	1087
Глава 30. Регулярные выражения и другие пакеты	1089
Пакеты из базового API	1089
Обработка регулярных выражений	1092
Класс Pattern	1092
Класс Matcher	1092
Синтаксис регулярных выражений	1093
Примеры, демонстрирующие совпадение с шаблоном	1094
Два варианта сопоставления с шаблоном	1100
Дальнейшее изучение регулярных выражений	1100
Рефлексия	1101

Удаленный вызов методов	1105
Простое приложение “клиент–сервер”, использующее механизм RMI	1105
Форматирование даты и времени средствами пакета <code>java.text</code>	1109
Класс <code>DateFormat</code>	1109
Класс <code>SimpleDateFormat</code>	1111
API даты и времени, введенный в версии JDK 8	1113
Основные классы даты и времени	1113
Форматирование даты и времени	1115
Синтаксический анализ символьных строк даты и времени	1117
Дальнейшее изучение пакета <code>java.time</code>	1118
Часть III. Введение в программирование ГПИ средствами Swing	1119
Глава 31. Введение в библиотеку Swing	1121
Происхождение библиотеки Swing	1121
Построение библиотеки Swing на основе библиотеки AWT	1122
Главные особенности библиотеки Swing	1122
Легковесные компоненты Swing	1123
Подключаемый стиль оформления	1123
Связь с архитектурой MVC	1124
Компоненты и контейнеры	1125
Компоненты	1125
Контейнеры	1126
Панели контейнеров верхнего уровня	1126
Пакеты библиотеки Swing	1127
Простое Swing-приложение	1127
Обработка событий	1132
Создание Swing-апплета	1135
Рисование средствами Swing	1138
Основы рисования	1138
Вычисление области рисования	1139
Пример рисования	1140
Глава 32. Исследование библиотеки Swing	1143
Классы <code>JLabel</code> и <code>ImageIcon</code>	1143
Класс <code>JTextField</code>	1145
Кнопки из библиотеки Swing	1147
Класс <code>JButton</code>	1148
Класс <code>JToggleButton</code>	1150
Флажки	1153
Кнопки-переключатели	1155
Класс <code>JTabbedPane</code>	1157
Класс <code>JScrollPane</code>	1160
Класс <code>JList</code>	1162
Класс <code>JComboBox</code>	1166
Деревья	1168
Класс <code>JTable</code>	1172
Основные положения о меню	1175

Глава 33. Введение в меню Swing	1175
Краткий обзор классов JMenuBar, JMenu и JMenuItem	1177
Класс JMenuBar	1177
Класс JMenu	1178
Класс JMenuItem	1180
Создание главного меню	1180
Ввод мнемоники и оперативных клавиш в меню	1184
Ввод изображений и всплывающих подсказок в пункты меню	1187
Классы JRadioButtonMenuItem и JCheckBoxMenuItem	1188
Создание всплывающего меню	1190
Создание панели инструментов	1193
Действия	1196
Составление окончательного варианта программы MenuDemo	1202
Дальнейшее изучение библиотеки Swing	1208
 Часть IV. Введение в программирование ГПИ средствами JavaFX	1209
 Глава 34. Введение в JavaFX	1211
Основные понятия JavaFX	1212
Пакеты JavaFX	1212
Классы подмостков и сцены	1213
Узлы и графы сцены	1213
Компоновки	1213
Класс приложения и методы его жизненного цикла	1214
Запуск JavaFX-приложения	1214
Скелет JavaFX-приложения	1215
Компиляция и выполнение JavaFX-приложения	1218
Поток исполнения приложения	1219
Метка – простейший элемент управления в JavaFX	1219
Применение кнопок и событий	1221
Основы обработки событий в JavaFX	1222
Элемент управления экранной кнопкой	1223
Демонстрация обработки событий на примере экранных кнопок	1224
Рисование непосредственно на холсте	1227
 Глава 35. Элементы управления JavaFX	1233
Классы Image и ImageView	1233
Ввод изображения в метку	1236
Применение изображения в экранной кнопке	1238
Класс ToggleButton	1240
Класс RadioButton	1243
Обработка событий изменения в группе кнопок-переключателей	1246
Другой способ управления кнопками-переключателями	1248
Класс CheckBox	1251
Класс ListView	1254
Представление списка с полосами прокрутки	1258
Активизация режима одновременного выбора нескольких элементов из списка	1258
Класс ComboBox	1260
Класс TextField	1263
Класс ScrollPane	1265
Класс TreeView	1269

Эффекты и преобразования	1273
Эффекты	1274
Преобразования	1275
Демонстрация эффектов и преобразований	1276
Ввод всплывающих подсказок	1279
Отключение элементов управления	1280

Глава 36. Введение в меню JavaFX 1281

Основные положения о меню	1281
Краткий обзор классов MenuBar, Menu и MenuItem	1283
Класс MenuBar	1283
Класс Menu	1284
Класс MenuItem	1285
Создание главного меню	1286
Ввод мнемоники и оперативных клавиш в меню	1291
Ввод изображений в пункты меню	1293
Классы RadioMenuItem и CheckMenuItem	1294
Создание контекстного меню	1296
Создание панели инструментов	1300
Составление окончательного варианта приложения MenuDemo	1302
Дальнейшее изучение JavaFX	1308

Часть V. Применение Java 1309

Глава 37. Компоненты Java Beans 1311

Общее представление о компонентах Java Beans	1311
Преимущества компонентов Java Beans	1312
Самоанализ	1312
Шаблоны проектирования для свойств компонентов Java Beans	1313
Шаблоны проектирования для событий	1314
Методы и шаблоны проектирования	1315
Применение интерфейса BeanInfo	1315
Привязанные и ограниченные свойства	1315
Сохраняемость компонентов Java Beans	1316
Настройки	1316
Прикладной программный интерфейс Java Beans API	1317
Класс Introspector	1319
КлассPropertyDescriptor	1319
КлассEventSetDescriptor	1319
КлассMethodDescriptor	1319
Пример компонента Java Bean	1320

Глава 38. Введение в сервлеты 1323

Предпосылки для разработки сервлетов	1323
Жизненный цикл сервлета	1324
Варианты разработки сервлетов	1325
Применение контейнера сервлетов Tomcat	1325
Простой пример сервлета	1327
Создание и компиляция исходного кода сервлета	1327
Запуск контейнера сервлетов Tomcat на выполнение	1328
Запуск веб-браузера и запрос сервлета	1328

Прикладной программный интерфейс Servlet API	1329
Пакет javax.servlet	1329
Интерфейс Servlet	1330
Интерфейс ServletConfig	1330
Интерфейс ServletContext	1331
Интерфейс ServletRequest	1331
Интерфейс ServletResponse	1332
Класс GenericServlet	1333
Класс ServletInputStream	1333
Класс ServletOutputStream	1333
Класс ServletException	1333
Ввод параметров сервлета	1334
Пакет javax.servlet.http	1335
Интерфейс HttpServletRequest	1336
Интерфейс HttpServletResponse	1337
Интерфейс HttpSession	1338
Класс Cookie	1339
Класс HttpServlet	1340
Обработка HTTP-запросов и ответов HTTP	1341
Обработка HTTP-запросов типа GET	1341
Обработка HTTP-запросов типа POST	1342
Применение cookie-файлов	1343
Отслеживание сеансов связи	1346
Приложение А. Применение документирующих комментариев в Java	1347
Дескрипторы утилиты javadoc	1347
Дескриптор \$author	1348
Дескриптор {@code}	1349
Дескриптор @deprecated	1349
Дескриптор {@docRoot}	1349
Дескриптор @exception	1349
Дескриптор {@inheritDoc}	1349
Дескриптор {@link}	1350
Дескриптор {@linkplain}	1350
Дескриптор {@literal}	1350
Дескриптор @param	1350
Дескриптор @return	1350
Дескриптор @see	1350
Дескриптор @serial	1351
Дескриптор @serialData	1351
Дескриптор @serialField	1351
Дескриптор @since	1351
Дескриптор @throws	1352
Дескриптор {@value}	1352
Дескриптор @version	1352
Общая форма документирующих комментариев	1352
Результаты, выводимые утилитой javadoc	1353
Пример применения документирующих комментариев	1353
Предметный указатель	1355



Об авторе

Герберт Шилдт является автором многочисленных книг по программированию, пользующихся большим успехом у читателей в течение почти трех десятилетий, а также признанным авторитетом по языку Java. Его книги продаются миллионными тиражами и переведены на многие языки мира. Его перу принадлежит немало книг по Java, в том числе *Java: руководство для начинающих*, *Java: методики программирования Шилдта*, *SWING: руководство для начинающих*, *Искусство программирования на Java*, а также настоящее издание. Он написал немало книг и по другим языкам программирования, включая C, C++ и C#. Интересуясь всеми аспектами вычислительной техники, Герберт уделяет основное внимание языкам программирования и, в частности, компиляторам, интерпретаторам и языкам управления роботами. Он также проявляет большой интерес к стандартизации языков. Герберт окончил Илинойский университет, получив обе степени — бакалавра и магистра. Подробнее об авторе можно узнать, посетив его веб-сайт по адресу www.HerbSchildt.com.

О научном редакторе

Д-р Дэнни Ковард редактировал все издания этой книги. Он ввел понятие сервлетов Java в первую версию платформы Java EE, внедрил веб-службы на платформе Java ME, составил общую стратегию и план разработки версии Java SE 7. Он также заложил основы технологии JavaFX, а совсем недавно разработал самое крупное дополнение к стандарту Java EE 7 и прикладному программному интерфейсу Java WebSocket API. Д-р Ковард обладает необычайно обширными знаниями всех аспектов технологии Java: от программирования на Java и до разработки прикладных программных интерфейсов API вместе с опытными специалистами в данной области, а также многолетним опытом работы в исполнительном комитете Java Community Process. Он также является автором книги *JavaWebSocket Programming* и готовящейся к изданию книги по Java EE. Д-р Ковард окончил Оксфордский университет, получив степени бакалавра, магистра и доктора математических наук.

Предисловие

Java — один из самых важных и широко применяемых языков программирования в мире на протяжении многих лет. В отличие от некоторых других языков программирования, влияние Java не только не уменьшилось со временем, а, наоборот, возросло. С момента первого выпуска он выдвинулся на передний край программирования приложений для Интернета. И каждая последующая версия лишь укрепляла эту позицию. Ныне Java по-прежнему остается первым и самым лучшим языком для разработки веб-ориентированных приложений. Проще говоря, большая часть современного кода написана на Java. И это свидетельствует об особом значении языка Java для программирования.

Основная причина успеха Java — его гибкость. Начиная с первой версии 1.0, этот язык непрерывно адаптируется к изменениям в среде программирования и подходам к написанию программ. А самое главное — он не просто следует тенденциям в программировании, а *помогает их создавать*. Способность Java адаптироваться к быстрым изменениям в вычислительной технике служит основной причиной, по которой этот язык программирования продолжает оставаться столь успешным.

Со времени публикации первого издания этой книги в 1996 году она претерпела немало изменений, которые отражали последовательное развитие языка Java. Настоящее, девятое, издание обновлено по версии Java SE 8 (JDK 8). А это означает, что оно содержит немало нового материала, поскольку в версии Java SE 8 появился ряд новых языковых средств. Наиболее важными из них являются лямбда-выражения, вводящие совершенно новый синтаксис и существенно повышающие выразительную силу языка. Вследствие особого значения лямбда-выражений им посвящена отдельная глава, а примеры их применения приведены и в других главах. Внедрение лямбда-выражений повлекло за собой появление других новых языковых средств. К их числу относится библиотека потоков ввода-вывода, входящая в пакет `java.util.stream` и поддерживающая конвейерные операции с данными. Этой библиотеке также посвящена отдельная глава. Еще одним нововведением является метод по умолчанию, позволяющий вводить в интерфейс функциональные возможности, используемые по умолчанию. А такие средства, как повторяющиеся и типовые аннотации, дополнительно повышают эффективность Java. В версии Java SE 8 сделаны также значительные усовершенствования в библиотеке Java API, и некоторые из них описываются в этой книге.

Еще одним важным дополнением настоящего издания служит описание JavaFX — новой технологии Java для построения графического пользовательского интерфейса (ГПИ) приложений на Java. В перспективе JavaFX отводится существенная роль в разработке приложений на Java, и поэтому этой технологии посвящены три отдельные главы. Это означает, что программирующим на Java просто необходимо овладеть технологией JavaFX. Еще одна глава книги посвящена разработке меню средствами Swing. Несмотря на то что технология JavaFX может в перспективе полностью заменить технологию Swing, на момент выхода настоящего издания Swing по-прежнему широко применяется для построения ГПИ приложений на Java, и поэтому подробное рассмотрение этой технологии в настоящем издании вполне обоснованно. И наконец, в отдельных главах упоминаются многочисленные мелкие усовершенствования в версии Java SE 8.

Книга для всех программистов

Эта книга предназначена для всех категорий программистов: от начинающих до опытных. Начинающий программист найдет в ней подробные пошаговые описания и немало полезных примеров написания кода на Java, а углубленное рассмотрение более сложных функций и библиотек Java должно привлечь внимание опытных программистов. Для обеих категорий читателей в книге указаны действующие ресурсы и полезные ссылки.

Структура книги

Эта книга служит исчерпывающим справочным пособием по языку Java, в котором описываются его синтаксис, ключевые слова и основополагающие принципы программирования. В ней рассматривается также значительная часть библиотеки Java API. Книга разделена на пять частей, каждая из которых посвящена отдельному аспекту среды программирования Java.

В части I представлено подробное учебное пособие по языку Java. Она начинается с рассмотрения таких основных понятий, как типы данных, операции, управляющие операторы и классы. Затем описываются правила наследования, пакеты, интерфейсы, обработка исключений и многопоточная обработка. Далее рассматриваются аннотации, перечисления, автоупаковка, обобщения, операции ввода-вывода и апплеты. А заключительная глава этой части посвящена лямбда-выражениям, которые, как упоминалось ранее, являются самым важным нововведением в версии Java SE 8.

В части II описываются основные компоненты стандартной библиотеки Java API. В ней обсуждаются следующие вопросы: символьные строки, операции ввода-вывода, работа в сети, стандартные утилиты, каркас коллекций Collections Framework, апплеты, библиотека AWT, обработка событий, формирование изображений, параллельная обработка (включая каркас Fork/Join Framework), регулярные выражения и новая библиотека потоков ввода-вывода.

Часть III состоит из трех глав, посвященных технологии Swing, а часть IV – из такого же количества глав, посвященных технологии JavaFX.

Часть V состоит из двух глав с примерами практического применения Java. Сначала в ней рассматривается технология Java Beans, а затем представлены сервлеты.

Исходный код примеров, доступный в Интернете

Помните, что исходный код всех примеров, приведенных в этой книге, доступен на веб-сайте издательства Oracle Press по адресу <http://www.mhprofessional.com/product.php?isbn=0071808558> или с сайта Издательского дома “Вильямс” по адресу: <http://archive.williamspublishing.com/cgi-bin/materials.cgi?isbn=978-5-8459-1918-2>.

Особые благодарности

Выражаю особую благодарность Патрику Нотону (Patrick Naughton), Джо О'Нилу (Joe O'Neil) и Дэнни Коварду (Danny Coward).

Патрик Нотон был одним из создателей языка Java. Он помог мне в написании первого издания этой книги. Значительная часть материала глав 20, 22 и 27 была предоставлена Патриком. Его проницательность, опыт и энергия в огромной степени способствовали успеху этой книги.

При подготовке второго и третьего изданий этой книги Джо О'Нил предоставил исходные черновые материалы, которые послужили основанием для написания глав 30, 32, 37 и 38. Джо помогал мне при написании нескольких книг, и я высоко ценю его вклад.

Дэнни Ковард выполнил научное редактирование книги. Он принимал участие в работе над несколькими моими книгами, и его полезные советы, поучительные замечания и дельные предложения всегда были ценными и достойными большой признательности.

Герберт Шилдт

Дополнительная литература

Настоящее издание открывает серию книг по программированию на Java, написанных Гербертом Шилдтом. Ниже перечислены другие книги этого автора, которые могут вас заинтересовать.

- *Java: методики программирования Шилдта*. И.Д. “Вильямс”, 2008 г.
- *Java: руководство для начинающих*, 6-е изд. И.Д. “Вильямс”, 2015 г.
- *SWING: руководство для начинающих*. И.Д. “Вильямс”, 2007 г.
- *Искусство программирования на Java*. И.Д. “Вильямс”, 2005 г.

ЧАСТЬ

I

Язык Java

ГЛАВА 1

История и развитие
языка Java

ГЛАВА 2

Краткий обзор Java

ГЛАВА 3

Типы данных,
переменные и массивы

ГЛАВА 4

Операции

ГЛАВА 5

Управляющие операторы

ГЛАВА 6

Введение в классы

ГЛАВА 7

Подробное рассмотрение
классов и методов

ГЛАВА 8

Наследование

ГЛАВА 9

Пакеты и интерфейсы

ГЛАВА 10

Обработка исключений

ГЛАВА 11

Многопоточное
программирование

ГЛАВА 12

Перечисления,
автоупаковка и аннотации
(метаданные)

ГЛАВА 13

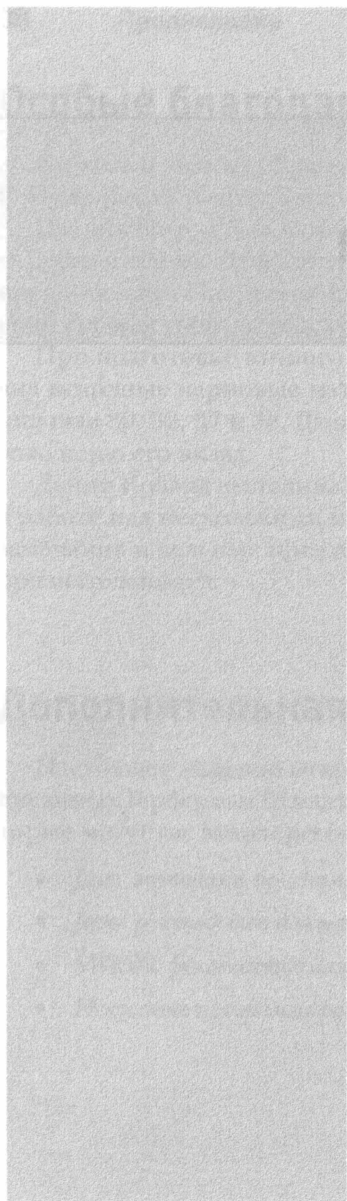
Ввод-вывод, апплеты
и прочие вопросы

ГЛАВА 14

Обобщения

ГЛАВА 15

Лямбда-выражения



ГЛАВА

1

История и развитие языка Java

Чтобы досконально изучить язык программирования Java, следует понять причины его создания, факторы, обусловившие его формирование, а также унаследованные им особенности. Подобно другим удачным языкам программирования, предшествовавшим Java, этот язык сочетает в себе лучшие элементы из своего богатого наследия и новаторские концепции, применение которых обусловлено его особым положением. В то время как остальные главы этой книги посвящены практическим вопросам программирования на Java, в том числе его синтаксису, библиотекам и приложениям, в этой главе поясняется, как и почему был разработан этот язык, что делает его столь важным и как он развивался за годы своего существования.

Несмотря на то что язык Java неразрывно связан с Интернетом, важно помнить, что это, прежде всего, язык программирования. Разработка и усовершенствование языков программирования обусловлены двумя основными причинами:

- адаптация к изменяющимся средам и областям применения;
- реализация улучшений и усовершенствований в области программирования.

Как будет показано в этой главе, разработка языка Java почти в равной мере была обусловлена обеими этими причинами.

Происхождение Java

Язык Java тесно связан с языком C++, который, в свою очередь, является прямым наследником языка C. Многие особенности Java унаследованы от обоих этих языков. От C язык Java унаследовал свой синтаксис, а многие его объектно-ориентированные свойства были перенесены из C++. Собственно говоря, ряд определяющих характеристик языка Java был перенесен — или разработан в ответ на возникшие потребности — из его предшественников. Более того, создание Java своими корнями уходит глубоко в процесс усовершенствования и адаптации, который происходит в языках программирования на протяжении нескольких последних десятилетий. Поэтому в этом разделе рассматривается последовательность со-

бытий и факторов, приведших к появлению Java. Как станет ясно в дальнейшем, каждое новшество в архитектуре языка, как правило, было обусловлено необходимостью найти решение той или иной основной проблемы, которую не могли разрешить существовавшие до этого языки. И Java не является исключением из этого правила.

Зарождение современного программирования: язык C

Язык C буквально потряс компьютерный мир. Его влияние нельзя недооценивать, поскольку он полностью изменил подход к программированию. Создание языка C было прямым следствием потребности в структурированном, эффективном и высокоуровневом языке, который мог бы заменить код ассемблера в процессе создания системных программ. Как вы, вероятно, знаете, при проектировании языка программирования часто приходится находить компромиссы между:

- простотой использования и предоставляемыми возможностями;
- безопасностью и эффективностью;
- устойчивостью и расширяемостью.

До появления языка C программистам, как правило, приходилось выбирать между языками, которые позволяли оптимизировать тот или иной ряд характеристик. Так, на языке FORTRAN можно было писать достаточно эффективные программы для научных вычислений, но он не очень подходил для написания системного кода. Аналогично язык BASIC был очень прост в изучении, но у него было не очень много функциональных возможностей, а недостаточная структурированность ставила под сомнение его полезность для написания крупных программ. На языке ассемблера можно писать очень эффективные программы, но его трудно изучать и эффективно использовать. Более того, отладка ассемблерного кода может оказаться весьма сложной задачей.

Еще одна усложнявшая дело проблема состояла в том, что первые языки программирования вроде BASIC, COBOL и FORTRAN были разработаны без учета принципов структурирования. Вместо этого основными средствами управления программой в них были операторы безусловного перехода GOTO. В результате программы, написанные на этих языках, проявляли тенденцию к появлению так называемого “макаронного кода” — множества запутанных переходов и условных ветвей, которые делали программу едва ли доступной для понимания. И хотя языки наподобие Pascal структурированы, они не были предназначены для написания эффективных программ и лишены ряда важных средств, необходимых для написания разнообразных программ. (В частности, применять Pascal для написания системного кода было нецелесообразно, принимая во внимание наличие нескольких стандартных диалектов этого языка.)

Таким образом, до изобретения языка C, несмотря на затраченные усилия, ни одному языку не удавалось разрешить существовавшие в то время противоречия в программировании. Вместе с тем потребность в таком языке становилась все более насущной. В начале 1970-х годов началась компьютерная революция, и потребность в программном обеспечении быстро превысила возможности программистов.

стов по его созданию. В академических кругах большие усилия были приложены к созданию более совершенного языка программирования. Но в то же время стало все больше ощущаться влияние еще одного, очень важного фактора. Компьютеры, наконец, получили достаточно широкое распространение, чтобы была достигнута “критическая масса”. Компьютерное оборудование больше не находилось за закрытыми дверями. Впервые программисты получили буквально неограниченный доступ к своим вычислительным машинам. Это дало свободу для экспериментирования. Программисты смогли также приступить к созданию своих собственных инструментальных средств. Накануне создания C произошел качественный скачок в области языков программирования.

Язык C, изобретенный и впервые реализованный Деннисом Ритчи на мини-ЭВМ DEC PDP-11, работавшей под управлением операционной системы UNIX, явился результатом процесса разработки, начавшегося с предшествующего ему языка BCPL, разработанного Мартином Ричардсом. BCPL оказал влияние на язык, получивший название B, который был изобретен Кеном Томпсоном и в начале 1970-х гг. привел к появлению языка C. В течение долгих лет фактическим стандартом языка C была его версия, которая поставлялась вместе с операционной системой UNIX (она описана в книге *Язык программирования C* Брайана Кернигана и Денниса Ритчи, 2-е издание, И.Д. “Вильямс”, 2007 г.). Язык C был формально стандартизован в декабре 1989 г., когда Национальный институт стандартизации США (American National Standards Institute – ANSI) принял стандарт на C.

Многие считают создание C началом современного этапа развития языков программирования. Этот язык успешно соединил противоречивые свойства, которые доставляли столько неприятностей в предыдущих языках. В итоге появился мощный, эффективный, структурированный язык, изучение которого было сравнительно простым. Кроме того, ему была присуща еще одна, почти непостижимая особенность: он был разработан *специально* для программистов. До появления C языки программирования проектировались в основном в качестве академических упражнений или бюрократическими организациями. Иное дело – язык C. Он был спроектирован, реализован и разработан практикующими программистами и отражал их подход к программированию. Его функции были отлажены, проверены и многократно переработаны людьми, которые действительно пользовались этим языком. Таким образом, получился язык, который сразу же понравился программистам. И действительно, язык C быстро приобрел много приверженцев, которые едва ли не молились на него. Благодаря этому C получил быстрое и широкое признание среди программистов. Короче говоря, C – это язык, разработанный программистами для программистов. Как станет ясно в дальнейшем, эту замечательную особенность унаследовали язык Java.

Следующий этап: язык C++

В конце 1970-х–начале 1980-х гг. язык C стал господствующим языком программирования и продолжает широко применяться до сих пор. А если C – удачный и удобный язык, то может возникнуть вопрос: чем обусловлена потребность в каком-то другом языке? Ответ состоит в постоянно растущей *сложности* программ. На протяжении всей истории развития программирования постоянно

растущая сложность программ порождала потребность в более совершенных способах преодоления их сложности. Язык C++ явился ответом на эту потребность. Чтобы лучше понять, почему потребность преодоления сложности программ является главной побудительной причиной создания языка C++, рассмотрим следующие факторы.

С момента изобретения компьютеров подходы к программированию коренным образом изменились. Когда компьютеры только появились, программирование осуществлялось изменением двоичных машинных инструкций вручную с панели управления компьютера. До тех пор, пока длина программ не превышала нескольких сотен инструкций, этот подход был вполне приемлем. В связи с разрастанием программ был изобретен язык ассемблера, который позволил программистам работать с более крупными и все более сложными программами, используя символьные представления машинных инструкций. По мере того как программы продолжали увеличиваться в объеме, появились языки высокого уровня, которые предоставили программистам дополнительные средства для преодоления сложности программ.

Первым языком программирования, который получил широкое распространение, был, конечно же, FORTRAN. Хотя он и стал первым впечатляющим этапом в программировании, его вряд ли можно считать языком, который способствует созданию ясных и простых для понимания программ. 1960-е годы ознаменовались зарождением *структурного программирования*. Эта методика программирования наиболее ярко проявилась в таких языках, как С. Пользуясь структурированными языками, программисты впервые получили возможность без особых затруднений создавать программы средней сложности. Но и методика структурного программирования уже не позволяла программистам справиться со сложными проектами, когда они достигали определенных масштабов. К началу 1980-х. сложность многих проектов начала превышать предел, позволявший справиться с ними, применяя структурный подход. Для решения этой проблемы была изобретена новая методика программирования, получившая название *объектно-ориентированного программирования* (ООП). Объектно-ориентированное программирование подробно рассматривается в последующих главах, а здесь приводится лишь краткое его определение: ООП — это методика программирования, которая помогает организовывать сложные программы, применяя принципы наследования, инкапсуляции и полиморфизма.

Из всего сказанного выше можно сделать следующий вывод: несмотря на то, что С является одним из лучших в мире языков программирования, существует предел его способности справляться со сложностью программ. Как только размеры программы превышают определенную величину, она становится слишком сложной, чтобы ее можно было охватить как единое целое. Точная величина этого предела зависит как от структуры самой программы, так и от подходов, используемых программистом, но начиная с определенного момента любая программа становится слишком сложной для понимания и внесения изменений, а следовательно, неуправляемой. Язык C++ предоставил возможности, которые позволили программистам преодолеть этот порог сложности, чтобы понимать крупные программы и управлять ими.

Язык C++ был изобретен Бьярне Страуструпом (Bjarne Stroustrup) в 1979 г., когда он работал в компании Bell Laboratories в городе Мюррей-Хилл, шт. Нью-

Джерси. Вначале Страуструп назвал новый язык “C with Classes” (C с классами). Но в 1983 г. это название было изменено на C++. Язык C++ расширяет функциональные возможности языка C, добавляя в него объектно-ориентированные свойства. А поскольку язык C++ построен на основе C, то в нем поддерживаются все функциональные возможности, свойства и преимущества C. Это обстоятельство явилось главной причиной успешного распространения C++ в качестве языка программирования. Изобретение языка C++ не было попыткой создать совершенно новый язык программирования. Напротив, все усилия были направлены на усовершенствование уже существующего очень удачного языка.

Предпосылки к созданию Java

К концу 1980-х и в начале 1990-х гг. объектно-ориентированное программирование на C++ стало основной методикой программирования. И в течение некоторого, хотя и непродолжительного периода времени казалось, что программисты наконец изобрели идеальный язык. Ведь язык C++ сочетал в себе высокую эффективность и стилистические элементы языка C наряду с объектно-ориентированным подходом, и поэтому его можно было использовать для создания самых разных программ. Но, как и прежде, уже вызревали факторы, которые должны были в очередной раз стимулировать развитие языков программирования. Прошло еще несколько лет, и развитие Всемирной паутины и Интернета достигло “критической массы”, что и привело к еще одной революции в программировании.

Создание языка Java

Начало разработке языка Java было положено в 1991 г. Джеймсом Гослингом (James Gosling), Патриком Нотоном (Patrick Naughton), Крисом Уартом (Chris Warth), Эдом Франком (Ed Frank) и Майком Шериданом (Mike Sheridan), работавшими в компании Sun Microsystems, Inc. Разработка первой рабочей версии заняла 18 месяцев. Вначале язык получил название “Oak” (Дуб), но в 1995 г. он был переименован в “Java”. Между первой реализацией языка Oak в конце 1992 г. и публичным объявлением о создании Java весной 1995 г. в разработке и развитии этого языка приняли участие немало других специалистов. В частности, Билл Джой (Bill Joy), Артур ван Хофф (Arthur van Hoff), Джонатан Пэйн (Jonathan Payne), Франк Йеллин (Frank Yellin) и Тим Линдхольм (Tim Lindholm) внесли основной вклад в развитие исходного прототипа Java.

Как ни странно, первоначальной побудительной причиной для создания языка Java послужил совсем не Интернет, а потребность в независимом от конкретной платформы (т.е. архитектурно нейтральном) языке, который можно было бы использовать для создания программного обеспечения, встраиваемого в различные бытовые электронные устройства, в том числе микроволновые печи и устройства дистанционного управления. Не трудно догадаться, что в контроллерах таких устройств применяются разнотипные процессоры. Но трудность применения C и C++ (как и большинства других языков) состоит в том, что написанные на них программы должны компилироваться для конкретной платформы. И хотя про-

граммы на C++ могут быть скомпилированы практически для любого типа процессора, тем не менее, для этого потребуются наличие полного компилятора C++, предназначенного для данного конкретного процессора. Но дело в том, что создание компиляторов обходится дорого и отнимает немало времени. Поэтому требовалось более простое и экономически выгодное решение. Пытаясь найти такое решение, Гослинг и другие разработчики начали работу над переносимым, не зависящим от конкретной платформы языком, который можно было бы использовать для создания кода, пригодного для выполнения на разнотипных процессорах в различных средах. И в конечном итоге их усилия привели к созданию языка Java.

Примерно в то время, когда определялись основные характеристики языка Java, появился второй, еще более важный фактор, который должен был сыграть решающую роль в судьбе этого языка. И этим вторым фактором, конечно же, стала Всемирная паутина (веб). Если бы формирование веб не происходило почти одновременно с реализацией языка Java, он мог бы остаться полезным, но незамеченным языком программирования бытовых электронных устройств. Но с появлением веб язык Java вышел на передний край разработки языков программирования, поскольку среда веб также нуждалась в переносимых программах.

С самых первых шагов своей карьеры большинству программистов приходится твердо усваивать, что переносимость программ столь же недостижима, сколь и желательна. Несмотря на то что потребность в средствах для создания эффективных, переносимых (не зависящих от конкретной платформы) программ существовала в программировании с самого начала, она отодвигалась на задний план другими, более насущными, задачами. Более того, большая часть отрасли вычислительной техники была разделена на три конкурирующих лагеря (Intel, Macintosh и UNIX), и поэтому большинство программистов оставались на своих закрепленных аппаратно-программных рубежах, что несколько снижало потребность в переносимом коде. Тем не менее с появлением Интернета и веб старая проблема переносимости возникла снова, а ее решение стало еще более актуальным. Ведь Интернет представляет собой разнообразную и распределенную сетевую среду с разнотипными компьютерами, операционными системами и процессорами. Несмотря на то что к Интернету подключены разнотипные платформы, пользователям желательно, чтобы на всех этих платформах можно было выполнять одинаковые программы. То, что в начале было неприятной, но не слишком насущной задачей, стало потребностью первостепенной важности.

К 1993 г. членам группы разработчиков Java стало очевидно, что проблемы переносимости, часто возникающие при создании кода, предназначенного для встраивания в контроллеры, возникают также и при попытках создания кода для Интернета. Фактически та же проблема, для решения которой в мелком масштабе предназначался язык Java, была актуальна в большем масштабе и в среде Интернета. Понимание этого обстоятельства вынудило разработчиков Java перенести свое внимание с бытовой электроники на программирование для Интернета. Таким образом, Интернет обеспечил крупномасштабный успех Java, несмотря на то, что потребность в архитектурно нейтральном языке программирования послужила для этого своего рода “первоначальной искрой”.

Как упоминалось ранее, язык Java наследует многие из своих характеристик от языков C и C++. Это сделано намеренно. Разработчики Java знали, что использование зна-

когого синтаксиса С и повторение объектно-ориентированных свойств С++ должно было сделать их язык привлекательным для миллионов, имеющих опыт программирования на С/С++. Помимо внешнего сходства, в Java используется ряд других характерных особенностей, которые способствовали успеху языков С и С++. Во-первых, язык Java был разработан, проверен и усовершенствован людьми, имевшими солидный практический опыт программирования. Этот язык разработан с учетом потребностей и опыта его создателей. Таким образом, Java — это язык для программистов. Во-вторых, Java целостен и логически непротиворечив. В-третьих, Java предоставляет программисту полный контроль над программой, если не учитывать ограничения, накладываемые средой Интернета. Если программирование выполняется грамотно, это непосредственно отражается на самих программах. В равной степени справедливо и обратное. Иначе говоря, Java — это язык не для обучающихся программированию, а для тех, кто занимается им профессионально.

Из-за сходства характеристик языков Java и С некоторые склонны считать Java просто “версией языка С++ для Интернета”. Но это серьезное заблуждение. Языку Java присущи значительные практические и концептуальные отличия. Язык С++ действительно оказал влияние на характеристики Java, но Java не является усовершенствованной версией С++. Например, Java несовместим с языком С++. Разумеется, у него немало сходств с языком С++, и в исходном коде программы на Java программирующий на С++ будет чувствовать себя почти как дома. Вместе с тем язык Java предназначен не для замены С++, а для решения одних задач, тогда как язык С++ — для решения других. Оба этих языка будут еще долго сосуществовать.

Как отмечалось в начале главы, развитие языков программирования обусловлено двумя причинами: адаптацией к изменениям в среде и реализацией новых идей в области программирования. Изменения в среде, которые побудили к разработке языка, подобного Java, вызвали потребность в независимых от платформы программах, предназначенных для распространения в Интернете. Но язык Java изменяет также подход к написанию программ. В частности, язык Java углубил и усовершенствовал объектно-ориентированный подход, использованный в С++, добавил в него поддержку многопоточной обработки и предоставил библиотеку, которая упростила доступ к Интернету. Но столь поразительный успех Java обусловлен не отдельными его особенностями, а их совокупностью как языка в целом. Он явился прекрасным ответом на потребности в то время лишь зарождающейся среды в высшей степени распределенных вычислительных систем. В области разработки программ для Интернета язык Java стал тем, чем язык С стал для системного программирования: революционной силой, которая изменила мир.

Связь с языком С#

Многообразие и большие возможности языка Java продолжают оказывать влияние на всю разработку языков программирования. Многие из его новаторских характеристик, конструкций и концепций становятся неотъемлемой частью основы любого нового языка. Успех Java слишком значителен, чтобы его можно было игнорировать.

Вероятно, наиболее наглядным примером влияния языка Java на программирование служит язык С#. Он создан в корпорации Microsoft для поддержки платфор-

мы .NET Framework и тесно связан с Java. В частности, в обоих этих языках используется один и тот же общий синтаксис, поддерживается распределенное программирование и применяется одна и та же объектная модель. Разумеется, у Java и C# имеется целый ряд отличий, но в целом они внешне очень похожи. Ныне такое “перекрестное опыление” Java и C# служит наилучшим доказательством того, что язык Java коренным образом изменил представление о языках программирования и их применении.

Каким образом язык Java изменил Интернет

Интернет способствовал выдвижению Java на передний край программирования, а язык Java, в свою очередь, оказал очень сильное влияние на Интернет. Язык Java не только упростил создание программ для Интернета в целом, но и обусловил появление нового типа прикладных программ, предназначенных для работы в сети и получивших название апплетов, которые изменили понятие содержимого сетевой среды. Кроме того, язык Java позволил решить две наиболее острые проблемы программирования, связанные с Интернетом: переносимость и безопасность. Рассмотрим каждую из этих проблем в отдельности.

Апплеты на Java

Апплет — это особый вид прикладной программы на Java, предназначенный для передачи через Интернет и автоматического выполнения в совместимом с Java веб-браузере. Более того, апплет загружается по требованию, не требуя дальнейшего взаимодействия с пользователем. Если пользователь щелкает кнопкой мыши на ссылке, которая содержит апплет, он автоматически загружается и запускается в браузере. Апплеты создаются в виде небольших программ. Как правило, они служат для отображения данных, предоставляемых сервером, обработки действий пользователя или выполнения локально, а не на сервере, таких простых функций, как вычисление процентов по кредитам. По существу, апплет позволяет перенести ряд функций со стороны сервера на сторону клиента.

Появление апплетов изменило характер программирования приложений для Интернета, поскольку они расширили совокупность объектов, которые можно свободно перемещать в киберпространстве. В общем, между сервером и клиентом передаются две крупные категории объектов: пассивные данные и динамически активизируемые программы. Например, чтение сообщений электронной почты подразумевает просмотр пассивных данных. Даже при загрузке программы ее код по-прежнему остается пассивными данными до тех пор, пока он не начнет выполняться. И напротив, апплет представляет собой динамическую, автоматически выполняющуюся прикладную программу. Такая программа является активным агентом на клиентской машине, хотя она иницируется сервером.

Но за привлекательностью динамических сетевых программ скрываются серьезные трудности обеспечения безопасности и переносимости. Очевидно, что клиентскую машину нужно обезопасить от нанесения ей ущерба программой, которая сначала загружается в нее, а затем автоматически запускается на выполнение.

Кроме того, такая программа должна быть в состоянии выполняться в различных вычислительных средах и под управлением разных операционных систем. Как станет ясно в дальнейшем, эти трудности эффективно и изящно решаются в Java. Рассмотрим их подробнее.

Безопасность

Вам, вероятно, известно, что каждая загрузка обычной программы сопряжена с риском, поскольку загружаемый код может содержать вирус, “тroyанский конь” или вредоносный код. Дело в том, что вредоносный код может сделать свое черное дело, если получит несанкционированный доступ к системным ресурсам. Например, просматривая содержимое локальной файловой системы компьютера, вирусная программа может собирать конфиденциальную информацию вроде номеров кредитных карточек, сведений о состоянии банковских счетов и паролей. Для безопасной загрузки и выполнения апплетов Java на клиентской машине нужно было предотвратить подобные атаки со стороны апплетов.

Java обеспечивает такую защиту, ограничивая действие апплета исполняющей средой Java и не предоставляя ему доступ к другим частям операционной системы компьютера. (Способы достичь этого рассматриваются далее.) Возможность загружать апплеты в полной уверенности, что это не нанесет системе никакого вреда и не нарушит ее безопасность, многие специалисты и пользователи считают наиболее новаторским аспектом Java.

Переносимость

Переносимость — основная особенность Интернета, поскольку эта глобальная сеть соединяет вместе множество разнотипных компьютеров и операционных систем. Чтобы программа на Java могла выполняться практически на любом компьютере, подключенном к Интернету, требуется каким-то образом обеспечить ее выполнение в разных системах. В частности, один и тот же апплет должен иметь возможность загружаться и выполняться на широком спектре процессоров, операционных систем и браузеров, подключенных к Интернету. А создавать разные версии апплетов для разнотипных компьютеров совершенно нерационально. *Один и тот же* код должен работать на *всех* компьютерах. Поэтому требовался какой-то механизм для создания переносимого исполняемого кода. Как станет ясно в дальнейшем, тот же самый механизм, который обеспечивает безопасность, способствует и созданию переносимых программ.

Чудо Java: байт-код

Основная особенность Java, которая позволяет решать описанные выше проблемы обеспечения безопасности и переносимости программ, состоит в том, что компилятор Java выдает не исполняемый код, а так называемый *байт-код* — в высшей степени оптимизированный набор инструкций, предназначенных для выполнения в исполняющей системе Java, называемой *виртуальной машиной Java* (Java

Virtual Machine — JVM). Собственно говоря, первоначальная версия виртуальной машины JVM разрабатывалась в качестве *интерпретатора байт-кода*. Это может вызывать недоумение, поскольку для обеспечения максимальной производительности компиляторы многих современных языков программирования призваны создавать исполняемый код. Но то, что программа на Java интерпретируется виртуальной машиной JVM, как раз помогает решить основные проблемы разработки программ для Интернета. И вот почему.

Трансляция программы Java в байт-код значительно упрощает ее выполнение в разнотипных средах, поскольку на каждой платформе необходимо реализовать только виртуальную машину JVM. Если в отдельной системе имеется исполняющий пакет, в ней можно выполнять любую программу на Java. Следует, однако, иметь в виду, что все виртуальные машины JVM на разных платформах, несмотря на некоторые отличия и особенности их реализации, способны правильно интерпретировать один и тот же байт-код. Если бы программа на Java компилировалась в машинозависимый код, то для каждого типа процессоров, подключенных к Интернету, должны были бы существовать отдельные версии одной и той же программы. Ясно, что такое решение неприемлемо. Таким образом, организация выполнения байт-кода виртуальной машиной JVM — простейший способ создания по-настоящему переносимых программ.

Тот факт, что программа на Java выполняется виртуальной машиной JVM, способствует также повышению ее безопасности. Виртуальная машина JVM управляет выполнением программы, поэтому она может изолировать программу и воспрепятствовать возникновению побочных эффектов от ее выполнения за пределами данной системы. Как станет ясно в дальнейшем, ряд ограничений, существующих в языке Java, также способствует повышению безопасности.

В общем, когда программа компилируется в промежуточную форму, а затем интерпретируется виртуальной машиной JVM, она выполняется медленнее, чем если бы она была скомпилирована в исполняемый код. Но в Java это отличие в производительности не слишком заметно. Байт-код существенно оптимизирован, и поэтому его применение позволяет виртуальной машине JVM выполнять программы значительно быстрее, чем следовало ожидать.

Язык Java был задуман как интерпретируемый, но ничто не препятствует ему оперативно выполнять компиляцию байт-кода в машинозависимый код для повышения производительности. Поэтому вскоре после выпуска Java появилась технология HotSpot, которая предоставляет *динамический* компилятор (или так называемый *JIT-компилятор*) байт-кода. Если динамический компилятор входит в состав виртуальной машины JVM, то избранные фрагменты байт-кода компилируются в исполняемый код по частям, в реальном времени и по требованию. Важно понимать, что одновременная компиляция всей программы Java в исполняемый код нецелесообразна, поскольку Java производит различные проверки, которые могут быть сделаны только во время выполнения. Вместо этого динамический компилятор компилирует код во время выполнения по мере надобности. Более того, компилируются не все фрагменты байт-кода, а только те, которым компиляция принесет выгоду, а остальной код просто интерпретируется. Тем не менее принцип динамической компиляции обеспечивает значительное повышение производительности. Даже при динамической компиляции байт-кода характеристики

переносимости и безопасности сохраняются, поскольку виртуальная машина JVM по-прежнему отвечает за целостность исполняющей среды.

Сервлеты: серверные программы на Java

Как ни полезны апплеты, они решают лишь половину задачи в архитектуре “клиент-сервер”. Вскоре после появления языка Java стало очевидно, что он может пригодиться и на серверах. В результате появились сервлеты. *Сервлет* — это небольшая прикладная программа, выполняемая на сервере. Подобно тому как апплеты динамически расширяют функциональные возможности веб-браузера, сервлеты динамически расширяют функциональные возможности веб-сервера. Таким образом, с появлением сервлетов язык Java охватил обе стороны соединения “клиент-сервер”.

Сервлеты служат для создания динамически генерируемого содержимого, которое затем предоставляется клиенту. Например, интерактивный склад может использовать сервлет для поиска стоимости товара в базе данных. Затем информация о цене используется для динамического создания веб-страницы, отправляемой браузеру. И хотя динамически создаваемое содержимое доступно и с помощью таких механизмов, как CGI (Common Gateway Interface — интерфейс общего шлюза), применение сервлетов дает ряд преимуществ, в том числе повышение производительности.

Подобно всем программам на Java, сервлеты компилируются в байт-код и выполняются виртуальной машиной JVM, поэтому они в высшей степени переносимы. Следовательно, один и тот же сервлет может применяться в различных серверных средах. Единственным необходимым условием для этого является поддержка на сервере виртуальной машины JVM и контейнера для сервлета.

Отличительные особенности Java

Рассмотрение истории создания и развития языка Java было бы неполным без описания особенностей, характерных для Java. Основным фактором, обусловившим изобретение Java, стала потребность в обеспечении переносимости и безопасности, но свою роль в формировании окончательной версии языка сыграли и другие факторы. Группа разработчиков обобщила основные понятия Java и составила следующий перечень его особенностей:

- простота;
- безопасность;
- переносимость;
- объектная ориентированность;
- надежность;
- многопоточность;
- архитектурная нейтральность;

- интерпретируемость;
- высокая производительность;
- распределенность;
- динамичность.

Мы уже рассмотрели такие особенности, как безопасность и переносимость. А теперь поясним значение остальных элементов этого списка.

Простота

Язык Java был задуман как простой в изучении и эффективный в употреблении профессиональными программистами. Овладеть языком Java тем, у кого имеется некоторый опыт программирования, не составит особого труда. Если же вы уже знакомы с основными принципами объектно-ориентированного программирования, то изучить Java вам будет еще проще. А от тех, кто имеет опыт программирования на C++, переход к Java вообще потребует минимум усилий. Язык Java наследует синтаксис C/C++ и многие объектно-ориентированные свойства C++, поэтому для большинства программистов изучение Java не составит больших трудностей.

Объектная ориентированность

Хотя предшественники языка Java и оказали влияние на его архитектуру и синтаксис, при его проектировании не ставилась задача совместимости по исходному коду с каким-нибудь другим языком. Это позволило группе разработчиков создавать Java, по существу, с чистого листа. Одним из следствий этого явился четкий, практичный, прагматичный подход к объектам. Помимо того, что язык Java позаимствовал свойства многих удачных объектно-программных сред, разработанных на протяжении нескольких последних десятилетий, в нем удалось достичь золотой середины между строгим соблюдением принципа “все элементы программы являются объектами” и более прагматичного принципа “прочь с дороги”. Объектная модель Java проста и легко расширяема. В то же время такие элементарные типы данных, как целочисленные, сохраняются в виде высокопроизводительных компонентов, не являющихся объектами.

Надежность

Многоплатформенная среда веб предъявляет к программам повышенные требования, поскольку они должны надежно выполняться в разнотипных системах. Поэтому способность создавать надежные программы была одним из главных приоритетов при разработке Java. Для обеспечения надежности в Java накладывается ряд ограничений в нескольких наиболее важных областях, что вынуждает программистов выявлять ошибки на ранних этапах разработки программы. В то же время Java избавляет от необходимости беспокоиться по поводу многих наиболее часто встречающихся ошибок программирования. А поскольку Java — строго типизированный язык, то проверка кода выполняется во время компиляции. Но проверка кода де-

ляется и во время выполнения. В результате многие трудно обнаруживаемые программные ошибки, которые часто приводят к возникновению с трудом воспроизводимых ситуаций во время выполнения, попросту невозможны в программе на Java. Предсказуемость кода в разных ситуациях — одна из основных особенностей Java.

Чтобы понять, каким образом достигается надежность программ на Java, рассмотрим две основные причины программных сбоев: ошибки управления памятью и неправильная обработка исключений (т.е. ошибки при выполнении). В традиционных средах создания программ управление памятью — сложная и трудоемкая задача. Например, в среде C/C++ программист должен вручную резервировать и освобождать всю динамически распределяемую память. Иногда это ведет к возникновению трудностей, поскольку программисты забывают освободить ранее зарезервированную память или, что еще хуже, пытаются освободить область памяти, все еще используемую другой частью кода. Java полностью исключает такие ситуации, автоматически управляя резервированием и освобождением памяти. (Освобождение оперативной памяти полностью выполняется автоматически, поскольку Java предоставляет средства сборки неиспользуемых объектов в “мусор”.) В традиционных средах условия для исключений часто возникают в таких ситуациях, как деление на ноль или отсутствие искомого файла, а управление ими должно осуществляться с помощью громоздких и трудных для понимания конструкций. Java облегчает выполнение этой задачи, предлагая объектно-ориентированный механизм обработки исключений. В грамотно написанной программе на Java все ошибки при выполнении могут (и должны) обрабатываться самой программой.

Многопоточность

Язык Java был разработан в ответ на потребность создавать интерактивные сетевые программы. Для этой цели в Java поддерживается написание многопоточных программ, способных одновременно выполнять многие действия. Исполняющая система Java содержит изящное, но вместе с тем сложное решение задачи синхронизации многих процессов, которое позволяет строить устойчиво работающие интерактивные системы. Простой подход к организации многопоточной обработки, реализованный в Java, позволяет программистам сосредоточивать основное внимание на конкретном поведении программы, а не на создании многозадачной подсистемы.

Архитектурная нейтральность

Основной задачей, которую ставили перед собой разработчики Java, было обеспечение долговечности и переносимости кода. Одной из главных трудностей, стоявших перед разработчиками, когда они создавали Java, было отсутствие всяких гарантий, что код, написанный сегодня, будет успешно выполняться завтра — даже на одном и том же компьютере. Операционные системы и процессоры постоянно совершенствуются, и любые изменения в основных системных ресурсах могут стать причиной неработоспособности программ. Пытаясь каким-то образом изменить это положение, разработчики приняли ряд жестких решений в самом языке и виртуальной машине Java. Они поставили перед собой следующую цель:

“написано однажды, выполняется везде, в любое время и всегда”. И эта цель была в значительной степени достигнута.

Интерпретируемость и высокая производительность

Как упоминалось ранее, выполняя компиляцию программ в промежуточное представление, называемое байт-кодом, Java позволяет создавать межплатформенные программы. Такой код может выполняться в любой системе, на которой реализована виртуальная машина JVM. С первых же попыток разработать межплатформенные решения удалось достичь поставленной цели, хотя и за счет снижения производительности. Как пояснялось ранее, байт-код Java был тщательно разработан таким образом, чтобы его можно было с высокой эффективностью преобразовывать непосредственно в машиннозависимый код на конкретной платформе с помощью динамического компилятора. Исполняющие системы Java, обеспечивающие такую возможность, сохраняют все преимущества кода, не зависящего от конкретной платформы.

Распределенность

Язык Java предназначен для распределенной среды Интернета, поскольку он поддерживает семейство сетевых протоколов TCP/IP. По существу, обращение к ресурсу по унифицированному указателю информационного ресурса (URL) мало чем отличается от обращения к файлу. В Java поддерживается также *удаленный вызов методов* (RMI — Remote Method Invocation). Такая возможность позволяет вызывать методы из программ через сеть.

Динамичность

Программы на Java содержат значительный объем данных динамического типа, используемых для проверки полномочий и разрешения доступа к объектам во время выполнения. Это позволяет безопасно и рационально выполнять динамическое связывание кода. Данное обстоятельство исключительно важно для устойчивости среды Java, где небольшие фрагменты байт-кода могут динамически обновляться в действующей системе.

Эволюция языка Java

Первоначальная версия Java не представляла собой ничего особенно революционного, но она и не ознаменовала завершение периода быстрого совершенствования этого языка. В отличие от большинства других систем программирования, совершенствование которых происходило понемногу и постепенно, язык Java продолжает стремительно развиваться. Вскоре после выпуска версии Java 1.0 разработчики уже создали версию Java 1.1. Внедренные в этой версии функциональные возможности оказались более значительными и существенными, чем можно было ожидать, судя по увеличению дополнительного номера версии. В новой версии

разработчики добавили много новых библиотечных элементов, переопределили механизм обработки событий и изменили конфигурацию многих библиотечных средств из версии 1.0. Кроме того, они отказались от нескольких средств, первоначально определенных в Java 1.0, но признанных затем устаревшими. Таким образом, в версии Java 1.1 были внедрены новые характерные свойства и в то же время исключены некоторые свойства, определенные в первоначальной спецификации.

Следующей основной стала версия Java 2, где номер “2” означает второе поколение. Создание версии Java 2 стало знаменательным событием, означавшим начало “современной эпохи” Java. Первой версии Java 2 был присвоен номер 1.2, что может показаться несколько странным. Дело в том, что вначале номер относился к внутреннему номеру версии библиотек Java, но затем он был распространен на всю версию в целом. С появлением версии Java 2 компания Sun Microsystems начала выпускать программное обеспечение Java в виде пакета J2SE (Java 2 Platform Standard Edition – Стандартная версия платформы Java 2), и с тех пор номера версий стали присваиваться именно этому программному продукту.

В версии Java 2 была добавлена поддержка ряда новых средств, в том числе Swing и Collections Framework. Кроме того, были усовершенствованы виртуальная машина JVM и различные инструментальные средства программирования. В то же время версия Java 2 содержала некоторые не рекомендованные к употреблению средства. Наибольшие изменения претерпел класс Thread, где методы `suspend()`, `resume()` и `stop()` стали рекомендованными к употреблению в многопоточном программировании.

Версия J2SE 1.3 стала первой серьезной модернизацией первоначальной версии Java J2SE. Эта модернизация состояла, главным образом, в расширении существующих функциональных возможностей и “уплотнении” среды разработки. В общем случае программы, написанные для версий 1.2 и 1.3, совместимы по исходному коду. И хотя изменений в версии 1.3 оказалось меньше, чем в трех предшествующих основных версиях, это не сделало ее менее важной.

Совершенствование языка Java было продолжено в версии J2SE 1.4. Эта версия содержала несколько важных усовершенствований и добавлений. Например, в нее были добавлены новое ключевое слово `assert`, цепочки исключений и подсистема ввода-вывода на основе каналов. Изменения были внесены и в каркас Collections Framework, а также в классы, предназначенные для работы в сети. В эту версию было также внесено много мелких изменений. Несмотря на значительное количество новых функциональных возможностей, версия 1.4 почти полностью сохранила совместимость по исходному коду с предшествующими версиями.

В следующую версию Java, названную J2SE 5, был внесен ряд революционных изменений. В отличие от большинства предшествующих обновлений Java, которые предоставляли важные, но постепенные усовершенствования, в версии J2SE 5 была коренным образом расширена область, возможности и пределы применения языка. Чтобы стало понятнее, насколько существенными оказались изменения, внесенные в версии J2SE 5, ниже перечислены лишь самые основные из новых функциональных возможностей Java в данной версии.

- Обобщения.
- Аннотации.

- Автоупаковка и автораспаковка.
- Перечисления.
- Усовершенствованный цикл `for` в стиле `for each`.
- Аргументы переменной длины (`varargs`).
- Статический импорт.
- Форматированный ввод-вывод.
- Утилиты параллельной обработки.

В этом перечне не указаны незначительные изменения или постепенные усовершенствования. Каждый пункт перечня представлял значительное дополнение языка Java. Одни из них, в том числе обобщения, усовершенствованный цикл `for` и аргументы переменной длины, представляли новые синтаксические элементы, а другие, например автоупаковка и автораспаковка, изменяли семантику языка. Аннотации придали программированию совершенно новые черты. В любом случае влияние всех этих дополнений вышло за рамки их прямого действия. Они полностью изменили сам характер языка Java.

Значение новых функциональных возможностей нашло отражение в присвоенном данной версии номере — “5”. Если следовать привычной логике, то следующим номером версии Java должен был быть 1.5. Однако новые средства оказались столь значительными, что переход от версии 1.4 к версии 1.5 не отражал бы масштабы внесенных изменений. Поэтому в компании Sun Microsystems решили присвоить новой версии номер 5, чтобы тем самым подчеркнуть значимость этого события. Поэтому версия данного программного продукта была названа J2SE 5, а комплект разработчика — JDK 5. Тем не менее для сохранения единообразия в компании Sun Microsystems решили использовать номер 1.5 в качестве внутреннего номера версии, называемого также номером *версии разработки*. Цифра 5 в обозначении версии называется номером *версии продукта*.

Следующая версия получила название Java SE 6. С выходом этой версии в компании Sun Microsystems решили в очередной раз изменить название платформы Java. В названии была опущена цифра 2. Таким образом, данная платформа теперь называется сокращенно *Java SE*, а официально — *Java Platform, Standard Edition 6* (Платформа Java, стандартная версия 6). Комплект разработчика был назван JDK 6. Как и в обозначении версии J2SE 5, цифра 6 в названии Java SE 6 означает номер версии продукта. Внутренним номером разработки этой версии является 1.6.

Версия Java SE 6 была построена на основе версии J2SE 5 с рядом дальнейших усовершенствований. Она не содержала дополнений к числу основных языковых средств Java, но расширяла библиотеки API, добавляя несколько новых пакетов и предоставляя ряд усовершенствований в исполняющей системе. В этой версии было сделано еще несколько усовершенствований и внесено несколько дополнений. В целом версия Java SE 6 призвана закрепить достижения в J2SE 5.

Далее была выпущена версия Java SE 7 с комплектом разработчика Java JDK 7 и внутренним номером версии 1.7. Это был первый главный выпуск Java с тех пор, как компания Sun Microsystems была приобретена компанией Oracle (этот процесс начался в апреле 2009 года и завершился в январе 2010 года). Версия Java SE 7 со-

держит много новых средств, включая существенные дополнения языка и библиотек Java API. В данной версии была усовершенствована исполняющая система Java, а также включена в нее поддержка других языков, кроме Java, но программирующих на Java больше интересовали дополнения, внесенные в сам язык и его библиотеку. Новые языковые средства были разработаны в рамках проекта *Project Coin*. Этот проект должен был обозначать многие, так называемые “мелкие” изменения в Java, которые предполагалось включить в JDK 7, хотя эффект от всех этих новых “мелких” новшеств весьма велик с точки зрения их воздействия на код. В действительности для большинства программистов эти изменения могут стать самым важным нововведением в Java SE 7. Ниже перечислены новые языковые средства, внедренные в JDK 7.

- Расширение типа `String` возможностями управлять ветвлением в операторе `switch`.
- Двоичные целочисленные литералы.
- Символы подчеркивания в числовых литералах.
- Расширение оператора `try`, называемое оператором `try с ресурсами`, обеспечивающим автоматическое управление ресурсами. (Например, потоки ввода-вывода могут теперь быть закрыты автоматически, когда они больше не нужны.)
- Выводимость типов (с помощью *ромбовидного* оператора `<>`) при создании обобщенного экземпляра.
- Улучшенная обработка исключений, благодаря которой два исключения или больше могут быть обработаны в одном блоке оператора `catch` (многократный перехват), а также улучшенный контроль соответствия типов повторно генерируемых исключений.
- Улучшение предупреждений компилятора, связанных с некоторыми типами методов с переменным количеством аргументов. И хотя это нельзя считать изменением в синтаксисе, оно дает больший контроль над предупреждениями.

Несмотря на то что средства, внедренные в рамках проекта *Project Coin*, считаются мелкими изменениями в языке Java, в целом они тем не менее дают много больше преимуществ, чем подразумевает само понятие “мелкие”. В частности, оператор `try с ресурсами` оказывает существенное влияние на порядок написания кода, опирающегося на потоки ввода-вывода. Кроме того, возможность использовать объекты типа `String` в операторе `switch` оказалась долгожданным усовершенствованием, позволившим во многом упростить код.

В версии Java SE 7 внесено несколько дополнений в библиотеку Java API. Важнейшими из них являются усовершенствования каркаса `NIO Framework` и дополнения каркаса `Fork/Join Framework`. Новая система ввода-вывода `NIO` (первоначально называвшаяся *New I/O*) была внедрена в версии Java 1.4. Но изменения, внесенные в версии Java SE 7, значительно расширили возможности этой системы, причем настолько, что она зачастую обозначается как *NIO.2*.

Каркас `Fork/Join Framework` обеспечивает важную поддержку параллельного программирования. Термином *параллельное программирование* обычно обозначаются технологии, повышающие эффективность применения компьютеров с несколь-

кими процессорами, включая многоядерные системы. Преимущества, которые дают многоядерные системы, позволяют в перспективе значительно повысить производительность программ. Каркас Fork/Join Framework позволяет решать задачи параллельного программирования в следующих областях:

- упрощение составления и использования заданий, которые могут выполняться параллельно;
- автоматическое использование нескольких процессоров.

Следовательно, с помощью каркаса Fork/Join Framework можно создавать приложения, в которых автоматически используются процессоры, доступные в исполняющей среде. Разумеется, не все алгоритмы оказываются параллельными, но те из них, которые таковыми все же являются, позволяют добиться существенных преимуществ в скорости выполнения.

Версия Java SE 8

В новейшем выпуске Java SE 8 с комплектом разработчика JDK 8 и внутренним номером версии 1.8 произошли значительные усовершенствования благодаря внедрению *лямбда-выражений* — нового языкового средства с далеко идущим последствием. Лямбда-выражения позволяют совершенно иначе подходить к программированию и написанию кода. Как поясняется более подробно в главе 15, лямбда-выражения вносят в Java возможности функционального программирования. В процессе программирования лямбда-выражения способны упростить и сократить объем исходного кода, требующегося для создания определенных конструкций, в том числе некоторых типов анонимных классов. Кроме того, лямбда-выражения вводят в язык новый оператор (\rightarrow) и элемент синтаксиса. И наконец, лямбда-выражения помогают сохранить живость и проворство языка, которые пользователи уже привыкли ожидать от Java.

Внедрение лямбда-выражений оказало обширное влияние и на библиотеки Java, которые были дополнены новыми средствами, выгодно использующими лямбда-выражения. К наиболее важным новшествам в библиотеке Java относится новый прикладной программный интерфейс API потоков ввода-вывода, входящий в пакет `java.util.stream`. В этом прикладном интерфейсе API, оптимизированном с учетом лямбда-выражений, поддерживаются конвейерные операции с данными. Еще одним важным новшеством является пакет `java.util.function`, в котором определен целый ряд *функциональных интерфейсов*, обеспечивающих дополнительную поддержку лямбда-выражений. В библиотеке Java API внедрены и другие, менее значительные средства, имеющие отношение к лямбда-выражениям.

Еще одно нововведение, навеянное лямбда-выражениями, касается *интерфейсов*. В версии JDK 8 появилась возможность определять реализацию по умолчанию метода, объявленного в интерфейсе. Если конкретная реализация такого метода отсутствует, то используется его реализация по умолчанию в интерфейсе. Таким образом, обеспечивается постепенное развитие интерфейсов, поскольку новый метод может быть введен в интерфейс, не нарушая существующий код. Это новшество позволя-

ет также рационализировать реализацию интерфейса при наличии подходящих средств, доступных по умолчанию. К числу других средств, появившихся в JDK 8, относятся новый прикладной программный интерфейс API для обработки даты и времени, типовые аннотации и возможность использовать параллельную обработку при сортировке массива. Кроме того, в состав JDK 8 включена поддержка JavaFX 8 — последней версии нового каркаса приложений Java с ГПИ. Предполагается, что каркас JavaFX станет неотъемлемой частью практически всех приложений Java и в конечном итоге заменит Swing для разработки большинства проектов на основе ГПИ. Введение в JavaFX представлено в части IV данной книги.

Подводя краткий итог, можно сказать, что Java SE 8 является основной версией, значительно расширяющей возможности Java и изменяющей порядок написания кода на этом языке. Последствия выпуска этой версии еще долго будут оказывать влияние на все области применения Java. И это действительно очень важное обновление Java.

Материал настоящего издания был обновлен таким образом, чтобы отражать переход к версии Java SE 8 со всеми новыми средствами, модификациями и дополнениями, обозначенными в соответствующих местах книги.

Культура нововведений

С самого начала язык Java оказался в центре культуры нововведений. Его первоначальная версия изменила подход к программированию для Интернета. Виртуальная машина Java (JVM) и байт-код совершенно изменили представление о безопасности и переносимости. Апплеты (а вслед за ними и сервлеты) вдохнули жизнь в веб. Процесс Java Community Process (JCP) изменил способ внедрения новых идей в язык. Область применения Java никогда не оставалась без изменений в течение длительного периода времени. И Java SE 8 остается на момент выхода этого издания из печати самой последней версией в непрекращающемся динамичном развитии Java.

Как и во всех остальных языках программирования, элементы Java существуют не сами по себе. Они скорее действуют совместно, образуя язык в целом. Но такая их взаимосвязанность может затруднять описание какого-то одного аспекта Java, не затрагивая ряда других. Зачастую для понимания одного языкового средства необходимо знать другое средство. Поэтому в этой главе представлен краткий обзор ряда основных языковых средств Java. Приведенный в ней материал послужит отправной точкой, чтобы создавать и понимать простые программы. Большинство рассмотренных в этой главе вопросов будут подробнее обсуждаться в остальных главах данной части.

Объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) составляет основу Java. По существу, все программы на Java являются в какой-то степени объектно-ориентированными. Язык Java связан с ООП настолько тесно, что прежде чем приступить к написанию на нем даже простейших программ, следует вначале ознакомиться с основными принципами ООП. Поэтому начнем с рассмотрения теоретических вопросов ООП.

Две методики

Все компьютерные программы состоят из двух элементов: кода и данных. Более того, программа концептуально может быть организована вокруг своего кода или своих данных. Иными словами, организация одних программ определяется тем, “что происходит”, а других – тем, “на что оказывается влияние”. Существуют две методики создания программ. Первая из них называется *моделью, ориентированной на процессы* и характеризует программу как последовательность линейных шагов (т.е. кода). Модель, ориентированную на процессы, можно рассматривать в качестве *кода, воздействующего на данные*. Такая модель довольно успешно применяется в процедурных языках вроде С. Но, как отмечалось в главе 1, подобный подход порождает ряд трудностей в связи с увеличением размеров и сложности программ.

С целью преодолеть увеличение сложности программ была начата разработка подхода, называемого *объектно-ориентированным программированием*. Объектно-ориентированное программирование позволяет организовать программу вокруг

ее данных (т.е. объектов) и набора вполне определенных интерфейсов с этими данными. Объектно-ориентированную программу можно охарактеризовать *как данные, управляющие доступом к коду*. Как будет показано далее, передавая функции управления данными, можно получить несколько организационных преимуществ.

Абстракция

Важным элементом ООП является *абстракция*. Человеку свойственно представлять сложные явления и объекты, прибегая к абстракции. Например, люди представляют себе автомобиль не в виде набора десятков тысяч отдельных деталей, а в виде совершенно определенного объекта, имеющего свое особое поведение. Эта абстракция позволяет не задумываться о сложности деталей, составляющих автомобиль, скажем, при поездке в магазин. Можно не обращать внимания на подробности работы двигателя, коробки передач и тормозной системы. Вместо этого объект можно использовать как единое целое.

Эффективным средством применения абстракции служат иерархические классификации. Это позволяет упрощать семантику сложных систем, разбивая их на более управляемые части. Внешне автомобиль выглядит единым объектом. Но стоит заглянуть внутрь, как становится ясно, что он состоит из нескольких подсистем: рулевого управления, тормозов, аудиосистемы, привязных ремней, обогревателя, навигатора и т.п. Каждая из этих подсистем, в свою очередь, собрана из более специализированных узлов. Например, аудиосистема состоит из радиоприемника, проигрывателя компакт-дисков и/или аудиокассет. Суть всего сказанного состоит в том, что структуру автомобиля (или любой другой сложной системы) можно описать с помощью иерархических абстракций.

Иерархические абстракции сложных систем можно применять и к компьютерным программам. Благодаря абстракции данные традиционной, ориентированной на процессы, программы можно преобразовать в составляющие ее объекты, а последовательность этапов процесса — в совокупность сообщений, передаваемых между этими объектами. Таким образом, каждый из этих объектов описывает свое особое поведение. Эти объекты можно считать конкретными сущностями, реагирующими на сообщения, предписывающие им *выполнить конкретное действие*. В этом, собственно, и состоит вся суть ООП.

Принципы ООП лежат как в основе языка Java, так и восприятия мира человеком. Важно понимать, каким образом эти принципы реализуются в программах. Как станет ясно в дальнейшем, ООП является еще одной, но более эффективной и естественной методикой создания программ, способных пережить неизбежные изменения, сопровождающие жизненный цикл любого крупного программного проекта, включая зарождение общего замысла, развитие и созревание. Например, при наличии тщательно определенных объектов и ясных, надежных интерфейсов с этими объектам можно безбоязненно и без особого труда извлекать или заменять части старой системы.

Три принципа ООП

Все языки объектно-ориентированного программирования предоставляют механизмы, облегчающие реализацию объектно-ориентированной модели. Этими

механизмами являются инкапсуляция, наследование и полиморфизм. Рассмотрим эти принципы ООП в отдельности.

Инкапсуляция

Механизм, связывающий код и данные, которыми он манипулирует, защищая оба эти компонента от внешнего вмешательства и злоупотреблений, называется *инкапсуляцией*. Инкапсуляцию можно считать защитной оболочкой, которая предохраняет код и данные от произвольного доступа со стороны другого кода, находящегося снаружи оболочки. Доступ к коду и данным, находящимся внутри оболочки, строго контролируется тщательно определенным интерфейсом. Чтобы провести аналогию с реальным миром, рассмотрим автоматическую коробку передач автомобиля. Она инкапсулирует немало сведений об автомобиле, в том числе величину ускорения, крутизну поверхности, по которой совершается движение, а также положение рычага переключения скоростей. Пользователь (в данном случае водитель) может оказывать влияние на эту сложную инкапсуляцию только одним способом: перемещая рычаг переключения скоростей. На коробку передач нельзя воздействовать, например, с помощью индикатора поворота или дворников. Таким образом, рычаг переключения скоростей является строго определенным, а по существу, единственным, интерфейсом с коробкой передач. Более того, происходящее внутри коробки передач не влияет на объекты, находящиеся вне ее. Например, переключение передач не включает фары! Функция автоматического переключения передач инкапсулирована, и поэтому десятки изготовителей автомобилей могут реализовать ее как угодно. Но с точки зрения водителя все эти коробки передач работают одинаково. Аналогичный принцип можно применять и в программировании. Сильная сторона инкапсулированного кода состоит в следующем: всем известно, как получить доступ к нему, а следовательно, его можно использовать независимо от подробностей реализации и не опасаясь неожиданных побочных эффектов.

Основу инкапсуляции в Java составляет класс. Подробнее классы будут рассмотрены в последующих главах, а до тех пор полезно дать хотя бы краткое их описание. *Класс* определяет структуру и поведение (данные и код), которые будут совместно использоваться набором объектов. Каждый объект данного класса содержит структуру и поведение, которые определены классом, как если бы объект был “отлит” в форме класса. Поэтому иногда объекты называют *экземплярами класса*. Таким образом, класс — это логическая конструкция, а объект — ее физическое воплощение.

При создании класса определяются код и данные, которые образуют этот класс. Совместно эти элементы называются *членами класса*. В частности, определенные в классе данные называются *переменными-членами*, или *переменными экземпляра*, а код, оперирующий данными, — *методами-членами*, или просто *методами*. (То, что программирующие на Java называют *методами*, программирующие на C/C++ называют *функциями*.) В программах, правильно написанных на Java, методы определяют, каким образом используются переменные-члены. Это означает, что поведение и интерфейс класса определяются методами, оперирующими данными его экземпляра.

Поскольку назначение класса состоит в инкапсуляции сложной структуры программы, существуют механизмы сокрытия сложной структуры реализации в самом классе. Каждый метод или переменная в классе могут быть помечены как закрытые или открытые. *Открытый* интерфейс класса представляет все, что должны или мо-

гут знать внешние пользователи класса. *Закрытые* методы и данные могут быть доступны только для кода, который является членом данного класса. Следовательно, любой другой код, не являющийся членом данного класса, не может получать доступ к закрытому методу или переменной. Закрытые члены класса доступны другим частям программы только через открытые методы класса, и благодаря этому исключается возможность выполнения неправомерных действий. Это, конечно, означает, что открытый интерфейс должен быть тщательно спроектирован и не должен раскрывать лишние подробности внутреннего механизма работы класса (рис 2.1).

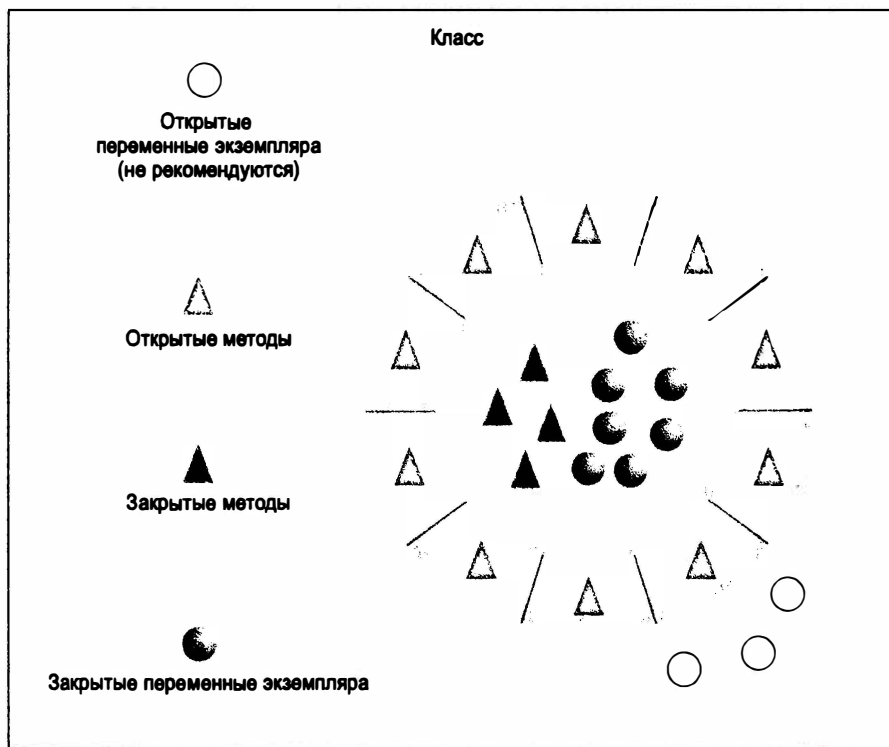


Рис. 2.1. Инкапсуляция: открытые методы можно использовать для защиты закрытых данных

Наследование

Процесс, в результате которого один объект получает свойства другого, называется *наследованием*. Это очень важный принцип ООП, поскольку наследование обеспечивает принцип иерархической классификации. Как отмечалось ранее, большинство знаний становятся доступными для усвоения благодаря иерархической (т.е. нисходящей) классификации. Например, золотистый ретривер — часть классификации *собак*, которая, в свою очередь, относится к классу *млекопитающих*, а тот — к еще большему классу *животных*. Без иерархий каждый объект должен был бы явно определять все свои характеристики. Но благодаря наследованию объект должен определять только те из них, которые делают его особым в классе. Объект может наследовать общие атрибуты от своего родительского объекта. Таким об-

разом, механизм наследования позволяет сделать один объект частным случаем более общего случая. Рассмотрим этот механизм подробнее.

Как правило, большинство людей воспринимают окружающий мир в виде иерархически связанных между собой объектов, подобных животным, млекопитающим и собакам. Если требуется привести абстрактное описание животных, можно сказать, что они обладают определенными свойствами: размеры, уровень интеллекта и костная система. Животным присущи также определенные особенности поведения: они едят, дышат и спят. Такое описание свойств и поведения составляет определение *класса* животных.

Если бы потребовалось описать более конкретный класс животных, например млекопитающих, следовало бы указать более конкретные свойства, в частности тип зубов и молочных желез. Такое определение называется *подклассом* животных, которые относятся к *суперклассу* (родительскому классу) млекопитающих. А поскольку млекопитающие — лишь более точно определенные животные, то они *наследуют* все свойства животных. Подкласс нижнего уровня *иерархии классов* наследует все свойства каждого из его родительских классов (рис. 2.2).

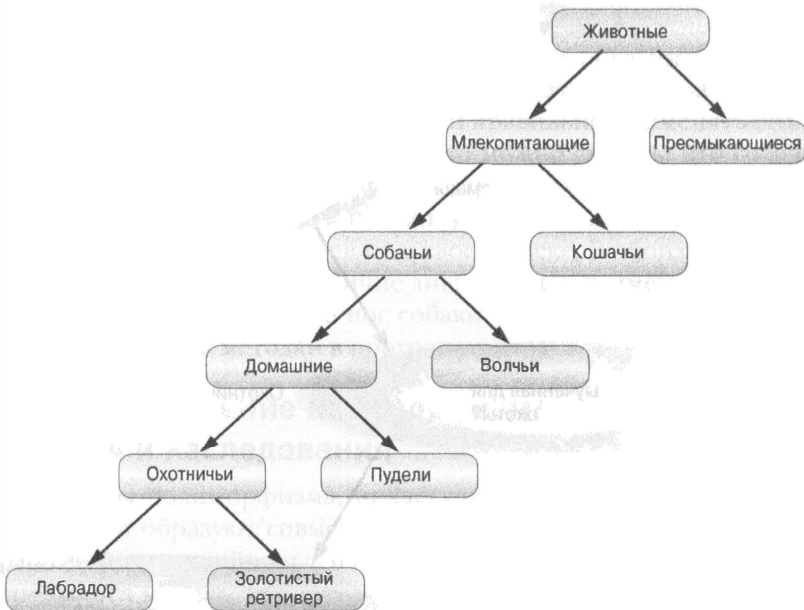
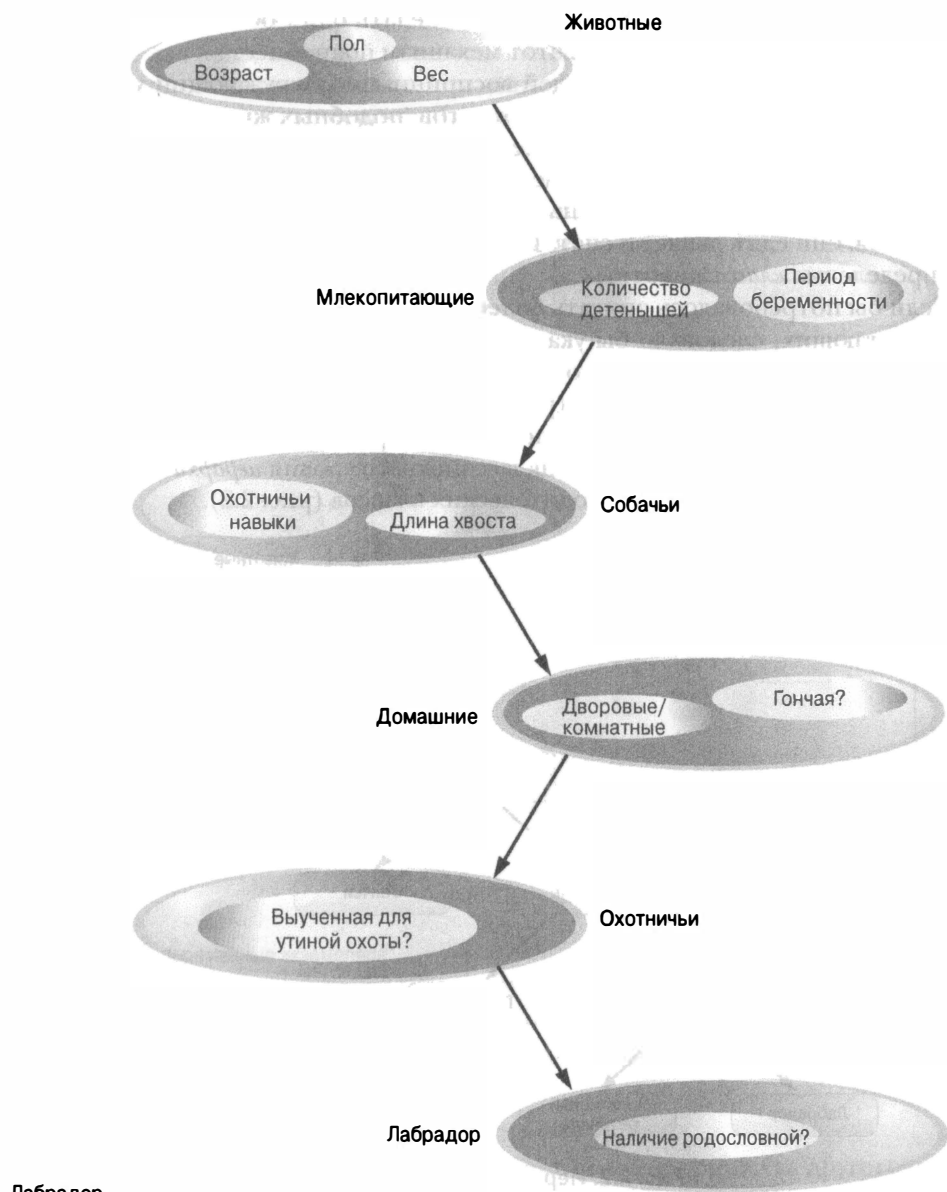


Рис. 2.2. Иерархия классов животных

Наследование связано также с инкапсуляцией. Если отдельный класс инкапсулирует определенные свойства, то любой его подкласс будет иметь те же самые свойства *плюс* любые дополнительные, определяющие его специализацию (рис. 2.3). Благодаря этому ключевому принципу сложность объектно-ориентированных программ нарастает в арифметической, а не геометрической прогрессии. Новый подкласс наследует атрибуты всех своих родительских классов и поэтому не содержит непредсказуемых взаимодействий с большей частью остального кода системы.



Лабрадор

Возраст	Период беременности	Гончая?
Пол	Охотничьи навыки	Выученная для утиной охоты?
Вес	Длина хвоста	Наличие родословной?
Количество детенышей	Дворовые/комнатные	

Рис. 2.3. Лабрадор полностью наследует инкапсулированные свойства всех родительских классов животных

Полиморфизм

Полиморфизм (от греч. “много форм”) — это принцип ООП, позволяющий использовать один и тот же интерфейс для общего класса действий. Каждое действие зависит от конкретной ситуации. Рассмотрим в качестве примера стек, действующий как список обратного магазинного типа. Допустим, в программе требуются стеки трех типов: для целочисленных значений, для числовых значений с плавающей точкой и для символов. Алгоритм реализации каждого из этих стеков остается неизменным, несмотря на отличия в данных, которые в них хранятся. В языке, не являющемся объектно-ориентированным, для обращения со стеком пришлось бы создавать три разных ряда служебных программ под отдельными именами. А в Java, благодаря принципу полиморфизма, для обращения со стеком можно определить общий ряд служебных программ под одними и теми же общими именами.

В более общем смысле принцип полиморфизма нередко выражается фразой “один интерфейс, несколько методов”. Это означает, что можно разработать общий интерфейс для группы связанных вместе действий. Такой подход позволяет уменьшить сложность программ, поскольку один и тот же интерфейс служит для указания *общего класса действий*. А выбор *конкретного действия* (т.е. метода) делается применительно к каждой ситуации и входит в обязанности компилятора. Это избавляет программиста от необходимости делать такой выбор вручную. Ему нужно лишь помнить об общем интерфейсе и правильно применять его.

Если продолжить аналогию с собаками, то можно сказать, что собачье обоняние — полиморфное свойство. Если собака почувствует запах кошки, она залает и погонится за ней. А если собака почувствует запах своего корма, то у нее начнется слюноотделение, и она поспешит к своей миске. В обоих случаях действует одно и то же чувство обоняния. Отличие лишь в том, что именно издает запах, т.е. в типе данных, воздействующих на нос собаки! Этот общий принцип можно реализовать, применив его к методам в программе на Java.

Совместное применение полиморфизма, инкапсуляции и наследования

Если принципы полиморфизма, инкапсуляции и наследования применяются правильно, то они образуют совместно среду программирования, поддерживающую разработку более устойчивых и масштабируемых программ, чем в том случае, когда применяется модель, ориентированная на процессы. Тщательно продуманная иерархия классов служит прочным основанием для многократного использования кода, на разработку и проверку которого были затрачены время и усилия. Инкапсуляция позволяет возвращаться к ранее созданным реализациям, не нарушая код, зависящий от открытого интерфейса применяемых в приложении классов. А полиморфизм позволяет создавать понятный, практичный, удобочитаемый и устойчивый код.

Из двух приведенных ранее примеров из реальной жизни пример с автомобилями более полно иллюстрирует возможности ООП. Если пример с собаками вполне подходит для рассмотрения ООП с точки зрения наследования, то пример с автомобилями имеет больше общего с программами. Садясь за руль различных типов

(подклассов) автомобилей, все водители пользуются наследованием. Независимо от того, является ли автомобиль школьным автобусом, легковым, спортивным автомобилем или семейным микроавтобусом, все водители смогут легко найти руль, тормоза, педаль акселератора и пользоваться ими. Немного повозившись с рычагом переключения передач, большинство людей могут даже оценить отличия ручной коробки передач от автоматической, поскольку они имеют ясное представление об общем родительском классе этих объектов – системе передач.

Пользуясь автомобилями, люди постоянно взаимодействуют с их инкапсулированными характеристиками. Педали тормоза и газа скрывают невероятную сложность соответствующих объектов за настолько простым интерфейсом, что для управления этими объектами достаточно нажать ступней педаль! Конкретная реализация двигателя, тип тормозов и размер шин не оказывают никакого влияния на порядок взаимодействия с определением класса педалей.

И наконец, полиморфизм ясно отражает способность изготовителей автомобилей предлагать большое разнообразие вариантов, по сути, одного и того же средства передвижения. Так, на автомобиле могут быть установлены система тормозов с защитой от блокировки или традиционные тормоза, рулевая система с гидроусилителем или с реечной передачей и 4-, 6- или 8-цилиндровые двигатели. Но в любом случае придется нажать на педаль тормоза, чтобы остановиться, вращать руль, чтобы повернуть, и нажать на педаль акселератора, чтобы автомобиль двигался быстрее. Один и тот же интерфейс может быть использован для управления самыми разными реализациями.

Как видите, благодаря совместному применению принципов инкапсуляции, наследования и полиморфизма отдельные детали удастся превратить в объект, называемый автомобилем. Это же относится и к компьютерным программам. Принципы ООП позволяют составить связную, надежную, сопровождаемую программу из многих отдельных частей.

Как отмечалось в начале этого раздела, каждая программа на Java является объектно-ориентированной. Точнее говоря, в каждой программе на Java применяются принципы инкапсуляции, наследования и полиморфизма. На первый взгляд может показаться, что не все эти принципы проявляются в коротких примерах программ, приведенных в остальной части этой главы и ряде последующих глав, тем не менее они в них присутствуют. Как станет ясно в дальнейшем, многие языковые средства Java являются составной частью встроенных библиотек классов, в которых широко применяются принципы инкапсуляции, наследования и полиморфизма.

Первый пример простой программы

А теперь, когда разъяснены самые основы объектно-ориентированного характера Java, рассмотрим несколько практических примеров программ, написанных на этом языке. Начнем с компиляции и запуска короткого примера программы, обсуждаемого в этом разделе. Оказывается, что эта задача не так проста, как может показаться на первый взгляд.

```
/*
    Это простая программа на Java.
    Присвоить исходному файлу имя "Example.java"
*/
class Example {
    // Эта программа начинается с вызова метода main()
    public static void main(String args[]) {
        System.out.println("Простая программа на Java.");
    }
}
```

На заметку! Здесь и далее используется стандартный комплект разработчика Java SE 8 Developer's Kit (JDK 8), предоставляемый компанией Oracle. Если же для написания программ на Java применяется интегрированная среда разработки (ИСР), то для компиляции и выполнения программ может потребоваться другая процедура. В таком случае обращайтесь за справкой к документации на применяемую ИСР.

Ввод кода программы

Для большинства языков программирования имя файла, который содержит исходный код программы, не имеет значения. Но в Java дело обстоит иначе. Прежде всего следует твердо усвоить, что исходному файлу очень важно присвоить имя. В данном примере исходному файлу должно быть присвоено **Example.java**. И вот почему.

В Java исходный файл официально называется *единицей компиляции*. Он, среди прочего, представляет собой текстовый файл, содержащий определения одного или нескольких классов. (Будем пока что пользоваться исходными файлами, содержащими только один класс.) Компилятор Java требует, чтобы исходный файл имел расширение **.java**.

Как следует из исходного кода рассматриваемого здесь примера программы, определенный в ней класс также называется **Example**. И это не случайно. В Java весь код должен размещаться в классе. По принятому соглашению имя главного класса должно совпадать с именем файла, содержащего исходный код программы. Кроме того, написание имени исходного файла должно точно соответствовать имени главного класса, включая строчные и прописные буквы. Дело в том, что в коде Java учитывается регистр символов. На первый взгляд, соглашение о строгом соответствии имен файлов и классов может показаться произвольным. Но на самом деле оно упрощает сопровождение и организацию программ.

Компиляция программы

Чтобы скомпилировать программу **Example**, запустите компилятор (**javac**), указав имя исходного файла в командной строке следующим образом:

```
C:\>javac Example.java
```

Компилятор **javac** создаст файл **Example.class**, содержащий версию байт-кода. Как пояснялось ранее, байт-код Java является промежуточным представлением программы, содержащим инструкции, которые будет выполнять виртуальная машина JVM. Следовательно, компилятор **javac** выдает результат, который не является непосредственно исполняемым кодом.

Чтобы выполнить программу, следует воспользоваться загрузчиком приложений Java, который называется `java`. Ему нужно передать имя класса `Example` в качестве аргумента командной строки, как показано ниже.

```
C:\>java Example
```

Выполнение данной программы приведет к выводу на экран следующего результата:

Простая программа на Java.

В процессе компиляции исходного кода каждый отдельный класс помещается в собственный выходной файл, называемый по имени класса и получающий расширение `.class`. Поэтому исходным файлам программ на Java целесообразно присваивать имена, совпадающие с именами классов, которые содержатся в файлах с расширением `.class`. При запуске загрузчика приложений `java` описанным выше способом в командной строке на самом деле указывается имя класса, который нужно выполнить. Загрузчик приложений автоматически будет искать файл с указанным именем и расширением `.class`. И если он найдет такой файл, то выполнит код, содержащийся в указанном классе.

Подробный анализ первого примера программы

Хотя сама программа `Example.java` небольшая, с ней связано несколько важных особенностей, характерных для всех программ на Java. Проанализируем подробно каждую часть этой программы. Начинается эта программа со следующих строк:

```
/*  
    Это простая программа на Java.  
    Присвоить исходному файлу имя "Example.java"  
*/
```

Эти строки кода содержат *комментарий*. Подобно большинству других языков программирования, Java позволяет вставлять примечания к коду программы в ее исходный файл. Компилятор игнорирует содержимое комментариев. Эти комментарии описывают или поясняют действия программы для тех, кто просматривает ее исходный код. В данном случае комментарий описывает программу и напоминает, что исходному файлу должно быть присвоено имя `Example.java`. Разумеется, в реальных прикладных программах комментарии служат главным образом для пояснения работы отдельных частей программы или действий, выполняемых отдельными языковыми средствами.

В Java поддерживаются три вида комментариев. Комментарий, приведенный в начале программы, называется *многострочным*. Этот вид комментариев должен начинаться с символов `/*` и оканчиваться символами `*/`. Весь текст, расположенный между этими двумя парами символов, игнорируется компилятором. Как следует из его названия, многострочный комментарий может содержать несколько строк.

Перейдем к следующей строке кода анализируемой здесь программы. Ниже показано, как она выглядит.

```
class Example {
```

В этой строке кода ключевое слово `class` служит для объявления вновь определяемого класса, а `Example` — в качестве *идентификатора*, обозначающего имя клас-

са. Все определение класса, в том числе его членов, должно располагаться между открывающей ({) и закрывающей (}) фигурными скобками. Мы не станем пока что останавливаться на особенностях реализации класса. Отметим только, что в среде Java все действия программы выполняются в пределах класса. В этом и состоит одна из причин, по которым все программы на Java являются (по крайней мере, частично) объектно-ориентированными.

Следующая строка кода данной программы содержит однострочный комментарий:

```
// Эта программа начинается с вызова метода main()
```

Это второй вид комментариев, поддерживаемых в Java. Он называется *однострочным комментарием* и начинается с символов //, а завершается символом конца строки. Как правило, программисты пользуются многострочными комментариями для вставки длинных примечаний, а однострочными — для коротких, построчных описаний. Третий вид комментариев, называемый *документирующим*, будет рассмотрен далее в разделе “Комментарии”.

Перейдем к следующей строке кода анализируемой здесь программы. Ниже показано, как она выглядит.

```
public static void main(String args[ ]) {
```

Эта строка кода начинается с объявления метода `main()`. Как следует из предшествующего ей комментария, выполнение программы начинается именно с этой строки кода. Выполнение всех прикладных программ на Java начинается с вызова метода `main()`. Мы не станем пока что разъяснять подробно назначение каждого элемента этой строки, потому что для этого требуется ясное представление о подходе к инкапсуляции, принятом в Java. Но поскольку эта строка кода присутствует в большинстве примеров из данной части книги, то проанализируем ее вкратце.

Ключевое слово **public** является *модификатором доступа*, который дает программисту возможность управлять видимостью членов класса. Когда члену класса предшествует ключевое слово **public**, этот член доступен из кода за пределами класса, где он определен. (Совершенно противоположное обозначает ключевое слово **private** — оно не разрешает доступ к члену класса из кода за пределами данного класса.) В данном случае метод `main()` должен быть определен как **public**, поскольку при запуске программы он должен вызываться из кода за пределами его класса. Ключевое слово **static** позволяет вызывать метод `main()` без получения экземпляра класса. Это необходимо потому, что метод `main()` вызывается виртуальной машиной JVM перед созданием любых объектов. А ключевое слово **void** просто сообщает компилятору, что метод `main()` не возвращает никаких значений. Как будет показано далее, методы могут также возвращать конкретные значения. Если это краткое пояснение покажется вам не совсем понятным, не отчаивайтесь, поскольку упомянутые здесь понятия и языковые средства Java будут подробно рассматриваться в последующих главах.

Как указывалось выше, метод `main()` вызывается при запуске прикладных программ на Java. Следует, однако, иметь в виду, что в Java учитывается регистр символов. Следовательно, имя `Main` не равнозначно имени `main`. Следует также иметь в виду, что компилятор Java скомпилирует классы, в которых отсутствует метод

`main()`, но загрузчик приложений (**java**) не сможет выполнить код таких классов. Так, если вместо имени `main` ввести имя `Main`, компилятор все равно скомпилирует программу, но загрузчик приложений **java** выдаст сообщение об ошибке, поскольку ему не удалось обнаружить метод `main()`.

Для передачи любой информации, требующейся методу, служат переменные, указываемые в скобках вслед за именем метода. Эти переменные называются *параметрами*. Если параметры не требуются методу, то указываются пустые скобки. У метода `main()` имеется единственный, хотя и довольно сложный параметр. Так, в выражении `String args[]` объявляется параметр `args`, обозначающий массив экземпляров класса `String`. (*Массивы* — это коллекции похожих объектов.) В объектах типа `String` хранятся символьные строки. В данном случае параметр `args` принимает любые аргументы командной строки, присутствующие во время выполнения программы. В данной программе эта информация, вводимая из командной строки, не используется, но в других, рассматриваемых далее примерах программ она будет применяться.

Последним элементом в рассматриваемой здесь строке кода оказывается символ открывающей фигурной скобки (`{`). Он обозначает начало тела метода `main()`. Весь код, составляющий тело метода, должен располагаться между открывающей и закрывающей фигурными скобками в определении этого метода.

Еще один важный момент: метод `main()` служит всего лишь началом программы. Сложная программа может включать в себя десятки классов, но только один из них должен содержать метод `main()`, чтобы программу можно было запустить на выполнение. Но в некоторых случаях метод `main()` вообще не требуется, например, при создании апплетов — прикладных программ на Java, внедряемых в веб-браузеры. Метод `main()` в апплетах не требуется потому, что для их запуска на выполнение применяются другие средства.

Перейдем к следующей строке кода анализируемой здесь программы. Ниже показано, как она выглядит. Следует также иметь в виду, что эта строка кода находится в теле метода `main()`.

```
System.out.println("Простая программа на Java.");
```

В этой строке кода на экран выводится текстовая строка "Простая программа на Java." с последующим переходом на новую строку. На самом деле вывод текста на экран выполняется встроенным методом `println()`. В данном случае метод `println()` отображает переданную ему текстовую строку. Как будет показано далее, с помощью этого метода можно выводить и другие типы данных. Анализируемая здесь строка кода начинается с обозначения стандартного потока вывода **System.out**. Это слишком сложная языковая конструкция, чтобы ее можно было просто объяснить на данной стадии изучения Java, но вкратце **System** обозначает предопределенный класс, предоставляющий доступ к системе, а **out** — поток вывода, связанный с консолью.

Нетрудно догадаться, что в реальных программах на Java консольный вывод применяется редко. Многие современные вычислительные среды по своему характеру являются оконными и графическими, поэтому консольный ввод-вывод зачастую применяется в простых служебных и демонстрационных программах. В дальнейшем будут рассмотрены другие способы ввода-вывода данных в Java, а до тех пор будут применяться методы консольного ввода-вывода.

Обратите внимание на то, что оператор, в котором вызывается метод `println()`, завершается точкой с запятой. В языке Java все операторы обычно должны оканчиваться этим символом. Причина отсутствия точки с запятой в конце остальных строк кода программы состоит в том, что формально они не являются операторами. Первый символ `}` завершает метод `main()`, а последний символ `}` — определение класса `Example`.

Второй пример короткой программы

Вероятно, ни одно другое понятие не является для языка программирования столь важным, как понятие переменных. Как вы, вероятно, знаете, *переменная* — это именованная ячейка памяти, которой может быть присвоено значение в программе. Во время выполнения программы значение переменной может изменяться. В следующем примере программы демонстрируются способы объявления переменной и присвоения ей значения. Этот пример иллюстрирует также некоторые новые аспекты консольного вывода. Как следует из комментариев в начале программы, ее исходному файлу следует присвоить имя `Example2.java`.

```
/*
  Это еще один короткий пример программы.
  Присвоить исходному файлу имя "Example2.java"
*/
class Example2 {
    public static void main(String args[]) {
        int num; // в этой строке кода объявляется переменная с именем num

        num = 100; // в этой строке кода переменной num
                  // присваивается значение 100

        System.out.println("Это переменная num: " + num);

        num = num * 2;

        System.out.print("Значение переменной num * 2 равно ");
        System.out.println(num);
    }
}
```

Выполнение данной программы приведет к выводу на экран следующего результата:

```
Это переменная num: 100
Значение переменной num * 2 равно 200
```

Рассмотрим подробнее получение такого результата. Ниже приведена строка кода из рассматриваемой здесь программы, которая еще не встречалась в предыдущем примере.

```
int num; // в этой строке кода объявляется переменная с именем num
```

В этой строке кода объявляется целочисленная переменная `num`. В Java, как и в большинстве других языков программирования, требуется, чтобы переменные были объявлены до их применения. Ниже приведена общая форма объявления переменных.

тип имя_переменной;

В этом объявлении *тип* обозначает конкретный тип объявляемой переменной, а *имя_переменной* — заданное имя переменной. Если требуется объявить несколько переменных заданного типа, это можно сделать в виде разделенного запятыми списка имен переменных. В Java определен целый ряд типов данных, в том числе целочисленный, символьный и числовой с плавающей точкой. Ключевое слово `int` обозначает целочисленный тип. В приведенной ниже строке кода из рассматриваемого здесь примера программы переменной `num` присваивается значение `100`. В Java операция присваивания обозначается одиночным знаком равенства.

```
num = 100; // в этой строке кода переменной num присваивается значение 100
```

В следующей строке кода выводится значение переменной `num`, которому предшествует текстовая строка "Это переменная num:":

```
System.out.println("Это переменная num: " + num);
```

В этом операторе знак `+` присоединяет значения переменной `num` в конце предшествующей ему текстовой строки, а затем выводится результирующая строка. (На самом деле значение переменной `num` сначала преобразуется из целочисленного в строковый эквивалент, а затем объединяется с предшествующей строкой. Подробнее этот процесс описывается далее в книге.) Такой подход можно обобщить. С помощью операции `+` в одном вызове метода `println()` можно объединить нужное количество символьных строк.

В следующей строке кода из рассматриваемого здесь примера программы переменной `num` присваивается хранящееся в ней значение, умноженное на `2`. Как и в большинстве других языков программирования, в Java знак `*` обозначает арифметическую операцию умножения. После выполнения этой строки кода переменная `num` будет содержать значение `200`.

Ниже приведены две следующие строки кода из рассматриваемого здесь примера программы.

```
System.out.print("Значение переменной num * 2 равно ");  
System.out.println(num);
```

В них выполняется ряд новых действий. В частности, метод `print()` вызывается для вывода текстовой строки "Значение переменной `num` * 2 равно". После этой строки *не* следует символ новой строки. Таким образом, следующий результат будет выводиться в той же самой строке. Метод `print()` действует аналогично методу `println()`, за исключением того, что после каждого вызова он не выводит символ новой строки. А теперь рассмотрим вызов метода `println()`. Обратите внимание на то, что имя переменной `num` указывается буквально. Методы `print()` и `println()` могут служить для вывода значений любых встроенных в Java типов данных.

Два управляющих оператора

Хотя управляющие операторы подробно описываются в главе 5, в этом разделе будут кратко рассмотрены два управляющих оператора, чтобы было понятно их назначение в примерах программ, приведенных в главах 3 и 4. Кроме того, они послужат хорошей иллюстрацией важного аспекта Java — блоков кода.

Условный оператор `if`

Оператор `if` действует подобно условному оператору в любом другом языке программирования. Более того, его синтаксис такой же, как и условных операторов `if` в языках C, C++ и C#. Простейшая форма этого оператора выглядит следующим образом:

```
if(условие) оператор;
```

где *условие* обозначает логическое выражение. Если *условие* истинно, то оператор выполняется. А если *условие* ложно, то оператор пропускается. Рассмотрим следующую строку кода:

```
if(num < 100) System.out.println("num меньше 100");
```

Если в данной строке кода переменная `num` содержит значение меньше 100, то условное выражение истинно и вызывается метод `println()`. А если переменная `num` содержит значение, большее или равное 100, то вызов метода `println()` пропускается.

Как будет показано в главе 4, в языке Java определен полный набор операций сравнения, которые можно использовать в условном выражении. Некоторые из них перечислены в табл. 2.1.

Таблица 2.1. Некоторые операции сравнения

Операция	Значение
<	Меньше
>	Больше
==	Равно

Следует иметь в виду, что проверка на равенство обозначается двойным знаком равенства (`==`). Ниже приведен пример программы, демонстрирующий применение условного оператора `if`.

```
/*
    Продемонстрировать применение условного оператора if.

    Присвоить исходному файлу имя "IfSample.java"
*/
class IfSample {
    public static void main(String args[]) {
        int x, y;

        x = 10;
        y = 20;

        if(x < y) System.out.println("x меньше y");

        x = x * 2;
        if(x == y) System.out.println("x теперь равно y");

        x = x * 2;
        if(x > y) System.out.println("x теперь больше y");

        // этот оператор не будет ничего выводить
        if(x == y) System.out.println("вы не увидите этого");
    }
}
```

```
    }
}
```

Эта программа выводит следующий результат:

```
x меньше y
x теперь равно y
x теперь больше y
```

Обратите внимание на еще одну особенность данного примера программы. В строке кода

```
int x, y;
```

объявляются две переменные — *x* и *y*. Для этого используется список, разделяемый запятыми.

Оператор цикла `for`

Из имеющегося опыта программирования вам, должно быть, известно, что операторы цикла являются важной составной частью практически любого языка программирования. И язык Java не является в этом отношении исключением. Как будет показано в главе 5, в Java поддерживаются разнообразные операторы цикла. И, вероятно, наиболее универсальным среди них является оператор цикла `for`. Ниже приведена простейшая форма этого оператора.

`for` (инициализация; условие; итерация) оператор;

В этой чаще всего встречающейся форме *инициализация* обозначает начальное значение переменной управления циклом, а *условие* — логическое выражение для проверки значения переменной управления циклом. Если результат проверки *условия* истинен, то выполнение цикла `for` продолжается. А если результат этой проверки ложен, то выполнение цикла прекращается. Выражение *итерация* определяет порядок изменения переменной управления циклом на каждом его шаге. В приведенном ниже кратком примере программы демонстрируется применение оператора цикла `for`.

```
/*
    Продемонстрировать применение цикла for.

    Присвоить исходному файлу имя "ForTest.java"
*/
class ForTest {
    public static void main(String args[]) {
        int x;

        for(x = 0; x<10; x = x+1)
            System.out.println("Значение x: " + x);
    }
}
```

Эта программа выводит следующий результат:

```
Значение x: 0
Значение x: 1
Значение x: 2
Значение x: 3
Значение x: 4
```

```
Значение x: 5
Значение x: 6
Значение x: 7
Значение x: 8
Значение x: 9
```

В данном примере `x` служит переменной управления циклом. В инициализирующей части цикла `for` ей присваивается начальное нулевое значение. В начале каждого шага цикла, включая и первый, выполняется проверка условия `x < 10`. Если результат этой проверки истинен, в программе вызывается метод `println()`, а затем выполняется итерационная часть цикла `for`. Процесс продолжается до тех пор, пока результат проверки условия не окажется ложным.

Следует заметить, что в программах, профессионально написанных на Java, итерационная часть цикла почти никогда не встречается в том виде, в каком она представлена в данном примере. Иными словами, операция вроде следующей в таких программах встречаются крайне редко:

```
x = x + 1;
```

Дело в том, что в Java предоставляется специальная, более эффективная операция инкремента `++` (т.е. два знака “плюс” подряд). Операция инкремента увеличивает значение операнда на единицу. Используя эту операцию, предшествующее выражение можно переписать так:

```
x++;
```

Таким образом, оператор цикла `for` из предыдущего примера программы можно переписать следующим образом:

```
for(x = 0; x<10; x++)
```

Можете проверить выполнение этого цикла и убедиться, что он действует точно так же, как и в предыдущем примере. В Java предоставляется также операция декремента `--`, которая уменьшает значение операнда на единицу.

Использование блоков кода

В языке Java два и более оператора допускается группировать в *блоки кода*, называемые также *кодowymi блоками*. С этой целью операторы заключают в фигурные скобки. Сразу после своего создания блок кода становится логической единицей, которую можно использовать в тех же местах, где и отдельные операторы. Например, блок кода может служить в качестве адресата для операторов `if` и `for`. Рассмотрим следующий условный оператор `if`:

```
if(x < y) { // начало блока
    x = y;
    y = 0;
}          // конец блока
```

Если значение переменной `x` в данном примере меньше `y`, то выполняются оба оператора, находящиеся в блоке кода. Оба эти оператора образуют логическую единицу, и поэтому выполнение одного из них невозможно без выполнения другого в блоке кода. Таким образом, всякий раз, когда требуется логически связать два или более оператора, создается блок кода.

Рассмотрим еще один пример. В следующей программе блок кода служит в качестве адресата для оператора цикла `for`:

```
/*
Продемонстрировать применение блока кода.

Присвоить исходному файлу имя "BlockTest.java"
*/
class BlockTest {
    public static void main(String args[]) {
        int x, y;

        y = 20;

        // адресатом этого оператора цикла служит блок кода
        for(x = 0; x<10; x++) {
            System.out.println("Значение x: " + x);
            System.out.println("Значение y: " + y);
            y = y - 2;
        }
    }
}
```

Эта программа выводит следующий результат:

```
Значение x: 0
Значение y: 20
Значение x: 1
Значение y: 18
Значение x: 2
Значение y: 16
Значение x: 3
Значение y: 14
Значение x: 4
Значение y: 12
Значение x: 5
Значение y: 10
Значение x: 6
Значение y: 8
Значение x: 7
Значение y: 6
Значение x: 8
Значение y: 4
Значение x: 9
Значение y: 2
```

В данном случае адресатом оператора цикла `for` служит блок кода, а не единственный оператор. Таким образом, на каждом шаге цикла будут выполняться три оператора из блока кода. Об этом свидетельствует и результат выполнения программы.

Как будет показано в последующих главах, блоки кода обладают дополнительными свойствами и областями применения. Но их основное назначение — создание логически неразрывных единиц кода.

Вопросы лексики

А теперь, когда было рассмотрено несколько кратких примеров программ на Java, настало время для более формального описания самых основных элемен-

тов языка. Исходный текст программ на Java состоит из совокупности пробелов, идентификаторов, литералов, комментариев, операторов, разделителей и ключевых слов. Операторы рассматриваются в следующей главе, а остальные элементы кратко описываются в последующих разделах этой главы.

Пробелы

Java — язык свободной формы. Это означает, что при написании программы не нужно следовать каким-то специальным правилам в отношении отступов. Например, программу Example можно было бы написать в одной строке или любым другим способом. Единственное обязательное требование — наличие, по меньшей мере, одного пробела между всеми лексемами, которые еще не разграничены оператором или разделителем. В языке Java пробелами считаются символы пробела, табуляции или новой строки.

Идентификаторы

Для именования классов, методов и переменных служат идентификаторы. Идентификатором может быть любая последовательность строчных и прописных букв, цифр или символов подчеркивания и денежной единицы. (Знак денежной единицы не предназначается для общего использования.) Идентификаторы не должны начинаться с цифры, чтобы компилятор не путал их с числовыми константами. Напомним еще раз, что в Java учитывается регистр символов, и поэтому VALUE и Value считаются разными идентификаторами. Ниже приведено несколько примеров допустимых идентификаторов.

AvgTemp	count	a4	\$test	this_is_ok
---------	-------	----	--------	------------

А следующие идентификаторы недопустимы:

2count	high-temp	Not/ok
--------	-----------	--------

На заметку! Начиная с версии JDK 8, сам знак подчеркивания не рекомендуется употреблять в качестве идентификатора.

Литералы

В Java постоянное значение задается его *литеральным* представлением. В качестве примера ниже показано несколько литералов.

100	98.6	'X'	"This is a test"
-----	------	-----	------------------

Первый литерал в данном примере обозначает целочисленное значение, следующий — числовое значение с плавающей точкой, третий — символьную константу, а последний — строковое значение. Литерал можно использовать везде, где допустимо применение значений данного типа.

Комментарии

Как отмечалось ранее, в Java определены три вида комментариев. Два из них (одно- и многострочные комментарии) уже встречались в рассмотренных ранее примерах программ. А третий вид комментариев называется *документирующим*. Этот вид комментариев служит для создания HTML-файла документации на программу. Документирующий комментарий начинается с символов `/**` и оканчивается символами `*/`. Подробнее документирующие комментарии описываются в приложении к книге.

Разделители

В Java допускается применение нескольких символов в качестве разделителей. Чаще всего в качестве разделителя употребляется точка с запятой. Но, как следует из приведенных ранее примеров программ, точка с запятой употребляется также для завершения строк операторов. Символы, допустимые в качестве разделителей, перечислены в табл. 2.2.

Таблица 2.2. Символы, допустимые в качестве разделителей

Символ	Название	Назначение
()	Круглые скобки	Употребляются для передачи списков параметров в определениях и вызовах методов. Применяются также для обозначения операции приведения типов и предшествования операторов в выражениях, употребляемых в управляющих операторах
{ }	Фигурные скобки	Употребляются для указания значений автоматически инициализируемых массивов, а также для определения блоков кода, классов, методов и локальных областей действия
[]	Квадратные скобки	Употребляются для объявления типов массивов, а также при обращении к элементам массивов
;	Точка с запятой	Завершает операторы
,	Запятая	Разделяет последовательный ряд идентификаторов в объявлениях переменных. Применяются также для создания цепочек операторов в операторе цикла <code>for</code>
.	Точка	Употребляется для отделения имен пакетов от подпакетов и классов, а также для отделения переменной или метода от ссылочной переменной

Ключевые слова Java

В настоящее время в языке Java определено 50 ключевых слов (табл. 2.3), которые вместе с синтаксисом операторов и разделителей образуют основу языка Java. Их нельзя использовать ни в качестве идентификаторов, ни для обозначения имен переменных, классов или методов.

Таблица 2.3. Ключевые слова Java

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Кроме ключевых слов, в Java зарезервированы также слова `true`, `false` и `null`. Они представляют значения, определенные в спецификации языка Java. Их нельзя использовать для обозначения имен переменных, классов и т.п.

Библиотеки классов Java

В приведенных в этой главе примерах программ использовались два встроенных метода Java: `println()` и `print()`. Как отмечалось ранее, эти методы доступны по ссылке `System.out` на класс `System`, который является стандартным в Java и автоматически включается в прикладные программы. В более широком смысле среда Java опирается на несколько встроенных библиотек классов, содержащих многие встроенные методы, обеспечивающие поддержку таких операций, как ввод-вывод, обработка символьных строк, работа в сети и отображение графики. Стандартные классы обеспечивают также поддержку ГПИ. Таким образом, среда Java в целом состоит из самого языка Java и его стандартных классов. Как станет ясно в дальнейшем, библиотеки классов предоставляют большую часть функциональных возможностей среды Java. В действительности стать программирующим на Java означает научиться пользоваться стандартными классами Java. В последующих главах этой части описание различных элементов стандартных библиотечных классов и методов приводится по мере надобности, а в части II библиотеки классов описаны более подробно.

Типы данных, переменные и массивы

В этой главе рассматриваются три самых основных элемента Java: типы данных, переменные и массивы. Как и во всех современных языках программирования, в Java поддерживается несколько типов данных. Их можно применять для объявления переменных и создания массивов. Как станет ясно в дальнейшем, в Java применяется простой, эффективный и связный подход к этим языковым средствам.

Java — строго типизированный язык

Прежде всего следует заметить, что Java — строго типизированный язык. Именно этим объясняется безопасность и надежность программ на Java. Выясним, что же это означает. Во-первых, каждая переменная и каждое выражение имеет конкретный тип, и каждый тип строго определен. Во-вторых, все операции присваивания, как явные, так и через параметры, передаваемые при вызове методов, проверяются на соответствие типов. В Java отсутствуют средства автоматического приведения или преобразования конфликтующих типов, как это имеет место в некоторых языках программирования. Компилятор Java проверяет все выражения и параметры на соответствие типов. Любые несоответствия типов считаются ошибками, которые должны быть исправлены до завершения компиляции класса.

Примитивные типы

В языке Java определены восемь *примитивных* типов данных: `byte`, `short`, `int`, `long`, `char`, `float`, `double` и `boolean`. Примитивные типы называют также *простыми*, и в данной книге употребляются оба эти термина. Примитивные типы можно разделить на следующие четыре группы.

- **Целые числа.** Эта группа включает в себя типы данных `byte`, `short`, `int` и `long`, представляющие целые числа со знаком.
- **Числа с плавающей точкой.** Эта группа включает в себя типы данных `float` и `double`, представляющие числа с точностью до определенного знака после десятичной точки.

- **Символы.** Эта группа включает в себя тип данных `char`, представляющий символы, например буквы и цифры, из определенного набора.
- **Логические значения.** Эта группа включает в себя тип данных `boolean`, специально предназначенный для представления логических истинных и ложных значений.

Эти типы данных можно использовать непосредственно или для создания собственных типов классов. Таким образом, они служат основанием для всех других типов данных, которые могут быть созданы.

Примитивные типы представляют одиночные значения, а не сложные объекты. Язык Java является полностью объектно-ориентированным, кроме примитивных типов данных. Они аналогичны простым типам данных, которые можно встретить в большинстве других не объектно-ориентированных языков программирования. Эта особенность Java объясняется стремлением обеспечить максимальную эффективность. Превращение примитивных типов в объекты привело бы к слишком заметному снижению производительности.

Примитивные типы определены таким образом, чтобы обладать явно выражаемым диапазоном допустимых значений и математически строгим поведением. Языки типа C и C++ допускают варьирование длины целочисленных значений в зависимости от требований исполняющей среды. Но в Java дело обстоит иначе. В связи с требованием переносимости программ на Java все типы данных обладают строго определенным диапазоном допустимых значений. Например, значения типа `int` всегда оказываются 32-разрядными, независимо от конкретной платформы. Это позволяет создавать программы, которые гарантированно будут выполняться в любой машинной архитектуре *без специального переноса*. В некоторых средах строгое указание длины целых чисел может приводить к незначительному снижению производительности, но это требование абсолютно необходимо для переносимости программ. Рассмотрим каждый из примитивных типов данных в отдельности.

Целые числа

Для целых чисел в языке Java определены четыре типа: `byte`, `short`, `int` и `long`. Все эти типы данных представляют целочисленные значения со знаком: как положительные, так и отрицательные. В Java не поддерживаются только положительные целочисленные значения без знака. Во многих других языках программирования поддерживаются целочисленные значения как со знаком, так и без знака, но разработчики Java посчитали целочисленные значения без знака ненужными. В частности, они решили, что понятие числовых значений *без знака* служило в основном для того, чтобы обозначить состояние *старшего бита*, определяющего знак целочисленного значения. Как будет показано в главе 4, в Java управление состоянием старшего бита осуществляется иначе: с помощью специальной операции “сдвига вправо без знака”. Благодаря этому отпадает потребность в целочисленном типе данных без знака.

Длина целочисленного типа означает не занимаемый объем памяти, а скорее поведение, определяемое им для переменных и выражений данного типа. В исполняющей среде Java может быть использована любая длина, при условии, что типы данных ведут себя так, как они объявлены. Как показано в табл. 3.1, длина и диапазон допустимых значений целочисленных типов данных изменяются в широких пределах.

Таблица 3.1. Длина и диапазон допустимых значений целочисленных типов данных

Наименование	Длина в битах	Диапазон допустимых значений
<code>long</code>	64	от -9223372036854775808 до 9223372036854775807
<code>int</code>	32	от -2147483648 до 2147483647
<code>short</code>	16	от -32768 до 32767
<code>byte</code>	8	от -128 до 127

А теперь рассмотрим каждый из целочисленных типов в отдельности.

Тип `byte`

Наименьшим по длине является целочисленный тип `byte`. Это 8-разрядный тип данных со знаком и диапазоном допустимых значений от -128 до 127. Переменные типа `byte` особенно удобны для работы с потоками ввода-вывода данных в сети или файлах. Они удобны также при манипулировании необработанными двоичными данными, которые могут и не быть непосредственно совместимы с другими встроенными типами данных в Java.

Для объявления переменных типа `byte` служит ключевое слово `byte`. Например, в следующей строке кода объявляются две переменные `b` и `c` типа `byte`:

```
byte b, c;
```

Тип `short`

Тип `short` представляет 16-разрядные целочисленные значения со знаком в пределах от -32768 до 32767. Этот тип данных применяется в Java реже всех остальных. Ниже приведены некоторые примеры объявления переменных типа `short`.

```
short s;  
short t;
```

Тип `int`

Наиболее часто употребляемым целочисленным типом является `int`. Это тип 32-разрядных целочисленных значений со знаком в пределах от -2147483648 до 2147483647. Среди прочего переменные типа `int` зачастую используются для управления циклами и индексирования массивов. На первый взгляд может показаться, что пользоваться типом `byte` или `short` эффективнее, чем типом `int`,

в тех случаях, когда не требуется более широкий диапазон допустимых значений, предоставляемый последним, но в действительности это не всегда так. Дело в том, что при указании значений типа `byte` и `short` в выражениях их тип *продвигается* к типу `int` при вычислении выражения. (Подробнее о продвижении типов речь пойдет далее в главе.) Поэтому тип `int` зачастую оказывается наиболее подходящим для обращения с целочисленными значениями.

Тип `long`

Этот тип 64-разрядных целочисленных значений со знаком удобен в тех ситуациях, когда длины типа `int` недостаточно для хранения требуемого значения. Диапазон допустимых значений типа `long` достаточно велик, что делает его удобным для обращения с большими целыми числами. Ниже приведен пример программы, которая вычисляет количество миль, проходимых лучом света за указанное число дней.

```
// Вычислить расстояние, проходимое светом,
// используя переменные типа long
class Light {
    public static void main(String args[]) {
        int lightspeed;
        long days;
        long seconds;
        long distance;

        // приблизительная скорость света, миль в секунду
        lightspeed = 186000;

        days = 1000; // указать количества дней

        seconds = days * 24 * 60 * 60; // преобразовать в секунды

        distance = lightspeed * seconds; // вычислить расстояние

        System.out.print("За " + days);
        System.out.print(" дней свет пройдет около ");
        System.out.println(distance + " миль.");
    }
}
```

Эта программа выводит следующий результат:

За 1000 дней свет пройдет около 16070400000000 миль.

Очевидно, что такой результат не поместился бы в переменной типа `int`.

Числа с плавающей точкой

Числа с плавающей точкой, называемые также *действительными* числами, используются при вычислении выражений, которые требуют результата с точностью до определенного знака после десятичной точки. Например, вычисление квадратного корня или трансцендентных функций вроде синуса или косинуса приводит к результату, который требует применения числового типа с плавающей

точкой. В Java реализован стандартный (в соответствии с нормой IEEE-754) ряд типов и операций над числами с плавающей точкой. Существуют два числовых типа с плавающей точкой: `float` и `double`, которые соответственно представляют числа одинарной и двойной точности. Их длина и диапазон допустимых значений приведены в табл.3.2.

Таблица 3.2. Длина и диапазон допустимых значений числовых типов с плавающей точкой

Наименование	Длина в битах	Приблизительный диапазон допустимых значений
<code>double</code>	64	от $4.9\text{e}-324$ до $1.8\text{e}+308$
<code>float</code>	32	от $1.4\text{e}-045$ до $3.4\text{e}+038$

Рассмотрим каждый из этих числовых типов с плавающей точкой в отдельности.

Тип `float`

Этот тип определяет числовое значение с плавающей точкой *одинарной* точности, для хранения которого в оперативной памяти требуется 32 бита. В некоторых процессорах обработка числовых значений с плавающей точкой одинарной точности выполняется быстрее и требует в два раза меньше памяти, чем обработка аналогичных значений двойной точности. Но если числовые значения слишком велики или слишком малы, то тип данных `float` не обеспечивает требуемую точность вычислений. Этот тип данных удобен в тех случаях, когда требуется числовое значение с дробной частью, но без особой точности. Например, значение типа `float` может быть удобно для представления денежных сумм в долларах и центах. Ниже приведен пример объявления переменных типа `float`.

```
float hightemp, lowtemp;
```

Тип `double`

Для хранения числовых значений с плавающей точкой двойной точности, как обозначает ключевое слово `double`, в оперативной памяти требуется 64 бита. На самом деле в некоторых современных процессорах, оптимизированных для выполнения математических расчетов с высокой скоростью, обработка значений двойной точности выполняется быстрее, чем обработка значений одинарной точности. Все трансцендентные математические функции наподобие `sin()`, `cos()` и `sqrt()` возвращают значения типа `double`. Рациональнее всего пользоваться типом `double`, когда требуется сохранять точность многократно повторяющихся вычислений или манипулировать большими числами.

Ниже приведен краткий пример программы, где переменные типа `double` используются для вычисления площади круга.

```
// Вычислить площадь круга
class Area {
    public static void main(String args[]) {
        double pi, r, a;
```

```
r = 10.8;          // радиус окружности
pi = 3.1416;       // приближительное значение числа пи
a = pi * r * r;    // вычислить площадь круга

System.out.println("Площадь круга равна " + a);
}
}
```

Символы

Для хранения символов в Java используется тип данных `char`. Но тем, у кого имеется опыт программирования на C/C++, следует иметь в виду, что тип `char` в Java не равнозначен типу `char` в C или C++. Если в C/C++ тип `char` является целочисленным и имеет длину 8 бит, то в Java дело обстоит иначе. Для представления символов типа `char` в Java используется кодировка в Юникод (Unicode), определяющем полный набор международных символов на всех известных языках мира. Юникод унифицирует десятки наборов символов, в том числе латинский, греческий, арабский алфавит, кириллицу, иврит, японские и тайские иероглифы и многие другие символы. Для хранения этих символов требуется 16 бит, и поэтому в Java тип `char` является 16-разрядным. Диапазон допустимых значений этого типа составляет от 0 до 65536. Отрицательных значений типа `char` не существует. Стандартный набор символов, известный как код ASCII, содержит значения от 0 до 127, а расширенный 8-разрядный набор символов ISO-Latin-1 — значения от 0 до 255. А поскольку язык Java предназначен для написания программ, которые можно применять во всем мире, употребление в нем кодировки в Юникоде для представления символов вполне обоснованно. Разумеется, пользоваться кодировкой в Юникоде для локализации программ на таких языках, как английский, немецкий, испанский или французский, не совсем эффективно, поскольку для представления символов из алфавита этих языков достаточно и 8 бит. Но это та цена, которую приходится платить за переносимость программ в глобальном масштабе.

На заметку! Подробнее о кодировке Юникод см. по адресу <http://www.unicode.org>.

Использование переменных типа `char` демонстрируется в следующем примере программы:

```
// Продемонстрировать применение типа данных char
class CharDemo {
    public static void main(String args[]) {
        char ch1, ch2;

        ch1 = 88; // код символа X
        ch2 = 'Y';

        System.out.print("ch1 и ch2: ");
        System.out.println(ch1 + " " + ch2);
    }
}
```

Эта программа выводит следующий результат:

ch1 и ch2: X Y

Обратите внимание на то, что переменной `ch1` присвоено значение **88**, обозначающее код ASCII (и Юникод) латинской буквы **X**. Как отмечалось ранее, набор символов в коде ASCII занимает первые 127 значений из набора символов в Юникоде. Поэтому все прежние приемы, применяемые для обращения с символами в других языках программирования, вполне пригодны и для Java.

Несмотря на то что тип `char` предназначен для хранения символов в Юникоде, им можно пользоваться и как целочисленным типом для выполнения арифметических операций. Например, он допускает сложение символов или приращение значения символьной переменной. Такое применение типа `char` демонстрируется в следующем примере программы:

```
// Символьные переменные ведут себя как целочисленные значения
class CharDemo2 {
    public static void main(String args[]) {
        char ch1;

        ch1 = 'X';
        System.out.println("ch1 содержит " + ch1);

        ch1++; // увеличить на единицу значение переменной ch1
        System.out.println("ch1 теперь содержит " + ch1);
    }
}
```

Эта программа выводит следующий результат:

```
ch1 содержит X
ch1 теперь содержит Y
```

Сначала в данном примере программы переменной `ch1` присваивается значение символа **X**, а затем происходит приращение значения этой переменной, т.е. его увеличение на единицу. В итоге в переменной `ch1` остается значение символа **Y** — следующего по порядку в наборе символов в коде ASCII (и Юникоде).

На заметку! В формальной спецификации Java тип `char` упоминается как целочисленный, а это означает, что он относится к той же общей категории, что и типы `int`, `short`, `long` и `byte`. Но поскольку основное назначение типа `char` — представлять символы в Юникоде, то он относится к собственной отдельной категории.

Логические значения

В языке Java имеется примитивный тип `boolean`, предназначенный для хранения логических значений. Переменные этого типа могут принимать только одно из двух возможных значений: `true` (истинное) или `false` (ложное). Значения этого логического типа возвращаются из всех операций сравнения вроде `a < b`. Тип `boolean` *обязателен* для употребления и в условных выражениях, которые управляют такими операторами, как `if` и `for`. В следующем примере программы демонстрируется применение логического типа `boolean`:

```
// Продемонстрировать применение значений типа boolean
class BoolTest {
    public static void main(String args[]) {
```



```

boolean b;

b = false;
System.out.println("b равно " + b);
b = true;
System.out.println("b равно " + b);

// значение типа boolean может управлять оператором if
if(b) System.out.println("Этот код выполняется.");
b = false;
if(b) System.out.println("Этот код не выполняется.");

// результат сравнения - значение типа boolean
System.out.println("10 > 9 равно " + (10 > 9));
}
}

```

Эта программа выводит следующий результат:

```

b равно false
b равно true
Этот код выполняется.
10 > 9 равно true

```

В приведенном выше примере программы особое внимание обращают на себя три момента. Во-первых, при выводе значения типа `boolean` с помощью метода `println()` на экране, как видите, появляется слово "true" или "false". Во-вторых, одного лишь значения переменной типа `boolean` оказывается достаточно для управления условным оператором `if`. Записывать условный оператор `if` так, как показано ниже, совсем не обязательно.

```
if (b == true) ...
```

И в-третьих, результатом выполнения операции сравнения вроде `<` является логическое значение типа `boolean`. Именно поэтому выражение `10 > 9` приводит к выводу слова "true". Более того, выражение `10 > 9` должно быть заключено в дополнительный ряд круглых скобок, поскольку операция `+` имеет более высокую степень предшествования, чем операция `>`.

Подробнее о литералах

Литералы были лишь упомянуты в главе 2. А теперь, когда формально описаны встроенные примитивные типы данных, настало время рассмотреть литералы более подробно.

Целочисленные литералы

Целочисленный тип едва ли не чаще всего используется в обычной программе. Любое целочисленное значение является числовым литералом. Примерами тому могут служить значения 1, 2, 3 и 42. Все они являются десятичными значениями, описывающими числа по основанию 10. В целочисленных литералах могут использоваться числа по еще двум основаниям: *восьмеричные* (по основанию 8) и *шестнадцатеричные* (по основанию 16). В Java восьмеричные значения обозна-

чаются начальным нулем, тогда как обычные десятичные числа не могут содержать начальный нуль. Таким образом, вполне допустимое, казалось бы, значение 09 приведет к ошибке компиляции, поскольку значение 9 выходит за пределы допустимых восьмеричных значений от 0 до 7. Программисты часто пользуются шестнадцатеричным представлением чисел, которое соответствует словам по длине 8, 16, 32 и 64 бит, состоящим из 8-разрядных блоков. Значения шестнадцатеричных констант обозначают начальным нулем и символом x (0x или 0X). Шестнадцатеричные цифры должны указываться в пределах от 0 до 15, поэтому цифры от 10 до 15 заменяют буквами от A до F (или от a до f).

Целочисленные литералы создают значение типа `int`, которое в Java является 32-битовым целочисленным значением. Язык Java — строго типизированный, и в связи с этим может возникнуть вопрос: каким образом присвоить целочисленный литерал одному из других целочисленных типов данных в Java, например `byte` или `long`, не приводя к ошибке несоответствия типов? К счастью, ответить на этот вопрос нетрудно. Когда значение литерала присваивается переменной типа `byte` или `short`, ошибки не происходит, если значение литерала находится в диапазоне допустимых значений данного типа. Кроме того, целочисленный литерал всегда можно присвоить переменной типа `long`. Но чтобы обозначить литерал типа `long`, придется явно указать компилятору, что значение литерала имеет этот тип. Для этого литерал дополняется строчной или прописной буквой `L`. Например, значение `0xffffffffffffffffL` или `9223372036854775807L` является наибольшим литералом типа `long`. Целочисленное значение типа `long` можно также присвоить переменной типа `char`, если оно находится в пределах допустимых значений данного типа.

Начиная с версии JDK 7 целочисленные литералы можно определить и в двоичной форме. Для этого перед присваиваемым значением указывается префикс `0b` или `0B`. Например, в следующей строке кода десятичное значение 10 определяется с помощью двоичного литерала:

```
int x = 0b1010;
```

Наличие двоичных литералов облегчает также ввод значений, используемых в качестве битовых масок. В таком случае десятичное (или шестнадцатеричное) представление значения внешне не передает его назначение, тогда как двоичный литерал передает его.

Начиная с версии JDK 7 в обозначении целочисленных литералов можно также указывать один знак подчеркивания или более. Это облегчает чтение крупных целочисленных литералов. При компиляции символы подчеркивания в литерале игнорируются. Например, в строке кода

```
int x = 123_456_789;
```

переменной `x` присваивается значение `123456789`, а знаки подчеркивания игнорируются. Эти знаки могут использоваться только для разделения цифр. Их нельзя указывать в начале или в конце литерала, но вполне допустимо между двумя цифрами, причем не один, а несколько. Например, следующая строка кода считается правильной:

```
int x = 123__456__789;
```

Пользоваться знаками подчеркивания в целочисленных литералах особенно удобно при указании в прикладном коде таких элементов, как номера телефонов,

идентификационные номера клиентов, номера частей и т.д. Они также полезны для визуальных группировок при определении двоичных литералов. Например, двоичные значения зачастую визуально группируются в блоки по четыре цифры:

```
int x = 0b1101_0101_0001_1010;
```

Литералы с плавающей точкой

Числа с плавающей точкой представляют десятичные значения с дробной частью. Они могут быть выражены в стандартной или экспоненциальной (или научной) форме записи. Число в *стандартной форме записи* состоит из целого числа с последующей десятичной точкой и дробной частью. Например, значения 2.0, 3.14159 и 0.6667 представляют допустимые числа с плавающей точкой в стандартной записи. В *экспоненциальной форме записи* используется стандартная форма записи чисел с плавающей точкой, дополненная суффиксом, обозначающим степень числа 10, на которую следует умножить данное число. Для указания экспоненциальной части в данной форме записи используется символ **E** или **e**, за которым следует десятичное число (положительное или отрицательное). Примерами такой формы записи могут служить значения **6.022E23**, **314159E-05** и **2e+100**.

По умолчанию в Java литералам с плавающей точкой присваивается тип `double`. Чтобы указать литерал типа `float`, его следует дополнить символом **F** или **f**. Литерал типа `double` можно также указать явно, дополнив символом **D** или **d**. Но это, конечно, излишне. Значения используемого по умолчанию типа `double` занимают в оперативной памяти 64 бита, тогда как для хранения значений более короткого типа `float` требуется только 32 бита.

Шестнадцатеричные литералы с плавающей точкой также поддерживаются, но они применяются редко. Они должны быть записаны в форме, подобной экспоненциальному представлению, но с обозначением **P** или **p** вместо **E** или **e**. Например, **0x12.2P2** вполне допустимый шестнадцатеричный литерал с плавающей точкой. Значение после буквы **P** называется *двоичным порядком* и обозначает степень числа два, на которое умножается заданное число. Поэтому литерал **0x12.2P2** представляет число 72,5.

Начиная с версии JDK 7 в литералы с плавающей точкой можно вводить один знак подчеркивания или больше, подобно описанному выше в отношении целочисленных литералов. Знаки подчеркивания облегчают чтение больших литералов с плавающей точкой. При компиляции символы подчеркивания игнорируются. Например, в следующей строке кода:

```
double num = 9_423_497_862.0;
```

переменной `num` присваивается значение **9423497862,0**. Знаки подчеркивания будут проигнорированы. Как и в целочисленных литералах, знаки подчеркивания могут служить только для разделения цифр. Их нельзя указывать в начале или в конце литералов, но вполне допустимо между двумя цифрами, причем не один, а несколько. Знаки подчеркивания допустимо также указывать в дробной части числа. Например, следующая строка кода считается правильной:

```
double num = 9_423_497.1_0_9;
```

В данном случае дробная часть составляет **.109**.

Логические литералы

Эти литералы очень просты. Тип `boolean` может иметь только два логических значения: `true` и `false`. Эти значения не преобразуются ни в одно из числовых представлений. В Java логический литерал `true` не равен 1, а литерал `false` — 0. Логические литералы в Java могут присваиваться только тем переменным, которые объявлены как `boolean`, или употребляться в выражениях с логическими операциями.

Символьные литералы

В Java символы представляют собой индексы из набора символов в Юникоде. Это 16-разрядные значения, которые могут быть преобразованы в целые значения и над которым можно выполнять такие целочисленные операции, как сложение и вычитание. Символьные литералы заключаются в одинарные кавычки. Все отображаемые символы в коде ASCII можно вводить непосредственно, заключая их в одинарные кавычки, например `'a'`, `'z'` и `'@'`. Если символы нельзя ввести непосредственно, то для их ввода можно воспользоваться рядом управляющих последовательностей, которые позволяют вводить нужные символы (например, символ одинарной кавычки как `'\"'` или символ новой строки как `'\n'`). Существует также механизм для непосредственного ввода значения символа в восьмеричной или шестнадцатеричной форме. Для ввода значений в восьмеричной форме служит символ обратной косой черты, за которым следует трехзначное число. Например, последовательность символов `'\141'` равнозначна букве `'a'`. Для ввода значений в шестнадцатеричной форме применяются символы обратной косой черты и `u` (`\u`), а вслед за ними следуют четыре шестнадцатеричные цифры. Например, литерал `'\u0061'` представляет букву `'a'` из набора символов ISO-Latin-1, поскольку старший байт является нулевым, а литерал `'\u432'` — символ японской катаканы. Управляющие последовательности символов перечислены в табл. 3.3.

Таблица 3.3. Управляющие последовательности символов

Управляющая последовательность	Описание
<code>\ddd</code>	Восьмеричный символ (<code>ddd</code>)
<code>\uxxxx</code>	Шестнадцатеричный символ в Юникоде (<code>xxxx</code>)
<code>\'</code>	Одинарная кавычка
<code>\"</code>	Двойная кавычка
<code>\\</code>	Обратная косая черта
<code>\r</code>	Возврат каретки
<code>\n</code>	Новая строка (или перевод строки)
<code>\f</code>	Подача страницы
<code>\t</code>	Табуляция
<code>\b</code>	Возврат на одну позицию (“забой”)

Строковые литералы

Строковые литералы обозначаются в Java таким же образом, как и в других языках программирования. С этой целью последовательность символов заключается в двойные кавычки. Ниже приведены некоторые примеры строковых литералов.

```
"Hello World"  
"two\nlines"  
"\\"This is in quotes\""
```

Управляющие символы и восьмеричная или шестнадцатеричная форма записи, определенные для символьных литералов, действуют точно так же и в строковых литералах. Однако в Java символьные строки должны начинаться и оканчиваться в одной строке. В отличие от других языков программирования, в Java отсутствует специальный управляющий символ для продолжения строки.

На заметку! Как вам, должно быть, известно, в некоторых языках, включая C/C++, символьные строки реализованы в виде массивов символов. Но совсем иначе дело состоит в Java. На самом деле символьные строки представляют собой объектные типы. Как поясняется далее, в Java символьные строки реализованы в виде объектов, поэтому в этом языке предоставляется целый ряд эффективных и простых в использовании средств для их обработки.

Переменные

Переменная служит основной единицей хранения данных в программе на Java. Переменная определяется в виде сочетания идентификатора, типа и необязательного начального значения. Кроме того, у всех переменных имеется своя область действия, которая определяет их доступность для других объектов и продолжительность существования. Все эти составляющие переменных будут рассмотрены в последующих разделах.

Объявление переменной

В Java все переменные должны быть объявлены до их использования. Основная форма объявления переменных выглядит следующим образом:

```
тип идентификатор [=значение] [, идентификатор [=значение] ...] ;
```

где параметр *тип* обозначает один из примитивных типов данных в Java, имя класса или интерфейса. (Типы классов и интерфейсов рассматриваются в последующих главах этой части.) А *идентификатор* — это имя переменной. Любой переменной можно присвоить начальное значение (инициализировать ее) через знак равенства. Следует, однако, иметь в виду, что инициализирующее выражение должно возвращать значение того же самого (или совместимого) типа, что и у переменной. Для объявления нескольких переменных указанного типа можно воспользоваться списком, разделяемым запятыми.

Ниже приведен ряд примеров объявления переменных различных типов. Обратите внимание на то, что в некоторых примерах переменные не только объявляются, но и инициализируются.

```
int a, b, c;           // объявление трех переменных a, b и c типа int
int d = 3, e, f = 5;   // объявление еще трех переменных типа int
                        // с инициализацией переменных d и f
byte z = 22;           // инициализация переменной z
double pi = 3.14159;   // объявление переменной pi и ее инициализация
                        // приближительным значением числа пи
char x = 'x';          // присваивание символа 'x' переменной x
```

Выбранные имена идентификаторов очень просты и указывают их тип. В языке Java допускается наличие любого объявленного типа в каком угодно правильно оформленном идентификаторе.

Динамическая инициализация

В приведенных ранее примерах в качестве начальных значений были использованы только константы, но в Java допускается и динамическая инициализация переменных с помощью любого выражения, действительного в момент объявления переменной. В качестве примера ниже приведена краткая программа, в которой длина гипотенузы прямоугольного треугольника вычисляется по длине его катетов.

```
// В этом примере демонстрируется динамическая инициализация
class DynInit {
    public static void main(String args[]) {
        double a = 3.0, b = 4.0;

        // динамическая инициализация переменной c
        double c = Math.sqrt(a * a + b * b);
        System.out.println("Гипотенуза равна " + c);
    }
}
```

В данном примере программы объявляются три локальные переменные — *a*, *b* и *c*. Две первые из них (*a* и *b*) инициализируются константами, тогда как третья (*c*) инициализируется динамически, принимая значение длины гипотенузы, вычисляемое по теореме Пифагора. Для вычисления квадратного корня аргумента в этой программе вызывается встроенный в Java метод `sqrt()`, который является членом класса `Math`. Самое главное в данном примере, что в инициализирующем выражении можно использовать любые элементы, действительные во время инициализации, в том числе вызовы методов, другие переменные или константы.

Область и срок действия переменных

В представленных до сих пор примерах программ все переменные объявлялись в начале метода `main()`. Но в Java допускается объявление переменных в любом блоке кода. Как пояснялось в главе 2, блок кода заключается в фигурные скобки, задавая тем самым *область действия*. Таким образом, при открытии каждого нового блока кода создается новая область действия. Область действия определяет, какие именно объекты доступны для других частей программы. Она определяет также продолжительность существования этих объектов.

Во многих других языках программирования различаются две основные категории области действия: глобальная и локальная. Но эти традиционные области дей-

ствия не вполне вписываются в строгую объектно-ориентированную модель Java. Несмотря на возможность задать глобальную область действия, в настоящее время такой подход является скорее исключением, чем правилом. Две основные области действия в Java определяются классом и методом, хотя такое их разделение несколько искусственно. Но такое разделение имеет определенный смысл, поскольку область действия класса обладает рядом характерных особенностей и свойств, не распространяющихся на область действия метода. Вследствие этих отличий рассмотрение области действия класса (и объявляемых в нем переменных) будет отложено до главы 6, в которой описываются классы. А пока рассмотрим только те области действия, которые определяются самим методом или в его теле.

Область действия, определяемая методом, начинается с его открывающей фигурной скобки. Но если у метода имеются параметры, то они также включаются в область действия метода. Подробнее параметры методов рассматриваются в главе 6, а до тех пор достаточно сказать, что они действуют точно так же, как и любая другая переменная в методе.

Как правило, переменные, объявленные в области действия, не доступны из кода за пределами этой области. Таким образом, объявление переменной в области действия обеспечивает ее локальность и защиту от несанкционированного доступа и/или внешних изменений. В действительности правила функционирования обработки области действия составляют основу инкапсуляции.

Области действия могут быть вложенными. Так, вместе с каждым блоком кода, по существу, создается новая, вложенная область действия. В таком случае внешняя область действия включает в себя внутреннюю область. Это означает, что объекты, объявленные во внешней области действия, будут доступны для кода из внутренней области действия, но не наоборот. Объекты, объявленные во внутренней области действия, будут недоступны за ее пределами. Чтобы стало понятнее, каким образом функционируют вложенные области действия, рассмотрим следующий пример программы:

```
// Продемонстрировать область действия блока кода
class Scope {
    public static void main(String args[]) {
        int x; // эта переменная доступна всему коду из метода main()

        x = 10;
        if(x == 10) {    // начало новой области действия,
            int y = 20; // доступной только этому блоку кода

            // обе переменные x и y доступны в этой области действия
            System.out.println("x и y: " + x + " " + y);
            x = y * 2;
        }
        // y = 100; // ОШИБКА! переменная y недоступна
                   // в этой области действия, тогда как
                   // переменная x доступна и здесь
        System.out.println("x равно " + x);
    }
}
```

Как следует из комментариев к данной программе, переменная `x` объявлена в начале области действия метода `main()` и доступна всему последующему коду из

этого метода. Объявление переменной `y` делается в блоке кода условного оператора `if`. А поскольку этот блок кода задает область действия, переменная `y` доступна только коду из этого блока. Именно поэтому закомментирована строка кода `y = 100;`, находящаяся за пределами этого блока. Если удалить символы комментария, это приведет к ошибке во время компиляции, поскольку переменная `y` недоступна за пределами своего блока кода. А к переменной `x` можно обращаться из блока условного оператора `if`, поскольку коду в этом блоке (т.е. во вложенной области действия) доступны переменные, объявленные в объемлющей области действия.

Переменные можно объявлять в любом месте блока кода, но они действительны только после объявления. Так, если переменная объявлена в начале метода, она доступна всему коду в теле этого метода. И наоборот, если переменная объявлена в конце блока кода, она, по существу, бесполезна, так как вообще недоступна для кода. Например, следующий фрагмент кода ошибочен, поскольку переменной `count` нельзя пользоваться до ее объявления:

```
// Этот фрагмент кода написан неверно!  
count = 100;  
// Переменную count нельзя использовать до ее объявления!  
int count;
```

Следует иметь в виду еще одну важную особенность: переменные создаются при входе в их область действия и уничтожаются при выходе из нее. Это означает, что переменная утратит свое значение сразу же после выхода из ее области действия. Следовательно, переменные, объявленные в теле метода, не будут хранить свои значения в промежутках между последовательными обращениями к этому методу. Кроме того, переменная, объявленная в блоке кода, утратит свое значение после выхода из него. Таким образом, срок действия переменной ограничивается ее областью действия.

Если объявление переменной включает в себя ее инициализацию, то инициализация переменной будет повторяться при каждом вхождении в блок кода, где она объявлена. Рассмотрим в качестве примера приведенную ниже программу.

```
// Продемонстрировать срок действия переменной  
class LifeTime {  
    public static void main(String args[]) {  
        int x;  
  
        for(x = 0; x < 3; x++) {  
            int y = -1; // переменная y инициализируется при  
                       // каждом вхождении в блок кода  
            System.out.println("y равно: " + y); // здесь всегда  
                                                    // выводится значение -1  
            y = 100;  
            System.out.println("y теперь равно: " + y);  
        }  
    }  
}
```

Эта программа выводит следующий результат:

```
y равно: -1  
y теперь равно: 100  
y равно: -1  
y теперь равно: 100
```


у равно: -1
 у, теперь равно: 100

Как видите, переменная `у` повторно инициализируется значением `-1` при каждом вхождении во внутренний цикл `for`. И хотя переменной `у` впоследствии присваивается значение `100`, тем не менее она теряется.

И последнее: несмотря на то, что блоки могут быть вложенными, во внутреннем блоке кода нельзя объявлять переменные с тем же именем, что и во внешней области действия. Например, следующая программа ошибочна:

```
// Скомпилировать эту программу нельзя
class ScopeErr {
    public static void main(String args[]) {
        int bar = 1;
        {
            // создается новая область действия
            int bar = 2; // Ошибка во время компиляции -
                        // переменная bar уже определена!
        }
    }
}
```

Преобразование и приведение типов

Тем, у кого имеется некоторый опыт программирования, должно быть известно, что переменной одного типа нередко приходится присваивать значение другого типа. Если оба типа данных совместимы, их преобразование будет выполнено в Java автоматически. Например, значение типа `int` всегда можно присвоить переменной типа `long`. Но не все типы данных совместимы, а, следовательно, не все преобразования типов разрешены неявно. Например, не существует какого-то определенного автоматического преобразования типа `double` в тип `byte`. Правда, преобразования между несовместимыми типами выполнять все-таки можно. Для этой цели служит *приведение типов*, при котором выполняется явное преобразование несовместимых типов. Рассмотрим автоматическое преобразование и приведение типов.

Автоматическое преобразование типов в Java

Когда данные одного типа присваиваются переменной другого типа, выполняется *автоматическое преобразование типов*, если удовлетворяются два условия:

- оба типа совместимы;
- длина целевого типа больше длины исходного типа.

При соблюдении этих условий выполняется *расширяющее преобразование*. Например, тип данных `int` всегда достаточно велик, чтобы хранить все допустимые значения типа `byte`, поэтому никакие операторы явного приведения типов в данном случае не требуются.

С точки зрения расширяющего преобразования числовые типы данных, в том числе целочисленные и с плавающей точкой, совместимы друг с другом. В то же

время не существует автоматических преобразований числовых типов в тип `char` или `boolean`. Типы `char` и `boolean` также не совместимы друг с другом. Как упоминалось ранее, автоматическое преобразование типов выполняется в Java при сохранении целочисленной константы в переменных типа `byte`, `short`, `long` или `char`.

Приведение несовместимых типов

Несмотря на все удобство автоматического преобразования типов, оно не в состоянии удовлетворить все насущные потребности. Например, что делать, если значение типа `int` нужно присвоить переменной типа `byte`? Это преобразование не будет выполняться автоматически, поскольку длина типа `byte` меньше, чем у типа `int`. Иногда этот вид преобразования называется *сужающим преобразованием*, поскольку значение явно сужается, чтобы уместиться в целевом типе данных.

Чтобы выполнить преобразование двух несовместимых типов данных, нужно воспользоваться приведением типов. *Приведение* – это всего лишь явное преобразование типов. Общая форма приведения типов имеет следующий вид:

(целевой_тип) значение

где параметр *целевой_тип* обозначает тип, в который нужно преобразовать указанное значение. Например, в следующем фрагменте кода тип `int` приводится к типу `byte`. Если значение целочисленного типа больше допустимого диапазона значений типа `byte`, оно будет сведено к результату деления по модулю (остатку от целочисленного деления) на диапазон типа `byte`.

```
int a;  
byte b;  
// ...  
b = (byte) a;
```

При присваивании значения с плавающей точкой переменной целочисленного типа выполняется другой вид преобразования типов – *усечение*. Как известно, целые числа не содержат дробной части. Таким образом, когда значение с плавающей точкой присваивается переменной целочисленного типа, его дробная часть отбрасывается. Так, если значение `1,23` присваивается целочисленной переменной, то в конечном итоге в ней остается только значение `1`, а дробная часть `0,23` усекается. Конечно, если длина целочисленной части числового значения слишком велика, чтобы уместиться в целевом целочисленном типе, это значение будет сокращено до результата деления по модулю на диапазон целевого типа.

В следующем примере программы демонстрируется ряд преобразований типов, которые требуют их приведения.

```
// Продемонстрировать приведение типов  
class Conversion {  
    public static void main(String args[]) {  
        byte b;  
        int i = 257;  
        double d = 323.142;  
  
        System.out.println("\nПреобразование типа int в тип byte.");  
        b = (byte) i;  
        System.out.println("i и b " + i + " " + b);  
    }  
}
```

```
System.out.println("\nПреобразование типа double в тип int.");
i = (int) d;
System.out.println("d и i " + d + " " + i);

System.out.println("\nПреобразование типа double в тип byte.");
b = (byte) d;
System.out.println("d и b " + d + " " + b);
}
}
```

Эта программа выводит следующий результат:

Преобразование типа int в тип byte.
i и b 257 1

Преобразование типа double в тип int.
d и i 323.142 323

Преобразование типа double в тип byte.
d и b 323.142 67

Рассмотрим каждое из этих преобразований. Когда значение 257 приводится к типу `byte`, его результатом будет остаток от деления 257 на 256 (диапазон допустимых значений типа `byte`), который в данном случае равен 1. А когда значение переменной `d` преобразуется в тип `int`, его дробная часть отбрасывается. И когда значение переменной `d` преобразуется в тип `byte`, его дробная часть отбрасывается и значение уменьшается до результата деления по модулю на 256, который в данном случае равен 67.

Автоматическое продвижение типов в выражениях

Помимо операций присваивания, определенное преобразование типов может выполняться и в выражениях. Это может происходить в тех случаях, когда требующаяся точность промежуточного значения выходит за пределы допустимого диапазона значений любого из операндов в выражении. В качестве примера рассмотрим следующий фрагмент кода:

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

Результат вычисления промежуточного члена `a * b` вполне может выйти за пределы диапазона допустимых значений его операндов типа `byte`. Для разрешения подобных затруднений при вычислении выражений в Java тип каждого операнда `byte`, `short` или `char` автоматически продвигается к типу `int`. Это означает, что вычисление промежуточного выражения `a * b` выполняется с помощью целочисленных, а не байтовых значений. Поэтому результат 2000 вычисления промежуточного выражения `50 * 40` оказывается вполне допустимым, несмотря на то, что оба операнда — `a` и `b` — объявлены как относящиеся к типу `byte`.

Несмотря на все удобства автоматического продвижения типов, оно может приводить к досадным ошибкам во время компиляции. Например, следующий правильный на первый взгляд код приводит к ошибке:

```
byte b = 50;
b = b * 2; // ОШИБКА! Значение типа int не может быть присвоено
           // переменной типа byte!
```

В этом фрагменте кода предпринимается попытка сохранить произведение $50 * 2$ (вполне допустимое значение типа `byte`) в переменной типа `byte`. Но поскольку во время вычисления этого выражения тип операндов автоматически продвигается к типу `int`, то и тип результата также продвигается к типу `int`. Таким образом, результат вычисления данного выражения относится к типу `int` и не может быть присвоен переменной типа `byte` без приведения типов. Это справедливо даже в том случае, когда присваиваемое значение умещается в переменной целевого типа, как в данном конкретном примере.

В тех случаях, когда последствия переполнения очевидны, следует использовать явное приведение типов. Так, в следующем примере кода переменной `b` присваивается правильное значение 100 благодаря приведению типов:

```
byte b = 50;
b = (byte)(b * 2);
```

Правила продвижения типов

В языке Java определен ряд правил *продвижения типов*, применяемых к выражениям. Сначала все значения типа `byte`, `short` и `char` продвигаются к типу `int`, как пояснялось выше. Затем тип всего выражения продвигается к типу `long`, если один из его операндов относится к типу `long`. Если же один из операндов относится к типу `float`, то тип всего выражения продвигается к типу `float`. А если любой из операндов относится к типу `double`, то и результат вычисления всего выражения относится к типу `double`.

В следующем примере программы демонстрируется продвижение типа одного из операндов для соответствия типу второго операнда в каждом операторе с двумя операндами:

```
class Promote {
    public static void main(String args[]) {
        byte b = 42;
        char c = 'a';
        short s = 1024;
        int i = 50000;
        float f = 5.67f;
        double d = .1234;
        double result = (f * b) + (i / c) - (d * s);
        System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
        System.out.println("result = " + result);
    }
}
```

Проанализируем порядок продвижения типов в следующей строке кода из данного примера:

```
double result = (f * b) + (i / c) - (d * s);
```

В первом промежуточном выражении $f * b$ тип переменной b продвигается к типу `float`, а результат вычисления этого выражения также относится к типу `float`. В следующем промежуточном выражении i/c тип переменной c продвигается к типу `int`, а результат вычисления этого выражения относится к типу `int`. Затем в промежуточном выражении $d * s$ тип переменной s продвигается к типу `double`, а результат его вычисления относится к типу `double`. И наконец, выполняются операции с этими тремя промежуточными результирующими значениями типа `float`, `int` и `double`. Результат сложения значений типа `float` и `int` относится к типу `float`. Затем тип разности суммарного значения типа `float` и последнего значения типа `double` продвигается к типу `double`, который и становится окончательным типом результата вычисления выражения в целом.

Массивы

Массив — это группа однотипных переменных, для обращения к которым используется общее имя. В Java допускается создание массивов любого типа и разной размерности. Доступ к конкретному элементу массива осуществляется по его индексу. Массивы предоставляют удобный способ группирования связанной вместе информации.

На заметку! Те, кто знаком с языками C/C++, должны быть особенно внимательны. Ведь в Java массивы действуют иначе, чем в этих языках.

Одномерные массивы

По существу, *одномерные массивы* представляют собой список однотипных переменных. Чтобы создать массив, нужно сначала объявить переменную массива требуемого типа. Общая форма объявления одномерного массива выглядит следующим образом:

```
тип имя_переменной[];
```

где параметр *тип* обозначает тип элемента массива, называемый также базовым типом. Тип элемента массива определяет тип данных каждого из элементов, составляющих массив. Таким образом, тип элемента массива определяет тип данных, которые будет содержать массив. Например, в следующей строке кода объявляется массив `month_days`, состоящий из элементов типа `int`:

```
int month_days[];
```

Несмотря на то что в этой строке кода утверждается, что `month_days` объявляется как массив переменных, на самом деле никакого массива пока еще не существует. Чтобы связать имя `month_days` с физически существующим массивом целочисленных значений, необходимо зарезервировать область памяти с помощью оператора `new` и назначить ее для массива `month_days`. Более подробно эта операция будет рассмотрена в следующей главе, но теперь она требуется для выделения памяти под массивы. Общая форма оператора `new` применительно к одномерным массивам выглядит следующим образом:

```
переменная_массива = new тип [размер];
```

где параметр *тип* обозначает тип данных, для которых резервируется память; параметр *размер* — количество элементов в массиве, а параметр *переменная_массива* означает переменную, непосредственно связанную с массивом. Иными словами, чтобы воспользоваться оператором `new` для резервирования памяти, придется указать тип и количество элементов, для которых требуется зарезервировать память. Элементы массива, для которых память была выделена оператором `new`, будут автоматически инициализированы нулевыми значениями (для числовых типов), логическими значениями `false` (для логического типа) или пустыми значениями `null` (для ссылочных типов, рассматриваемых в следующей главе). В приведенном ниже примере резервируется память для 12-элементного массива целых значений, которые связываются с массивом `month_days`. После выполнения этого оператора переменная массива `month_days` будет ссылаться на массив, состоящий из 12 целочисленных значений. При этом все элементы данного массива будут инициализированы нулевыми значениями:

```
month_days = new int[12];
```

Подведем краткий итог: процесс создания массива происходит в два этапа. Во-первых, следует объявить переменную нужного типа массива. Во-вторых, с помощью оператора `new` необходимо зарезервировать память для хранения массива и присвоить ее переменной массива. Таким образом, все массивы в Java распределяются динамически. Если вы еще не знакомы с понятием и процессом динамического распределения памяти, не отчаивайтесь. Этот процесс будет подробно описан в последующих главах.

Как только будет создан массив и зарезервирована память для него, к конкретному элементу массива можно обращаться, указывая его индекс в квадратных скобках. Индексы массива начинаются с нуля. Например, в приведенной ниже строке кода значение **28** присваивается второму элементу массива `month_days`.

```
month_days[1] = 28;
```

А в следующей строке кода выводится значение, хранящееся в элементе массива по индексу **3**:

```
System.out.println(month_days[3]);
```

Для демонстрации процесса в целом ниже приведен пример программы, в которой создается массив, содержащий количество дней в каждом месяце.

```
// Продемонстрировать применение одномерного массива
class Array {
    public static void main(String args[]) {
        int month_days[];
        month_days = new int[12];
        month_days[0] = 31;
        month_days[1] = 28;
        month_days[2] = 31;
        month_days[3] = 30;
        month_days[4] = 31;
        month_days[5] = 30;
        month_days[6] = 31;
        month_days[7] = 31;
        month_days[8] = 30;
        month_days[9] = 31;
```

```
        month_days[10] = 30;
        month_days[11] = 31;
        System.out.println("В апреле " + month_days[3] + " дней.");
    }
}
```

Если выполнить эту программу, она выведет на экран количество дней в апреле. Как упоминалось ранее, в Java индексация элементов массивов начинается с нуля, поэтому количество дней в апреле хранится в элементе массива `month_days[3]` и равно 30.

Объявление переменной массива можно объединять с выделением для него памяти:

```
int month_days[] = new int[12];
```

Именно так обычно и поступают в программах, профессионально написанных на Java. Массивы можно инициализировать при их объявлении. Этот процесс во многом аналогичен инициализации простых типов. *Инициализатор массива* – это список выражений, разделяемый запятыми и заключаемый в фигурные скобки. Запятые разделяют значения элементов массива. Массив автоматически создается настолько большим, чтобы вмещать все элементы, указанные в инициализаторе массива. В этом случае потребность в операторе `new` отпадает. Например, чтобы сохранить количество дней в каждом месяце, можно воспользоваться приведенным ниже кодом, где создается и инициализируется массив целых значений.

```
// Усовершенствованная версия предыдущей программы
class AutoArray {
    public static void main(String args[]) {

        int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,
                             30, 31 };
        System.out.println("В апреле " + month_days[3] + " дней.");
    }
}
```

При выполнении этого кода на экран выводится такой же результат, как и в предыдущей его версии. Исполняющая система Java производит тщательную проверку, чтобы убедиться, не была ли случайно предпринята попытка присвоения или обращения к значениям, которые выходят за пределы допустимого диапазона индексов массива. Например, исполняющая система Java будет проверять соответствие значения каждого индекса массива `month_days` допустимому диапазону от 0 до 11 включительно. Любая попытка обратиться к элементам массива `month_days` за пределами этого диапазона (т.е. указать отрицательные индексы или же индексы, превышающие длину массива) приведет к ошибке при выполнении данного кода.

Приведем еще один пример программы, в которой используется одномерный массив. В этой программе вычисляется среднее значение ряда чисел.

```
// Вычисление среднего из массива значений
class Average {
    public static void main(String args[]) {
        double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
        double result = 0;
        int i;
```

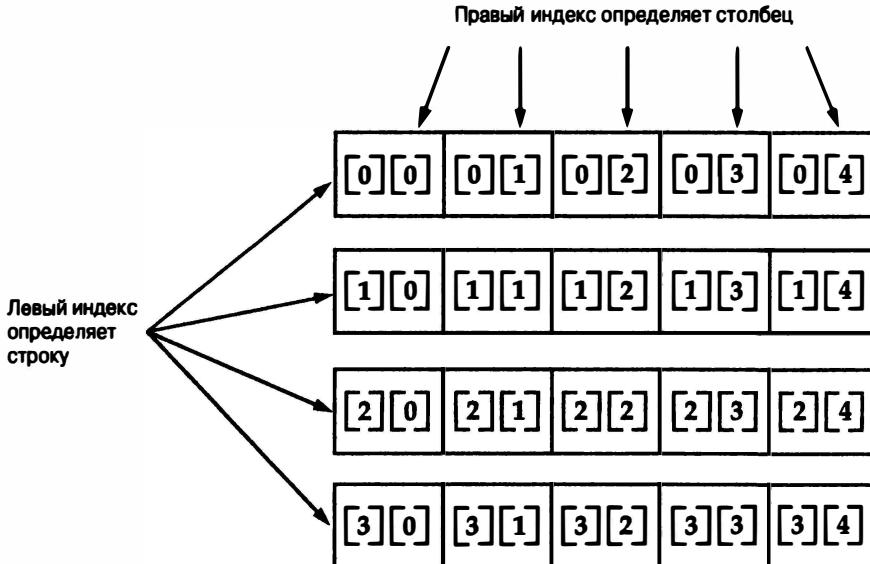
```
for(i=0; i<5; i++)  
    result = result + nums[i];  
System.out.println("Среднее значение равно " + result / 5);  
}  
}
```

Многомерные массивы

В языке Java *многомерные массивы* представляют собой массивы массивов. Нетрудно догадаться, что они внешне выглядят и действуют подобно обычным многомерным массивам. Но, как будет показано ниже, у них имеется ряд незначительных отличий. При объявлении переменной многомерного массива для указания каждого дополнительного индекса используется отдельный ряд квадратных скобок. Например, в следующей строке кода объявляется переменная двухмерного массива `twoD`:

```
int twoD[][] = new int[4][5];
```

В этой строке кода резервируется память для массива размерностью 4×5, который присваивается переменной `twoD`. Эта матрица реализована внутренним образом как *массив массивов* значений типа `int`. Концептуально этот массив будет выглядеть так, как показано на рис. 3.1.



Дано: `int twoD[][] = new int[4][5];`

Рис. 3.1. Концептуальное представление массива размерностью 4×5

В следующем примере программы элементы массива нумеруются слева направо и сверху вниз, а затем выводятся их значения:


```
// Продемонстрировать применение двумерного массива
class TwoDArray {
    public static void main(String args[]) {
        int twoD[][] = new int[4][5];
        int i, j, k = 0;

        for(i=0; i<4; i++)
            for(j=0; j<5; j++) {
                twoD[i][j] = k;
                k++;
            }

        for(i=0; i<4; i++) {
            for(j=0; j<5; j++)
                System.out.print(twoD[i][j] + " ");
            System.out.println();
        }
    }
}
```

Эта программа выводит следующий результат:

```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

При резервировании памяти под многомерный массив необходимо указать память только для первого (левого) измерения массива. А для каждого из остальных измерений массива память можно резервировать отдельно. Например, в приведенном ниже фрагменте кода память резервируется только для первого измерения массива `twoD` при его объявлении. А резервирование памяти для второго измерения массива осуществляется вручную.

```
int twoD[][] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

Если в данном примере отдельное резервирование памяти для второго измерения массива не дает никаких преимуществ, то в других случаях это может быть полезно. Например, при резервировании памяти для отдельных измерений массива вручную совсем не обязательно резервировать одинаковое количество элементов для каждого измерения. Как отмечалось ранее, программисту предоставляется полная свобода действий управлять длиной каждого массива, поскольку многомерные массивы на самом деле представляют собой массивы массивов. Например, в следующем примере программы создается двумерный массив с разной размерностью второго измерения:

```
// Резервирование памяти вручную для массива с разной
// размерностью второго измерения
class TwoDAgain {
    public static void main(String args[]) {
        int twoD[][] = new int[4][];
        twoD[0] = new int[1];
        twoD[1] = new int[2];
        twoD[2] = new int[3];
        twoD[3] = new int[4];
    }
}
```

```

int i, j, k = 0;

for(i=0; i<4; i++)
    for(j=0; j<i+1; j++) {
        twoD[i][j] = k;
        k++;
    }

for(i=0; i<4; i++) {
    for(j=0; j<i+1; j++)
        System.out.print(twoD[i][j] + " ");
    System.out.println();
}
}

```

Эта программа выводит следующий результат:

```

0
1 2
3 4 5
6 7 8 9

```

Созданный в итоге массив показан на рис. 3.2.

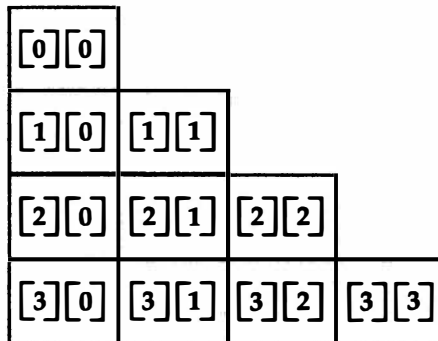


Рис. 3.2. Двухмерный массив с разной размерностью второго измерения

Применение неоднородных (или нерегулярных) массивов может быть неприемлемо во многих приложениях, поскольку их поведение отличается от обычного поведения многомерных массивов. Но в некоторых случаях нерегулярные массивы могут оказаться весьма эффективными. Например, нерегулярный массив может быть идеальным решением, если требуется очень большой двухмерный разреженный массив (т.е. массив, в котором будут использоваться не все элементы).

Многомерные массивы можно инициализировать. Для этого достаточно заключить инициализатор каждого измерения в отдельный ряд фигурных скобок. В приведенном ниже примере программы создается матрица, в которой каждый элемент содержит произведение индексов строки и столбца. Обратите также внимание на то, что в инициализаторах массивов можно применять как литеральное значение, так и выражения:

```

// Инициализировать двухмерный массив
class Matrix {

```

```

public static void main(String args[]) {
    double m[][] = {
        { 0*0, 1*0, 2*0, 3*0 },
        { 0*1, 1*1, 2*1, 3*1 },
        { 0*2, 1*2, 2*2, 3*2 },
        { 0*3, 1*3, 2*3, 3*3 }
    };
    int i, j;
    for(i=0; i<4; i++) {
        for(j=0; j<4; j++)
            System.out.print(m[i][j] + " ");
        System.out.println();
    }
}

```

Эта программа выводит следующий результат:

```

0.0    0.0    0.0    0.0
0.0    1.0    2.0    3.0
0.0    2.0    4.0    6.0
0.0    3.0    6.0    9.0

```

Как видите, каждая строка массива инициализируется в соответствии со значениями, указанными в списках инициализации. Рассмотрим еще один пример применения многомерного массива. В приведенном ниже примере программы сначала создается трехмерный массив размерностью **3×4×5**, а затем каждый элемент массива заполняется произведением его индексов, и, наконец, эти произведения выводятся на экран.

```

// Продемонстрировать применение трехмерного массива
class ThreeDMatrix {
    public static void main(String args[]) {
        int threeD[][][] = new int[3][4][5];
        int i, j, k;
        for(i=0; i<3; i++)
            for(j=0; j<4; j++)
                for(k=0; k<5; k++)
                    threeD[i][j][k] = i * j * k;

        for(i=0; i<3; i++) {
            for(j=0; j<4; j++) {
                for(k=0; k<5; k++)
                    System.out.print(threeD[i][j][k] + " ");
                System.out.println();
            }
        }
    }
}

```

Эта программа выводит следующий результат:

```

0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0

0 0 0 0 0
0 1 2 3 4
0 2 4 6 8

```

```
0 3 6 9 12
0 0 0 0 0
0 2 4 6 8
0 4 8 12 16
0 6 12 18 24
```

Альтернативный синтаксис объявления массивов

Для объявления массивов можно использовать и вторую, приведенную ниже форму.

```
тип[] имя_переменной;
```

В этой форме квадратные скобки следуют за спецификатором типа, а не за именем переменной массива. Например, следующие два объявления равнозначны:

```
int a1[] = new int[3];
int[] a2 = new int[3];
```

И оба приведенных ниже объявления также равнозначны.

```
char twod1[][] = new char[3][4];
char[][] twod2 = new char[3][4];
```

Вторая форма объявления массивов удобна для одновременного объявления нескольких массивов. Например, в объявлении

```
int[] nums, nums2, nums3; // создать три массива
```

создаются три переменные массивов типа `int`. Оно равнозначно приведенному ниже объявлению.

```
int nums[], nums2[], nums3[]; // создать три массива
```

Альтернативная форма объявления массивов удобна также для указания массива в качестве типа данных, возвращаемого методом. В примерах, приведенных в этой книге, используются обе формы объявления массивов.

Введение в символьные строки

Как вы, вероятно, заметили, при рассмотрении типов данных и массивов не упоминались символьные строки или строковый тип данных. Это объясняется не тем, что строковый тип данных не поддерживается в Java, а просто тем, что он не относится к примитивным типам данных. Он не является также массивом символов. Символьная строка скорее является объектом класса `String`, и для ясного представления о ней требуется полное описание целого ряда характеристик объектов. Поэтому строковый тип данных будет рассматриваться в последующих главах книги после описания объектов. Но для того чтобы использовать простые символьные строки в рассматриваемых до этого примерах программ, ниже приводится краткое введение в строковый тип данных.

Тип данных `String` служит для объявления строковых переменных, а также массивов символьных строк. Переменной типа `String` можно присвоить заклю-

ченную в кавычки строковую константу. Переменная типа `String` может быть присвоена другой переменной типа `String`. Объект класса `String` можно указывать в качестве аргумента метода `println()`. Рассмотрим, например, следующий фрагмент кода:

```
String str = "это тестовая строка";  
System.out.println(str);
```

В этом примере `str` обозначает объект класса `String`. Ему присваивается символьная строка "это тестовая строка", которую метод `println()` выводит на экран. Как будет показано далее, объекты класса `String` обладают многими характерными особенностями и свойствами, которые делают их довольно эффективными и простыми в употреблении. Но в примерах программ, представленных в ряде последующих глав, они будут применяться только в своей простейшей форме.

Замечание по поводу указателей для программирующих на C/C++

Тем, у кого имеется опыт программирования на C/C++, известно, что в этих языках поддерживаются указатели. Но в этой главе они не упоминались просто потому, что в Java указатели не только не применяются, но и не поддерживаются. (Точнее говоря, в Java не поддерживаются указатели, которые доступны и/или могут быть видоизменены программистом.) В языке Java не разрешается использовать указатели, поскольку это позволило бы программам на Java преодолевать защитный барьер между исполняющей средой Java и тем компьютером, где она действует. (Напомним, что указателю может быть присвоен любой адрес оперативной памяти — даже те адреса, которые могут находиться за пределами исполняющей системы Java.) В программах на C/C++ указатели используются довольно интенсивно, поэтому их отсутствие может показаться существенным недостатком Java, но на самом деле это не так. Исполняющая среда Java построена таким образом, чтобы для выполнения всех действий в ней указатели не требовались, а их применение в ее пределах не давало никаких преимуществ.

ГЛАВА

4

Операции

В языке Java поддерживается обширный ряд операций. Большинство из них может быть отнесено к одной из следующих четырех групп: арифметические, поразрядные, логические и отношения. В Java также определен ряд дополнительных операций для особых случаев. В этой главе описаны все доступные в Java операции, за исключением операции сравнения типов `instanceof`, рассматриваемой в главе 13, а также новой операции “стрелки” (`->`), описываемой в главе 15.

Арифметические операции

Арифметические операции применяются в математических выражениях таким же образом, как и в алгебре. Все арифметические операции, доступные в Java, перечислены в табл. 4.1

Таблица 4.1. Арифметические операции в Java

Операция	Описание
+	Сложение (а также унарный плюс)
-	Вычитание (а также унарный минус)
*	Умножение
/	Деление
%	Деление по модулю
++	Инкремент (приращение на 1)
+=	Сложение с присваиванием
--	Вычитание с присваиванием
*=	Умножение с присваиванием
/=	Деление с присваиванием
%=	Деление по модулю с присваиванием
--	Декремент (отрицательное приращение на 1)

Операнды арифметических операций должны иметь числовой тип. Арифметические операции нельзя выполнять над логическими типами данных, но допускается над типами данных `char`, поскольку в Java этот тип, по существу, является разновидностью типа `int`.

Основные арифметические операции

Все основные арифметические операции (сложения, вычитания, умножения и деления) воздействуют на числовые типы данных так, как этого и следовало ожидать. Операция унарного вычитания изменяет знак своего единственного операнда. Операция унарного сложения просто возвращает значение своего операнда. Следует, однако, иметь в виду, что когда операция деления выполняется над целочисленным типом данных, ее результат не будет содержать дробный компонент.

В следующем примере простой программы демонстрируется применение арифметических операций. В нем иллюстрируется также отличие в операциях деления с плавающей точкой и целочисленного деления:

```
// Продемонстрировать основные арифметические операции
class BasicMath {
    public static void main(String args[]) {
        // арифметические операции над целочисленными значениями
        System.out.println("Целочисленная арифметика");
        int a = 1 + 1;
        int b = a * 3;
        int c = b / 4;
        int d = c - a;
        int e = -d;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
        System.out.println("d = " + d);
        System.out.println("e = " + e);

        // арифметические операции над значениями типа double
        System.out.println("\nАрифметика с плавающей точкой");
        double da = 1 + 1;
        double db = da * 3;
        double dc = db / 4;
        double dd = dc - a;
        double de = -dd;
        System.out.println("da = " + da);
        System.out.println("db = " + db);
        System.out.println("dc = " + dc);
        System.out.println("dd = " + dd);
        System.out.println("de = " + de);
    }
}
```

При выполнении этой программы на экране появляется следующий результат:

Целочисленная арифметика

```
a = 2
b = 6
c = 1
d = -1
e = 1
```

Арифметика с плавающей точкой

```
da = 2.0
db = 6.0
dc = 1.5
dd = -0.5
de = 0.5
```

Операция деления по модулю

Операция деления по модулю `%` возвращает остаток от деления. Эту операцию можно выполнять как над числовыми типами данных с плавающей точкой, так и над целочисленными типами данных. В следующем примере программы демонстрируется применение операции `%`:

```
// Продемонстрировать применение операции %
class Modulus {
    public static void main(String args[]) {
        int x = 42;
        double y = 42.25;

        System.out.println("x mod 10 = " + x % 10);
        System.out.println("y mod 10 = " + y % 10);
    }
}
```

Эта программа выводит следующий результат:

```
x mod 10 = 2
y mod 10 = 2.25
```

Составные арифметические операции с присваиванием

В Java имеются специальные операции, объединяющие арифметические операции с операцией присваивания. Как вы, вероятно, знаете, операции вроде приведенной ниже встречаются в программах достаточно часто.

```
a = a + 4;
```

В Java эту операцию можно записать следующим образом:

```
a += 4;
```

В этой версии использована *составная операция с присваиванием* `+=`. Обе операции выполняют одно и то же действие: увеличивают значение переменной `a` на **4**. Ниже приведен еще один пример совместного применения арифметической операции и операции присваивания.

```
a = a % 2;
```

Эту строку кода можно переписать следующим образом:

```
a %= 2;
```

В данном случае при выполнении операции `%=` вычисляется остаток от деления `a/2`, а результат размещается обратно в переменной `a`. Составные операции с присваиванием существуют для всех арифметических операций с двумя операндами. Таким образом, любую операцию следующей формы:

переменная = переменная операция выражение;

можно записать так:

переменная операция = выражение;

Составные операции с присваиванием дают два преимущества. Во-первых, позволяют уменьшить объем вводимого кода, поскольку являются “сокращенным” вариантом соответствующих длинных форм. Во-вторых, их реализация в исполняющей системе Java оказывается эффективнее реализации эквивалентных длинных форм. Поэтому в программах, профессионально написанных на Java, составные операции с присваиванием встречаются довольно часто. Ниже приведен еще один пример программы, демонстрирующий практическое применение нескольких составных операций с присваиванием.

```
// Продемонстрировать применени нескольких операций с присваиванием
class OpEquals {
    public static void main(String args[]) {
        int a = 1;
        int b = 2;
        int c = 3;
        a += 5;
        b *= 4;
        c += a * b;
        c %= 6;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("c = " + c);
    }
}
```

Эта программа выводит следующий результат:

```
a = 6
b = 8
c = 3
```

Операции инкремента и декремента

Операции ++ и -- выполняют инкремент и декремент. Эти операции были представлены в главе 2, а в этой главе они будут рассмотрены более подробно. Как станет ясно в дальнейшем, эти операции обладают рядом особенностей, благодаря которым они становятся довольно привлекательными для программирования. Рассмотрим подробнее, что именно выполняют операции инкремента и декремента.

Операция инкремента увеличивает значение операнда на единицу, а операция декремента уменьшает значение операнда на единицу. Например, выражение

```
x = x + 1;
```

с операцией инкремента можно переписать так:

```
x++;
```

Аналогично следующее выражение:

```
x = x - 1;
```

равнозначно приведенному ниже выражению.

```
x--;
```

Эти операции отличаются тем, что могут быть записаны в *постфиксной* форме, когда операция следует за операндом, а также в *префиксной* форме, когда операция

предшествует операнду. В приведенных выше примерах применение любой из этих форм не имеет никакого значения, но когда операции инкремента и декремента являются частью более сложного выражения, проявляется хотя и незначительное, но очень важное отличие этих двух форм. Так, в префиксной форме значение операнда увеличивается или уменьшается до извлечения значения для применения в выражении. А в постфиксной форме предыдущее значение извлекается для применения в выражении, и только после этого изменяется значение операнда. Обратимся к конкретному примеру:

```
x = 42;  
y = ++x;
```

В данном примере переменной `y` присваивается значение **43**, как и следовало ожидать, поскольку увеличение значения переменной `x` выполняется перед его присваиванием переменной `y`. Таким образом, строка кода `y=++x` равнозначна следующим двум строкам кода:

```
x = x + 1;  
y = x;
```

Но если эти операторы переписать следующим образом:

```
x = 42;  
y = x++;
```

то значение переменной `x` извлекается до выполнения операции инкремента, и поэтому переменной `y` присваивается значение **42**. Но в обоих случаях значение переменной `x` устанавливается равным **43**. Следовательно, строка кода `y=x++` равнозначна следующим двум операторам:

```
y = x;  
x = x + 1;
```

В приведенной ниже программе демонстрируется применение операции инкремента.

```
// Продемонстрировать применение операции инкремента ++  
class IncDec {  
    public static void main(String args[]) {  
        int a = 1;  
        int b = 2;  
        int c;  
        int d;  
        c = ++b;  
        d = a++;  
        c++;  
        System.out.println("a = " + a);  
        System.out.println("b = " + b);  
        System.out.println("c = " + c);  
        System.out.println("d = " + d);  
    }  
}
```

Эта программа выводит следующий результат:

```
a = 2  
b = 3  
c = 4  
d = 1
```

Поразрядные операции

В языке Java определяется несколько *поразрядных операций* (табл. 4.2), которые можно выполнять над целочисленными типами данных: long, int, short, char и byte. Эти операции воздействуют на отдельные двоичные разряды операндов.

Таблица 4.2. Поразрядные операции в Java

Операция	Описание
~	Поразрядная унарная операция НЕ
&	Поразрядная логическая операция И
	Поразрядная логическая операция ИЛИ
^	Поразрядная логическая операция исключающее ИЛИ
>>	Сдвиг вправо
>>>	Сдвиг вправо с заполнением нулями
<<	Сдвиг влево
&=	Поразрядная логическая операция И с присваиванием
=	Поразрядная логическая операция ИЛИ с присваиванием
^=	Поразрядная логическая операция исключающее ИЛИ с присваиванием
>>=	Сдвиг вправо с присваиванием
>>>=	Сдвиг вправо с заполнением нулями и присваиванием
<<=	Сдвиг влево с присваиванием

Поразрядные операции манипулируют двоичными разрядами (битами) в целочисленном значении, поэтому очень важно понимать, какое влияние подобные манипуляции могут оказывать на целочисленное значение. В частности, важно иметь в виду, каким образом целочисленные значения хранятся в исполняющей среде Java и как в ней представляются отрицательные числа. Следовательно, прежде чем продолжить рассмотрение поразрядных операций, следует вкратце обсудить два важных вопроса.

Все целочисленные типы данных представлены двоичными числами разной длины. Например, десятичное значение 42 типа byte в двоичном представлении имеет вид 00101010, где позиция каждого двоичного разрядного представляет степень числа два, начиная с 2^0 в крайнем справа разряде. Двоичный разряд на следующей позиции представляет степень числа 2^1 , т.е. 2, следующий — 2^2 , или 4, затем 8, 16, 32 и т.д. Таким образом, двоичное представление числа 42 содержит единичные двоичные разряды на позициях 1, 3 и 5, начиная с 0 на крайней справа позиции. Следовательно, $42 = 2^1 + 2^3 + 2^5 = 2 + 8 + 32$.

Все целочисленные типы данных (за исключением char) представлены со знаком. Это означает, что они могут представлять как положительные, так и отрицательные целочисленные значения. Отрицательные числа в Java представлены в дополнительном коде путем инвертирования (изменения 1 на 0, и наоборот) всех двоичных разрядов исходного значения и последующего добавления 1 к ре-

ультату. Например, число -42 получается путем инвертирования всех двоичных разрядов числа 42 , что дает двоичное значение 11010101 , к которому затем добавляется 1 , а в итоге это дает двоичное значение 11010110 , или -42 в десятичной форме. Чтобы получить положительное число из отрицательного, нужно сначала инвертировать все его двоичные разряды, а затем добавить 1 к результату. Например, инвертирование числа -42 , или 11010110 в двоичной форме, дает двоичное значение 00101001 , или 41 в десятичной форме, а после добавления к нему 1 получается число 42 .

Причина, по которой в Java (и большинстве других языков программирования) применяется дополнительный код, становится понятной при рассмотрении процесса *перехода через ноль*. Если речь идет о значении типа `byte`, то ноль представлен значением 00000000 . Для получения его обратного кода достаточно инвертировать все его двоичные разряды и получить двоичное значение 11111111 , которое представляет отрицательный ноль. Но дело в том, что отрицательный ноль недопустим в целочисленной математике. Выходом из этого затруднения служит дополнительный код для представления отрицательных чисел. В этом случае к обратному коду нулевого значения добавляется 1 и получается двоичное значение 10000000 . Старший единичный разряд оказывается сдвинутым влево слишком далеко, чтобы уместиться в значении типа `byte`. Тем самым достигается требуемое поведение, когда значения -0 и 0 равнозначны, а 11111111 — двоичный код значения -1 . В данном примере использовано значение типа `byte`, но тот же самый принцип можно применить ко всем целочисленным типам данных в Java.

Для хранения отрицательных значений в Java используется дополнительный код, а все целочисленные значения представлены со знаком, поэтому выполнение поразрядных операций может легко привести к неожиданным результатам. Например, установка 1 в самом старшем двоичном разряде может привести к тому, что получающееся в итоге значение будет интерпретироваться как отрицательное число, независимо от того, какого именно результата предполагалось добиться. Во избежание неприятных сюрпризов не следует забывать, что старший двоичный разряд определяет знак целого числа независимо от того, как он был установлен.

Поразрядные логические операции

Поразрядными логическими являются операции $\&$, $|$, \wedge и \sim . Результаты выполнения каждой из этих операций приведены в табл. 4.3. Изучая последующий материал книги, не следует забывать, что поразрядные логические операции выполняются отдельно над каждым двоичным разрядом каждого операнда.

Таблица 4.3. Результаты выполнения поразрядных логических операций

A	B	A B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

Поразрядная унарная операция НЕ

Эта операция обозначается знаком \sim и называется также *поразрядным отрицанием*, инвертируя все двоичные разряды своего операнда. Например, число 42, представленное в следующей двоичной форме:

```
00101010
```

преобразуется в результате выполнения поразрядной унарной операции НЕ в следующую форму:

```
11010101
```

Поразрядная логическая операция И

При выполнении поразрядной логической операции И, обозначаемой знаком $\&$, в двоичном разряде результата устанавливается 1 лишь в том случае, если соответствующие двоичные разряды в операндах также равны 1. Во всех остальных случаях в двоичном разряде результата устанавливается 0, как показано ниже.

```

00101010    42
& 00001111    15
-----
00001010    10

```

Поразрядная логическая операция ИЛИ

При выполнении поразрядной логической операции ИЛИ, обозначаемой знаком $|$, в двоичном разряде результата устанавливается 1, если соответствующий двоичный разряд в любом из операндов равен 1, как показано ниже.

```

00101010    42
| 00001111    15
-----
00101111    47

```

Поразрядная логическая операция исключающее ИЛИ

При выполнении поразрядной логической операции исключающее ИЛИ, обозначаемой знаком \wedge , в двоичном разряде результата устанавливается 1, если двоичный разряд только в одном из операндов равен 1, а иначе в двоичном разряде результата устанавливается 0, как показано ниже. Приведенный ниже пример демонстрирует также полезную особенность поразрядной логической исключающее операции ИЛИ. Обратите внимание на инвертирование последовательности двоичных разрядов числа 42 во всех случаях, когда в двоичном разряде второго операнда установлена 1. А во всех случаях, когда в двоичном разряде второго операнда установлен 0, двоичный разряд первого операнда остается без изменения. Этим свойством удобно пользоваться при манипулировании отдельными битами числовых значений.

```

00101010    42
^ 00001111    15
-----
00100101    37

```

Применение поразрядных логических операций

В следующем примере программы демонстрируется применение поразрядных логических операций:

```
// Продемонстрировать применение поразрядных логических операций
class BitLogic {
    public static void main(String args[]) {
        String binary[] = {
            "0000", "0001", "0010", "0011", "0100", "0101", "0110", "0111",
            "1000", "1001", "1010", "1011", "1100", "1101", "1110", "1111"
        };
        int a = 3; // 0 + 2 + 1, или 0011 в двоичном представлении
        int b = 6; // 4 + 2 + 0, или 0110 в двоичном представлении
        int c = a | b;
        int d = a & b;
        int e = a ^ b;
        int f = (~a & b) | (a & ~b);
        int g = ~a & 0x0f;
        System.out.println(" a = " + binary[a]);
        System.out.println(" b = " + binary[b]);
        System.out.println(" a|b = " + binary[c]);
        System.out.println(" a&b = " + binary[d]);
        System.out.println(" a^b = " + binary[e]);
        System.out.println("~a&b|a&~b = " + binary[f]);
        System.out.println(" ~a = " + binary[g]);
    }
}
```

В данном примере последовательности двоичных разрядов в переменных *a* и *b* представляют все четыре возможные комбинации двух двоичных цифр: 0-0, 0-1, 1-0 и 1-1. О воздействии операций *|* и *&* на каждый двоичный разряд можно судить по результирующим значениям переменных *c* и *d*. Значения, присвоенные переменным *e* и *f*, иллюстрируют действие операции *^*. Массив символьных строк *binary* содержит удобочитаемые двоичные представления чисел от 0 до 15. Этот массив индексирован, что позволяет увидеть двоичное представление каждого результирующего значения. Он построен таким образом, чтобы соответствующее строковое представление двоичного значения *n* хранилось в элементе массива *binary[n]*. Значение *~a* уменьшается до величины меньше 16 в результате поразрядной логической операции И со значением 0x0f (0000 1111 в двоичном представлении), чтобы вывести его в двоичном представлении из массива *binary*. Эта программа выводит следующий результат:

```
a = 0011
b = 0110
a|b = 0111
a&b = 0010
a^b = 0101
~a&b|a&~b = 0101
a = 1100
```

Сдвиг влево

Операция сдвига влево, обозначаемая знаками *<<*, смещает все двоичные разряды значения влево на указанное количество позиций. Эта операция имеет следующую общую форму:

значение << количество

где *количество* обозначает число позиций, на которое следует сдвинуть влево двоичные разряды в заданном значении. Это означает, что операция *<<* смещает

влево двоичные разряды заданного значения на количество позиций, указанных в операнде *количество*. При каждом сдвиге влево самый старший двоичный разряд смещается за пределы допустимого диапазона значений (и при этом теряется), а справа добавляется ноль. Это означает, что при выполнении операции сдвига влево двоичные разряды в операнде типа `int` теряются, как только они сдвигаются за пределы 31-й позиции. Если же операнд относится к типу `long`, то двоичные разряды теряются после сдвига за пределы 63-й позиции.

Автоматическое продвижение типа в Java приводит к непредвиденным результатам при сдвиге влево двоичных разрядов в значениях типа `byte` и `short`. Как известно, типы `byte` и `short` продвигаются к типу `int` при вычислении выражения. Более того, результат вычисления такого выражения также имеет тип `int`. Это означает, что в результате сдвига влево двоичных разрядов значения типа `byte` или `short` получится значение типа `int`, и сдвинутые влево двоичные разряды не будут отброшены до тех пор, пока они не сдвинутся за пределы 31-й позиции. Более того, при продвижении к типу `int` отрицательное значение типа `byte` или `short` приобретает дополнительный знаковый разряд. Следовательно, старшие двоичные разряды заполняются единицами. Поэтому выполнение операции сдвига влево по отношению к значению типа `byte` или `short` подразумевает отбрасывание старших байтов результата типа `int`. Например, при сдвиге влево двоичных разрядов значения типа `byte` сначала происходит продвижение к типу `int` и только затем сдвиг. Это означает, что для получения требуемого сдвинутого влево значения типа `byte` придется отбросить три старших байта результата. Простейший способ добиться этого — привести результат обратно к типу `byte`. Такой способ демонстрируется в следующем примере программы:

```
// Сдвиг влево значения типа byte
class ByteShift {
    public static void main(String args[]) {
        byte a = 64, b;
        int i;

        i = a << 2;
        b = (byte) (a << 2);

        System.out.println("Первоначальное значение a: " + a);
        System.out.println("i and b: " + i + " " + b);
    }
}
```

Эта программа выводит следующий результат:

```
Первоначальное значение a: 64
i and b: 256 0
```

Для целей вычисления тип переменной `a` продвигается к типу `int`, поэтому сдвиг значения **64** (**0100 0000**) влево на две позиции приводит к значению **256** (**1 0000 0000**), присваиваемому переменной `i`. Но переменная `b` содержит нулевое значение, поскольку после сдвига в младшем двоичном разряде устанавливается 0. Единственный единичный двоичный разряд оказывается сдвинутым за пределы допустимого диапазона значений.

Каждый сдвиг влево на одну позицию, по существу, удваивает исходное значение, поэтому программисты нередко пользуются такой возможностью в качестве

эффективной альтернативы умножению на 2. Но при этом следует соблюдать осторожность. Ведь при сдвиге единичного двоичного разряда на старшую (31-ю или 63-ю) позицию значение становится отрицательным. Такое применение операции сдвига влево демонстрируется в следующем примере программы:

```
// Применение сдвига влево в качестве быстрого способа умножения на 2
class MultByTwo {
    public static void main(String args[]) {
        int i;
        int num = 0xFFFFFFFF;

        for(i=0; i<4; i++) {
            num = num << 1;
            System.out.println(num);
        }
    }
}
```

Эта программа выводит следующий результат:

```
536870908
1073741816
2147483632
-32
```

Начальное значение было специально выбрано таким, чтобы после сдвига влево на четыре позиции оно стало равным **-32**. Как видите, после сдвига единичного двоичного разряда на 31-ю позицию числовое значение интерпретируется как отрицательное.

Сдвиг вправо

Операция сдвига вправо, обозначаемая знаками **>>**, смещает все двоичные разряды заданного значения вправо на указанное количество позиций. В общем виде эта операция обозначается следующим образом:

значение >> количество

где *количество* обозначает число позиций, на которое следует сдвинуть вправо двоичные разряды в заданном значении. То есть операция **>>** смещает все двоичные разряды в заданном значении вправо на число позиций, указанное в операнде *количество*. В следующем фрагменте кода значение **32** сдвигается вправо на две позиции. В итоге переменной *a* присваивается значение **8**:

```
int a = 32;
a = a >> 2; // теперь переменная a содержит значение 8
```

Когда какие-нибудь двоичные разряды значения сдвигаются за его пределы, они теряются. Например, в приведенном ниже фрагменте кода значение **35** сдвигается на две позиции вправо. В итоге теряются два младших двоичных разряда и переменной *a* снова присваивается значение **8**.

```
int a = 35;
a = a >> 2; // переменная a содержит значение 8
```

Чтобы лучше понять, как выполняется операция сдвига вправо, ниже показано, каким образом сдвиг вправо происходит в двоичном представлении.


```

00100011    35
>> 2
-----
00001000     8

```

При каждом сдвиге вправо выполняется деление заданного значения на два с отбрасыванием любого остатка. Этой особенностью данной операции можно воспользоваться для эффективного целочисленного деления на 2. Но при этом следует проявлять осторожность, чтобы не потерять двоичные разряды, безвозвратно сдвинутые за пределы правой границы числового значения.

При выполнении операции сдвига вправо старшие двоичные разряды на крайних слева позициях освобождаются и заполняются предыдущим содержимым старшего двоичного разряда. В итоге происходит так называемое *расширение знака*, которое служит для сохранения знака отрицательных чисел при их сдвиге вправо. Например, результат выполнения операции `-8 >> 1` равен `-4`, что в двоичном представлении выглядит следующим образом:

```

11111000    -8
>> 1
-----
11111100    -4

```

Любопытно, что результат сдвига значения `-1` вправо всегда равен `-1`, поскольку расширение знака приводит к переносу дополнительных единиц в старшие двоичные разряды. Иногда при выполнении сдвига вправо расширение знака числовых значений нежелательно. Так, в приведенном ниже примере программы значение типа `byte` преобразуется в соответствующее шестнадцатеричное строковое представление. Обратите внимание на то, что для индексации массива символов, обозначающих шестнадцатеричные цифры, сдвинутое вправо исходное значение маскируется значением `0x0f` в поразрядной логической операции И, что приводит к отбрасыванию любых двоичных разрядов расширения знака.

```

// Маскирование двоичных разрядов расширения знака
class HexByte {
    static public void main(String args[]) {
        char hex[] = {
            '0', '1', '2', '3', '4', '5', '6', '7',
            '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'
        };

        byte b = (byte) 0xf1;

        System.out.println("b = 0x" + hex[(b >> 4) & 0x0f]
            + hex[b & 0x0f]);
    }
}

```

Эта программа выводит следующий результат:

```
b = 0xf1
```

Беззнаковый сдвиг вправо

Как пояснялось выше, при каждом выполнении операции `>>` старший двоичный разряд автоматически заполняется своим предыдущим содержимым. В итоге

сохраняется знак сдвигаемого значения. Но иногда это нежелательно. Например, при сдвиге вправо двоичных разрядов какого-нибудь нечислового значения расширение знака может быть нежелательным. Подобная ситуация нередко возникает при обработке значений отдельных пикселей в графических изображениях. Как правило, в подобных случаях требуется сдвиг нуля на позицию старшего двоичного разряда независимо от его первоначального значения. Это так называемый *беззнаковый сдвиг*. Для этой цели в Java имеется операция беззнакового сдвига вправо. Она обозначается тремя знаками `>>>` и всегда вставляет нуль на позиции старшего двоичного разряда.

В приведенном ниже примере кода демонстрируется применение операции `>>>`. Сначала в данном примере переменной `a` присваивается значение `-1`, где во всех 32 битах двоичного представления устанавливается 1. Затем в этом значении выполняется сдвиг вправо на 24 бита, в ходе которого 24 старших двоичных разряда заполняются нулями и игнорируется обычное расширение знака. Таким образом, в переменной `a` устанавливается значение 255.

```
int a = -1;
a = a >>> 24;
```

Чтобы операция беззнакового сдвига вправо стала понятнее, она представлена ниже в двоичной форме.

```
11111111 11111111 11111111 11111111  значение -1 типа int в двоичном виде
>>> 24
-----
00000000 00000000 00000000 11111111  значение 255 типа int в двоичном виде
```

Нередко операция `>>>` оказывается не такой полезной, как хотелось бы, поскольку ее выполнение имеет смысл только для 32- и 64-разрядных значений. Не следует забывать, что в выражениях тип меньших значений автоматически продвигается к типу `int`. Это означает, что расширение знака и сдвиг происходят в 32-разрядных, а не 8- или 16-разрядных значениях. Так, в результате беззнакового сдвига вправо двоичных разрядов значения типа `byte` можно было бы ожидать заполнения нулями, начиная с 7-й позиции, но на самом деле этого не происходит, поскольку сдвиг вправо фактически выполняется в 32-разрядном значении. Именно это и демонстрируется в приведенном ниже примере программы.

```
// Беззнаковый сдвиг двоичных разрядов значения типа byte
class ByteUShift {
    static public void main(String args[]) {
        char hex[] = {
            '0', '1', '2', '3', '4', '5', '6', '7',
            '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'
        };
        byte b = (byte) 0xf1;
        byte c = (byte) (b >> 4);
        byte d = (byte) (b >>> 4);
        byte e = (byte) ((b & 0xff) >> 4);

        System.out.println(" b = 0x"
            + hex[(b >> 4) & 0x0f] + hex[b & 0x0f]);
        System.out.println(" b >> 4 = 0x"
            + hex[(c >> 4) & 0x0f] + hex[c & 0x0f]);
        System.out.println(" b >>> 4 = 0x"
```

```

        + hex[(d >> 4) & 0x0f] + hex[d & 0x0f]);
    System.out.println("(b & 0xff) >> 4 = 0x"
        + hex[(e >> 4) & 0x0f] + hex[e & 0x0f]);
    }
}

```

Как следует из приведенного ниже результата выполнения данной программы, операция `>>>` не оказывает никакого воздействия на значения типа `byte`. Сначала в данном примере программы переменной `b` присваивается произвольное отрицательное значение типа `byte`. Затем переменной `c` присваивается значение переменной `b` типа `byte`, сдвинутое на четыре позиции вправо и равное `0xff` вследствие расширения знака. Далее переменной `d` присваивается значение переменной `b` типа `byte`, сдвинутое на четыре позиции вправо без знака. Это значение должно было бы быть равно `0x0f`, но в действительности она оказывается равным `0xff` из-за расширения знака, которое произошло при продвижении типа переменной `b` к типу `int` перед сдвигом. И в последнем выражении переменной `e` присваивается значение переменной `b` типа `byte`, сначала замаскированное до 8 двоичных разрядов с помощью поразрядной логической операции И, а затем сдвинутое вправо на четыре позиции. В итоге получается предполагаемое значение `0x0f`. Обратите внимание на то, что операция беззнакового сдвига вправо не применяется к значению переменной `d`, поскольку состояние знакового двоичного разряда известно после выполнения поразрядной логической операции И.

```

        b = 0xf1
        b >> 4 = 0xff
        b >>> 4 = 0xff
(b & 0xff) >> 4 = 0x0f

```

Поразрядные составные операции с присваиванием

Подобно арифметическим операциям, все двоичные поразрядные операции имеют составную форму, в которой поразрядная операция объединяется с операцией присваивания. Например, две приведенные ниже операции, выполняющие сдвиг на четыре позиции вправо двоичных разрядов значения переменной `a`, равнозначны.

```

a = a >> 4;
a >>= 4;

```

Равнозначны и следующие две операции, присваивающие переменной `a` результат выполнения поразрядной логической операции а ИЛИ b:

```

a = a | b;
a |= b;

```

Следующая программа создает несколько целочисленных переменных, а затем использует составные побитовые операторы с присваиванием для манипулирования этими переменными:

```

class OpBitEquals {
    public static void main(String args[]) {
        int a = 1;
        int b = 2;
        int c = 3;
    }
}

```

```
a |= 4;
b >>= 1;
c <<= 1;
a ^= c;

System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
}
}
```

Эта программа выводит следующий результат:

```
a = 3
b = 1
c = 6
```

Операции отношения

Операции отношения, называемые иначе *операциями сравнения*, определяют отношение одного операнда к другому. В частности, они определяют равенство и упорядочение. Все доступные в Java операции отношения перечислены в табл. 4.4.

Таблица 4.4. Операции отношения в Java

Операция	Описание
<code>==</code>	Равно
<code>!=</code>	Не равно
<code>></code>	Больше
<code><</code>	Меньше
<code>>=</code>	Больше или равно
<code><=</code>	Меньше или равно

Результатом выполнения этих операций оказывается логическое значение. Чаще всего операции отношения применяются в выражениях, управляющих условным оператором `if` и различными операторами цикла.

В Java допускается сравнивать значения любых типов, в том числе целочисленные значения, числовые значения с плавающей точкой, символы и логические значения, выполняя проверку на равенство `==` и неравенство `!=`. (Напомним, что в Java равенство обозначается двумя знаками `==`, тогда как операция присваивания — одним знаком `=`.) Сравнение с помощью операций упорядочения допускается только для числовых типов данных. Таким образом, сравнение, позволяющее определить, какой из двух операндов больше или меньше, можно выполнить только над целочисленными, символьными операндами или числовыми операндами с плавающей точкой.

Как отмечалось выше, результатом операции отношения является логическое значение. Например, приведенный ниже фрагмент кода вполне допустим. В данном примере логическое значение `false`, получающееся в результате выполнения операции `a < b`, сохраняется в переменной `c`.

```
int a = 4;
int b = 1;
boolean c = a < b;
```

Тем, у кого имеется опыт программирования на C/C++, следует обратить внимание на то, что в программах на C/C++ очень часто встречаются приведенные ниже типы операторов.

```
int done;
// ...
if(!done) ... // Допустимо в C/C++,
if(done) ... // но не в Java.
```

А в программе на Java эти операторы должны быть написаны следующим образом:

```
if(done == 0) ... // Это стиль Java
if(done != 0) ...
```

Дело в том, что в Java определение “истинного” и “ложного” значений отличается от их определения в C/C++. В языках C/C++ истинным считается любое ненулевое значение, а ложным — нулевое. Но в Java логические значения `true` (истинно) и `false` (ложно) являются нечисловыми и никак не связаны с нулевым или ненулевым значением. Поэтому в выражениях приходится явно указывать одну или несколько операций отношения, чтобы проверить значение на равенство или неравенство нулю.

Логические операции

Описываемые в этом разделе логические операции выполняются только с операндами типа `boolean`. Все логические операции с двумя операндами соединяют два логических значения, образуя результирующее логическое значение. Все доступные в Java логические операции перечислены в табл. 4.5.

Таблица 4.5. Логические операции в Java

Операция	Описание
<code>&</code>	Логическая операция И
<code> </code>	Логическая операция ИЛИ
<code>^</code>	Логическая операция исключающее ИЛИ
<code> </code>	Укороченная логическая операция ИЛИ
<code>&&</code>	Укороченная логическая операция И
<code>!</code>	Логическая унарная операция НЕ
<code>&=</code>	Логическая операция И с присваиванием
<code> =</code>	Логическая операция ИЛИ с присваиванием
<code>^=</code>	Логическая операция исключающее ИЛИ с присваиванием
<code>==</code>	Равенство
<code>!=</code>	Неравенство
<code>?:</code>	Тернарная условная операция типа <i>если ..., то ..., иначе ...</i>

Логические операции `&`, `|` и `^` воздействуют на значения типа `boolean` точно так же, как и на отдельные двоичные разряды целочисленных значений. Так, логическая операция `!` инвертирует логическое состояние: `!true == false` и `!false == true`. В табл. 4.6 приведены результаты выполнения каждой из логических операций.

Таблица 4.6. Результаты выполнения логических операций

A	B	A B	A & B	A ^ B	!A
false	false	false	false	false	true
true	false	true	false	true	false
false	true	true	false	true	true
true	true	true	true	false	false

Ниже приведен пример программы, в которой выполняются практически те же самые действия, что и в представленном выше примере программы `BitLogic`. Но эта программа оперирует логическими значениями типа `boolean`, а не двоичными разрядами.

```
// Продемонстрировать применение логических операций
class BoolLogic {
    public static void main(String args[]) {
        boolean a = true;
        boolean b = false;
        boolean c = a | b;
        boolean d = a & b;
        boolean e = a ^ b;
        boolean f = (!a & b) | (a & !b);
        boolean g = !a;

        System.out.println("      a = " + a);
        System.out.println("      b = " + b);
        System.out.println("    a|b = " + c);
        System.out.println("    a&b = " + d);
        System.out.println("    a^b = " + e);
        System.out.println(" !a&b|a&!b = " + f);
        System.out.println("      !a = " + g);
    }
}
```

Выполняя эту программу, легко убедиться, что к значениям типа `boolean` применяются те же самые логические правила, что и к двоичным разрядам. Как следует из приведенного ниже результата выполнения данной программы, в Java строковым представлением значения типа `boolean` служит значение одного из литералов `true` или `false`.

```
a = true
b = false
a|b = true
a&b = false
a^b = true
!a&b|a&!b = true
!a = false
```

Укороченные логические операции

В языке Java предоставляются две любопытные логические операции, отсутствующие во многих других языках программирования. Эти вторые версии логических операций И и ИЛИ обычно называются *укороченными* логическими операциями. Как следует из табл. 4.6, результат выполнения укороченной логической операции ИЛИ равен true, когда значение операнда A равно true независимо от значения операнда B. Аналогично результат выполнения укороченной логической операции И равен false, когда значение операнда A равно false независимо от значения операнда B. При использовании форм `||` и `&&` этих операторов вместо форм `|` и `&` в программе на Java не будет вычисляться значение правого операнда, если результат выражения можно определить по значению только левого операнда. Этим обстоятельством очень удобно пользоваться в тех случаях, когда значение правого операнда зависит от значения левого. Например, в следующей строке кода демонстрируется преимущество применения укороченных логических операций, чтобы выяснить достоверность операции деления перед вычислением ее результата.

```
if (denom != 0 && num / denom > 10)
```

Благодаря применению укороченной формы логической операции И (`&&`) отсутствует риск возникновения исключений во время выполнения в связи с равенством нулю знаменателя (denom). Если бы в данной строке кода был указан одинарный знак `&` логической операции И, то вычислялись бы обе части выражения, что привело бы к исключению во время выполнения в связи с тем, что значение переменной denom равно нулю.

Укороченные формы логических операций И и ИЛИ принято применять в тех случаях, когда требуются логические операции, а их полные формы — исключительно для выполнения поразрядных операций. Но из этого правила есть исключения. Рассмотрим в качестве примера приведенную ниже строку кода. В данном примере одиночный знак `&` гарантирует выполнение операции инкремента над значением переменной e, независимо от того, равно ли 1 значение переменной c.

```
if (c==1 & e++ < 100) d = 100;
```

На заметку! В формальной спецификации языка Java укороченные логические операции называются *условными*, например, условная логическая операция И или условная логическая операция ИЛИ.

Операция присваивания

Операция присваивания применялась в примерах программ с главы 2. Теперь настало время рассмотреть ее подробно. *Операция присваивания* обозначается одиночным знаком равенства `=`. В Java операция присваивания действует таким же образом, как и во многих других языках программирования. Она имеет следующую общую форму:

переменная = *выражение*;

В этой форме *переменная* и *выражение* должны иметь совместимый тип. Операция присваивания имеет одну интересную особенность, с которой вы, возможно, еще не знакомы: она позволяет объединять присваивания в цепочки. Рассмотрим в качестве примера следующий фрагмент кода:

```
int x, y, z;  
x = y = z = 100; // установить значение 100 в переменных x, y и z
```

В этом фрагменте кода единственная операция присваивания позволяет установить значение 100 сразу в трех переменных: x, y и z. Это происходит благодаря тому, что в операции = используется значение правого выражения. Таким образом, в результате вычисления выражения $z = 100$ получается значение 100, которое присваивается сначала переменной y, а затем — переменной x. Применение “цепочки присваивания” — удобный способ установки общего значения в группе переменных.

Тернарная операция ?

В синтаксисе Java имеется специальная *тернарная операция*, которая обозначается знаком ? и которой можно заменить определенные типы условных операторов вроде *если..., то..., иначе...* (if-then-else). На первый взгляд эта операция может показаться вам не совсем непонятной, но со временем вы убедитесь в ее особой эффективности. Эта операция имеет следующую общую форму:

выражение1 ? выражение2 : выражение3

где *выражение1* обозначает любое выражение, вычисление которого дает логическое значение типа boolean. Если это логическое значение true, то вычисляется *выражение2*, в противном случае — *выражение3*. Результат выполнения тернарной операции ? равен значению вычисленного выражения. Из обеих ветвей *выражение2* и *выражение3* тернарной операции ? должно возвращаться значение одинакового (или совместимого) типа, которым не может быть тип void. Ниже приведен пример применения тернарной операции ?.

```
ratio = denom == 0 ? 0 : num / denom;
```

Когда вычисляется данное выражение присваивания, сначала проверяется выражение слева от знака вопроса. Если значение переменной denom равно 0, то вычисляется выражение, указанное между знаком вопроса и двоеточием, а вычисленное значение используется в качестве значения всего тернарного выражения ?. Если же значение переменной denom не равно 0, то вычисляется выражение, указанное после двоеточия, и оно используется в качестве значения всего тернарного выражения ?. И наконец, значение, полученное в результате выполнения тернарной операции ?, присваивается переменной ratio.

В приведенном ниже примере программы демонстрируется применение тернарной операции ?. Эта программа служит для получения абсолютного значения переменной.

```
// Продемонстрировать применение тернарной операции ?  
class Ternary {  
    public static void main(String args[]) {
```



```

int i, k;

i = 10;
k = i < 0 ? -i : i; // получить абсолютное значения переменной i
System.out.print("Абсолютное значение ");
System.out.println(i + " равно " + k);

i = -10;
k = i < 0 ? -i : i; // получить абсолютное значения переменной i
System.out.print("Абсолютное значение ");

System.out.print("Абсолютное значение ");
System.out.println(i + " равно " + k);
}
}

```

Эта программа выводит следующий результат:

```

Абсолютное значение 10 равно 10
Абсолютное значение -10 равно 10

```

Предшествование операций

Предшествование (от высшего к низшему) операций в Java приведено в табл. 4.7. Операции, находящиеся в одном ряду таблицы, имеют одинаковое предшествование. Операции с двумя операндами имеют порядок вычисления слева направо (за исключением операции присваивания, которая выполняется справа налево). Формально скобки `[]` и `()`, а также точка `.` считаются разделителями, но они могут служить и в качестве операций. Именно в этом качестве они имеют наивысшее предшествование. Обратите также внимание на операцию “стрелка” `(->)`, которая была внедрена в версии JDK 8 и применяется в лямбда-выражениях.

Таблица 4.7. Предшествование операций в Java

Наивысшее предшествование						
++ (постфиксная операция)	-- (постфиксная операция)					
++ (префиксная операция)	-- (префиксная операция)	~	!	+(унарная операция)	-(унарная операция)	(приведение типов)
*	/	%				
+	-					
>>	>>>	<<				
>	>=	<	<=	instanceof		
=	!=					
&						
^						

Окончание табл. 4,7

**Наивысшее
предшествование**

|

&&

||

?:

->

= операция=

**Наинизшее
предшествование**

Применение круглых скобок

Круглые скобки повышают предшествование заключенных в них операций. Нередко это требуется для получения нужного результата. Рассмотрим в качестве примера следующее выражение:

```
a >> b + 3
```

Сначала в этом выражении к значению переменной *b* добавляется значение 3, а затем двоичные разряды значения переменной *a* сдвигаются вправо на полученное в итоге количество позиций. Используя избыточные круглые скобки, это выражение можно было бы записать следующим образом:

```
a >> (b + 3)
```

Но если требуется сначала сдвинуть двоичные разряды значения переменной *a* вправо на *b* позиций, а затем добавить 3 к полученному результату, то круглые скобки следует использовать так, как показано ниже.

```
(a >> b) + 3
```

Кроме изменения обычного предшествования операций, круглые скобки можно иногда использовать с целью упростить понимание смысла выражения. Сложные выражения могут оказаться трудными для понимания. Добавление избыточных, но облегчающих понимание круглых скобок может способствовать устранению недоразумений впоследствии. Например, какое из приведенных ниже выражений легче прочесть?

```
a | 4 + c >> b & 7
```

```
(a | ((4 + c) >> b) & 7)
```

И наконец, следует иметь в виду, что применение круглых скобок (избыточных или не избыточных) не ведет к снижению производительности программы. Следовательно, добавление круглых скобок для повышения удобочитаемости исходного текста программы не оказывает никакого влияния на эффективность ее работы.

ГЛАВА

5

Управляющие операторы

В языках программирования управляющие операторы применяются для реализации переходов и ветвлений в потоке исполнения команд программы, исходя из ее состояния. Управляющие операторы в программе на Java можно разделить на следующие категории: операторы выбора, операторы цикла и операторы перехода. Операторы выбора позволяют выбирать разные ветви выполнения команд в соответствии с результатом вычисления заданного выражения или состоянием переменной. Операторы *цикла* позволяют повторять выполнение одного или нескольких операторов (т.е. они образуют циклы). Операторы *перехода* обеспечивают возможность нелинейного выполнения программы. В этой главе будут рассмотрены все управляющие операторы, доступные в Java.

Операторы выбора

В языке Java поддерживаются два оператора выбора: `if` и `switch`. Эти операторы позволяют управлять порядком выполнения команд программы в соответствии с условиями, которые известны только во время выполнения. Читатели будут приятно удивлены возможностями и гибкостью этих двух операторов.

Условный оператор `if`

В этой главе подробно рассматривается условный оператор `if`, вкратце представленный в главе 2. Это оператор условного ветвления программы на Java. Его можно использовать с целью направить выполнение программы по двум разным ветвям. Общая форма этого условного оператора выглядит следующим образом:

```
if (условие) оператор1;  
else оператор2;
```

где каждый *оператор* обозначает одиночный или составной оператор, заключенный в фигурные скобки (т.е. *блок кода*); *условие* — любое выражение, возвращающее логическое значение типа `boolean`. А оператор `else` указывать необязательно.

Условный оператор `if` действует следующим образом: если *условие* истинно, то выполняет *оператор1*, а иначе — *оператор2*, если таковой имеется. Но ни в коем случае не будут выполняться оба оператора. Рассмотрим в качестве примера следующий фрагмент кода:

```
int a, b;
// ...
if(a < b) a = 0;
else b = 0;
```

Если в данном примере значение переменной *a* меньше значения переменной *b*, то нулевое значение устанавливается в переменной *a*, в противном случае — в переменной *b*. Но ни в коем случае нулевое значение не может быть установлено сразу в обеих переменных, *a* и *b*.

Чаще всего в управляющих выражениях оператора *if* применяются операции отношения, хотя это и не обязательно. Для управления условным оператором *if* можно применять и одиночную переменную типа *boolean*, как показано в следующем фрагменте кода:

```
boolean dataAvailable;
// ...
if (dataAvailable)
    processData();
else
    waitForMoreData();
```

Следует иметь в виду, что только один оператор может следовать непосредственно за ключевым словом *if* или *else*. Если же нужно ввести больше операторов, то придется написать код, аналогичный приведенному ниже. В данном примере оба оператора, размещенные в блоке кода, будут выполняться в том случае, если значение переменной *bytesAvailable* окажется больше нуля.

```
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
    processData();
    bytesAvailable -= n;
} else
    waitForMoreData();
```

Некоторые программисты предпочитают использовать в условном операторе *if* фигурные скобки даже при наличии только одного оператора в каждом выражении. Это упрощает добавление операторов в дальнейшем и избавляет от необходимости проверять наличие фигурных скобок. На самом деле пропуск определения блока в тех случаях, когда он действительно требуется, относится к числу довольно распространенных ошибок. Рассмотрим в качестве примера следующий фрагмент кода:

```
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
    processData();
    bytesAvailable -= n;
} else
    waitForMoreData();
    bytesAvailable = n;
```

Если судить по величине отступа, то в данном примере кода изначально предполагалось, что оператор *bytesAvailable=n*; должен выполняться в ветви оператора *else*. Но не следует забывать, что в Java отступы не имеют никакого значения,

а компилятору никоим образом не известны намерения программиста. Данный код будет скомпилирован без вывода каких-нибудь предупреждающих сообщений, но во время выполнения он будет вести себя не так, как предполагалось. В приведенном ниже фрагменте кода исправлена ошибка, допущенная в предыдущем примере.

```
int bytesAvailable;
// ...
if (bytesAvailable > 0) {
    ProcessData();
    bytesAvailable -= n;
} else {
    waitForMoreData();
    bytesAvailable = n;
}
```

Вложенные условные операторы **if**

Вложенным называется такой условный оператор **if**, который является целью другого условного оператора **if** или **else**. В программах вложенные условные операторы **if** встречаются очень часто. Пользуясь вложенными условными операторами **if**, не следует забывать, что оператор **else** всегда связан с ближайшим условным оператором **if**, находящимся в том же самом блоке кода и еще не связанным с другим оператором **else**.

```
if(i == 10) {
    if(j < 20) a = b;
    if(k > 100) c = d; // этот условный оператор if
    else a = c;        // связан с данным оператором else,
}
else a = d;           // а этот оператор else — с оператором if(i == 10)
```

Как следует из комментариев к данному фрагменту кода, внешний оператор **else** не связан с оператором **if(j < 20)**, поскольку тот не находится в том же самом блоке кода, несмотря на то, что он является ближайшим условным оператором **if**, который еще не связан с оператором **else**. Следовательно, внешний оператор **else** связан с оператором **if(i==10)**. А внутренний оператор **else** связан с оператором **if(k > 100)**, поскольку тот является ближайшим к нему в том же самом блоке кода.

Конструкция **if-else-if**

Конструкция **if-else-if**, состоящая из последовательности вложенных условных операторов **if**, весьма распространена в программировании. В общем виде она выглядит следующим образом:

```
if(условие)
    оператор;
else if(условие)
    оператор;
else if(условие)
    оператор;
...
...
...
```

```
else
    оператор;
```

Условные операторы `if` выполняются последовательно, сверху вниз. Как только одно из условий, управляющих оператором `if`, оказывается равным `true`, выполняется оператор, связанный с данным условным оператором `if`, а остальная часть конструкции `if-else-if` пропускается. Если ни одно из условий не выполняется (т.е. не равно `true`), то выполняется заключительный оператор `else`. Этот последний оператор служит условием по умолчанию. Иными словами, если проверка всех остальных условий дает отрицательный результат, выполняется последний оператор `else`. Если же заключительный оператор `else` не указан, а результат проверки всех остальных условий равен `false`, то не выполняется никаких действий. Ниже приведен пример программы, в которой конструкция `if-else-if` служит для определения времени года, к которому относится конкретный месяц.

```
// Продемонстрировать применение конструкции if-else-if
class IfElse {
    public static void main(String args[]) {
        int month = 4; // Апрель
        String season;
        if(month == 12 || month == 1 || month == 2)
            season = "зиме";
        else if(month == 3 || month == 4 || month == 5)
            season = "весне";
        else if(month == 6 || month == 7 || month == 8)
            season = "лету";
        else if(month == 9 || month == 10 || month == 11)
            season = "осени";
        else
            season = "вымышленным месяцам";
        System.out.println("Апрель относится к " + season + ".");
    }
}
```

Эта программа выводит следующий результат:

Апрель относится к весне

Прежде чем продолжить чтение, поэкспериментируйте с этой программой. Убедитесь сами, что, независимо от значения, присвоенного переменной `month`, в конструкции `if-else-if` будет выполняться только одна операция присваивания.

Оператор `switch`

В языке Java оператор `switch` является оператором ветвления. Он предоставляет простой способ направить поток исполнения команд по разным ветвям кода в зависимости от значения управляющего выражения. Зачастую оператор `switch` оказывается эффективнее длинных последовательностей операторов в конструкции `if-else-if`. Общая форма оператора `switch` имеет следующий вид:

```
switch (выражение) {
    case значение1:
        // последовательность операторов
        break;
    case значение2:
```

```

        // последовательность операторов
        break;
...
...
...
    case значениеN:
        // последовательность операторов
        break;
    default:
        // последовательность операторов по умолчанию
}

```

Во всех версиях Java до JDK 7 указанное выражение должно иметь тип `byte`, `short`, `int`, `char` или перечислимый тип. (Перечисления рассматриваются в главе 12.) Начиная с JDK 7, выражение может также иметь тип `String`. Каждое значение, определенное в операторах ветвей `case`, должно быть однозначным константным выражением (например, литеральным значением). Дублирование значений в операторах ветвей `case` не допускается. Каждое значение должно быть совместимо по типу с указанным выражением.

Оператор `switch` действует следующим образом. Значение выражения сравнивается с каждым значением в операторах ветвей `case`. При обнаружении совпадения выполняется последовательность кода, следующая после оператора данной ветви `case`. Если значения ни одной из констант в операторах ветвей `case` не совпадают со значением выражения, то выполняется оператор в ветви `default`. Но указывать этот оператор не обязательно. В отсутствие совпадений со значениями констант в операторах ветвей `case`, а также оператора `default` никаких дальнейших действий не выполняется.

Оператор `break` служит для прерывания последовательности операторов в ветвях оператора `switch`. Как только очередь доходит до оператора `break`, выполнение продолжается с первой же строки кода, следующей после всего оператора `switch`. Оператор `break` служит для немедленного выхода из оператора `switch`. Ниже представлен простой пример применения оператора `switch`.

```

// Простой пример применения оператора switch
class SampleSwitch {
    public static void main(String args[]) {
        for(int i=0; i<6; i++)
            switch(i) {
                case 0:
                    System.out.println("i равно нулю.");
                    break;
                case 1:
                    System.out.println("i равно единице.");
                    break;
                case 2:
                    System.out.println("i равно двум.");
                    break;
                case 3:
                    System.out.println("i равно трем.");
                    break;
                default:
                    System.out.println("i больше трех.");
            }
    }
}

```


Эта программа выводит следующий результат:

```
i равно нулю.
i равно единице.
i равно двум.
i равно трем.
i больше трех.
i больше трех.
```

Как видите, на каждом шаге цикла в данной программе выполняются операторы, связанные с константой в той ветви `case`, которая соответствует значению переменной `i`, а все остальные операторы пропускаются. После того как значение переменной `i` становится больше 3, оно перестает соответствовать значению константы в любой ветви `case`, и поэтому выполняется оператор `default`.

Указывать оператор `break` необязательно. Если его опустить, то выполнение продолжится с оператора следующей ветви `case`. В некоторых случаях желательно использовать несколько операторов ветвей `case` без разделяющих их операторов `break`. Рассмотрим в качестве примера следующую программу:

```
// В операторе switch необязательно указывать операторы break
class MissingBreak {
    public static void main(String args[]) {
        for(int i=0; i<12; i++)
            switch(i) {
                case 0:
                case 1:
                case 2:
                case 3:
                case 4:
                    System.out.println("i меньше 5");
                    break;
                case 5:
                case 6:
                case 7:
                case 8:
                case 9:
                    System.out.println("i меньше 10");
                    break;
                default:
                    System.out.println("i равно или больше 10");
            }
    }
}
```

Эта программа выводит следующий результат:

```
i меньше 5
i меньше 5
i меньше 5
i меньше 5
i меньше 5
i меньше 10
i меньше 10
i меньше 10
i меньше 10
i меньше 10
i меньше 10
i равно или больше 10
i равно или больше 10
```

Как видите, операторы выполняются в каждой ветви `case` до тех пор, пока не будет достигнут оператор `break` (или конец оператора `switch`). Приведенный выше пример специально создан в качестве иллюстрации, тем не менее пропуск операторов `break` находит немало применений в реальных программах. В качестве более практического примера ниже приведена переделанная версия рассмотренной ранее программы, в которой определяется принадлежность месяца времени года. В этой версии использован оператор `switch`, что позволило добиться более эффективной реализации данной программы.

```
// Усовершенствованная версия программы, в которой
// определяется принадлежность месяца времени года
class Switch {
    public static void main(String args[]) {
        int month = 4;
        String season;
        switch (month) {
            case 12:
            case 1:
            case 2:
                season = "зиме";
                break;
            case 3:
            case 4:
            case 5:
                season = "весне";
                break;
            case 6:
            case 7:
            case 8:
                season = "лету";
                break;
            case 9:
            case 10:
            case 11:
                season = "осени";
                break;
            default:
                season = "вымышленным месяцам";
        }
        System.out.println("Апрель относится к" + season + ".");
    }
}
```

Как упоминалось ранее, начиная с **JDK 7**, для управления оператором `switch` можно использовать символьные строки.

```
// Использовать символьные строки для управления оператором switch
class StringSwitch {
    public static void main(String args[]) {
        String str = "два";
        switch(str) {
            case "один":
                System.out.println("один");
                break;
            case "два":
                System.out.println("два");
                break;
            case "три":
                System.out.println("три");
        }
    }
}
```

```

        break;
    default:
        System.out.println("не совпало");
        break;
    }
}

```

Как и следовало ожидать, результат выполнения этой программы будет следующим:

два

Символьная строка, содержащаяся в переменной `str` (в данном случае — "два"), сравнивается с константами в операторах ветвей `case`. Как только обнаружится совпадение (в операторе второй ветви `case`), выполняется связанная с ним последовательность кода.

Возможность использования символьных строк в операторе `switch` зачастую упрощает дело. В частности, применение оператора `switch` с символьными строками является значительным усовершенствованием по сравнению с эквивалентной последовательностью операторов `if/else`. Но с точки зрения эффективности кода выбор среди символьных строк обходится дороже выбора среди целых чисел. Поэтому символьные строки лучше применять только в тех случаях, когда управляющие данные уже находятся в строковой форме. Иными словами, пользоваться символьными строками в операторе `switch` без особой необходимости не следует.

Вложенные операторы `switch`

Оператор `switch` можно использовать в последовательности операторов внешнего оператора `switch`. Такой оператор `switch` называется *вложенным*. В каждом операторе `switch` определяется свой блок кода, и поэтому никаких конфликтов между константами в ветвях `case` внутреннего и внешнего операторов `switch` не возникает. Например, следующий фрагмент кода вполне допустим:

```

switch(count) {
    case 1:
        switch(target) { // вложенный оператор switch
            case 0:
                System.out.println("target равно 0");
                break;
            case 1: // конфликты с внешним оператором switch отсутствуют
                System.out.println("target равно 1");
                break;
        }
        break;
    case 2: // ...

```

В данном случае оператор ветви `case 1`: внутреннего оператора `switch` не конфликтует с оператором ветви `case 1`: внешнего оператора `switch`. Значение переменной `count` сравнивается только с рядом внешних ветвей `case`. Если значение переменной `count` равно 1, то значение переменной `target` сравнивается с рядом внутренних ветвей `case`.

Таким образом, можно выделить следующие важные особенности оператора `switch`.

- Оператор `switch` отличается от условного оператора `if` тем, что в нем допускается выполнять проверку только на равенство, тогда как в условном операторе `if` можно вычислять результат логического выражения любого типа. Следовательно, в операторе `switch` обнаруживается совпадение выражения с константой только в одной из ветвей `case`.
- Константы ни в одной из двух ветвей `case` того же самого оператора `switch` не могут иметь одинаковые значения. Безусловно, внутренний оператор `switch` и содержащий его внешний оператор `switch` могут иметь одинаковые константы в ветвях `case`.
- Как правило, оператор `switch` действует эффективнее ряда вложенных условных операторов `if`.

Последняя особенность представляет особый интерес, поскольку она позволяет лучше понять принцип действия компилятора Java. Компилируя оператор `switch`, компилятор Java будет проверять константу в каждой ветви `case` и создавать “таблицу переходов”, чтобы использовать ее для выбора ветви программы в зависимости от получаемого значения выражения. Поэтому в тех случаях, когда требуется делать выбор среди большой группы значений, оператор `switch` будет выполняться значительно быстрее последовательности операторов `if-else`. Ведь компилятору известно, что константы всех ветвей `case` имеют один и тот же тип, и их достаточно проверить на равенство значению выражения `switch`. В то же время компилятор не располагает подобными сведениями о длинном перечне выражений условного оператора `if`.

Операторы цикла

Для управления конструкциями, которые обычно называются *циклами*, в Java предоставляются операторы `for`, `while` и `do-while`. Вам, должно быть, известно, что циклы многократно выполняют один и тот же набор инструкций до тех пор, пока не будет удовлетворено условие завершения цикла. Как станет ясно в дальнейшем, операторы цикла в Java способны удовлетворить любые потребности в программировании.

Цикл `while`

Оператор цикла `while` является самым основополагающим для организации циклов в Java. Он повторяет оператор или блок операторов до тех пор, пока значение его управляющего выражения истинно. Этот оператор цикла имеет следующую общую форму:

```
while (условие) {  
    // тело цикла  
}
```

где *условие* обозначает любое логическое выражение. Тело цикла будет выполняться до тех пор, пока условное выражение истинно. Когда *условие* становится ложным, управление передается строке кода, непосредственно следующей за циклом. Фигурные скобки могут быть опущены только в том случае, если в цикле повторяется лишь один оператор.

В качестве примера рассмотрим цикл `while`, в котором выполняется обратный отсчет, начиная с 10, и выводится ровно 10 строк “тактов”:

```
// Продемонстрировать применение оператора цикла while
class While {
    public static void main(String args[]) {
        int n = 10;
        while(n > 0) {
            System.out.println("такт " + n);
            n--;
        }
    }
}
```

После запуска эта программа выводит десять “тактов” следующим образом:

```
такт 10
такт 9
такт 8
такт 7
такт 6
такт 5
такт 4
такт 3
такт 2
такт 1
```

В начале цикла `while` вычисляется условное выражение, поэтому тело цикла не будет выполнено ни разу, если в самом начале условие оказывается ложным. Например, в следующем фрагменте кода метод `println()` вообще не будет вызван:

```
int a = 10, b = 20;

while(a > b)
    System.out.println("Эта строка выводиться не будет");
```

Тело цикла `while` (или любого другого цикла в Java) может быть пустым. Это обусловлено тем, что синтаксис Java допускает применение *пустого оператора*, содержащего только знак точки с запятой. Рассмотрим в качестве примера следующую программу:

```
// Целевая часть цикла может быть пустой
class NoBody {
    public static void main(String args[]) {
        int i, j;

        i = 100;
        j = 200;

        // рассчитать среднее значение переменных i и j
        while(++i < --j) ; // у этого цикла отсутствует тело
```

```
        System.out.println("Среднее значение равно " + i);  
    }  
}
```

В этой программе вычисляется среднее значение переменных *i* и *j* и выводится следующий результат:

Среднее значение равно 150

Приведенный выше цикл `while` действует следующим образом. Значение переменной *i* увеличивается, а значение переменной *j* уменьшается на единицу. Затем эти два значения сравниваются. Если новое значение переменной *i* по-прежнему меньше нового значения переменной *j*, цикл повторяется. Если же значение переменной *i* равно значению переменной *j* или больше него, то цикл завершается. После выхода из цикла переменная *i* будет содержать среднее исходных значений переменных *i* и *j*. (Безусловно, такая процедура оказывается работоспособной только в том случае, если в самом начале цикла значение переменной *i* меньше значения переменной *j*.) Как видите, никакой потребности в наличии тела цикла не существует. Все действия выполняются в самом условном выражении. В профессионально написанной программе на Java короткие циклы зачастую не содержат тела, если само по себе управляющее выражение может выполнять все необходимые действия.

Цикл `do-while`

Как было показано выше, если в начальный момент условное выражение, управляющее циклом `while`, ложно, то тело цикла вообще не будет выполняться. Но иногда тело цикла желательно выполнить хотя бы один раз, даже если в начальный момент условное выражение ложно. Иначе говоря, возможны случаи, когда проверку условия прерывания цикла желательно выполнять в конце цикла, а не в начале. Для этой цели в Java предоставляется цикл, который называется `do-while`. Тело этого цикла всегда выполняется хотя бы один раз, поскольку его условное выражение проверяется в конце цикла. Общая форма цикла `do-while` следующая:

```
do {  
    // тело цикла  
} while (условие);
```

При каждом повторении цикла `do-while` сначала выполняется тело цикла, а затем вычисляется условное выражение. Если это выражение истинно, цикл повторяется. В противном случае выполнение цикла прерывается. Как и во всех циклах в Java, заданное условие должно быть логическим выражением.

Ниже приведена переделанная программа вывода тактов, в которой демонстрируется применение оператора цикла `do-while`. Эта версия программы выводит такой же результат, как и ее предыдущая версия.

```
// Продемонстрировать применение оператора цикла do-while  
class DoWhile {  
    public static void main(String args[]) {  
        int n = 10;
```

```

do {
    System.out.println("такт " + n);
    n--;
} while(n > 0);
}

```

Формально цикл в приведенной выше программе организован правильно. Тем не менее его можно переписать и в более эффективной форме, как показано ниже.

```

do {
    System.out.println("такт " + n);
} while(--n > 0);

```

В данном случае декремент переменной `n` и сравнение результирующего значения с нулем объединены в одном выражении (`--n > 0`). Это выражение действует следующим образом. Сначала выполняется операция декремента `--n`, уменьшая значение переменной `n` на единицу и возвращая новое значение переменной `n`, а затем это значение сравнивается с нулем. Если оно больше нуля, то выполнение цикла продолжается, а иначе цикл завершается.

Цикл `do-while` особенно удобен при выборе пункта меню, поскольку в этом случае обычно требуется, чтобы тело цикла меню выполнялось, по меньшей мере, один раз. Рассмотрим в качестве примера следующую программу, в которой реализуется очень простая система справки по операторам выбора и цикла в Java:

```

// Использовать оператор цикла do-while для выбора пункта меню
class Menu {
    public static void main(String args[])
        throws java.io.IOException {
        char choice;
        do {
            System.out.println("Справка по оператору:");
            System.out.println("    1. if");
            System.out.println("    2. switch");
            System.out.println("    3. while");
            System.out.println("    4. do-while");
            System.out.println("    5. for\n");
            System.out.println("Выберите нужный пункт:");
            choice = (char) System.in.read();
        } while( choice < '1' || choice > '5');

        System.out.println("\n");

        switch(choice) {
            case '1':
                System.out.println("if:\n");
                System.out.println("if(условие) оператор;");
                System.out.println("else оператор;");
                break;
            case '2':
                System.out.println("switch:\n");
                System.out.println("switch(выражение) {");
                System.out.println("    case константа:");
                System.out.println("        последовательность операторов");
                System.out.println("    break;");
                System.out.println("    // ...");
                System.out.println("}");
                break;
            case '3':
                System.out.println("while:\n");

```

```
        System.out.println("while(условие) оператор;");
        break;
    case '4':
        System.out.println("do-while:\n");
        System.out.println("do {");
        System.out.println("    оператор;");
        System.out.println("} while (условие);");
        break;
    case '5':
        System.out.println("for:\n");
        System.out.print("for(инициализация; условие; итерация)");
        System.out.println(" оператор;");
        break;
    }
}
```

Ниже приведен пример выполнения этой программы.

Справка по оператору:

1. if
2. switch
3. while
4. do-while
5. for

Выберите нужный пункт:

```
4
do-while:
do {
    оператор;
} while (условие);
```

В данной программе проверка допустимости введенного пользователем значения осуществляется в цикле do-while. Если это значение недопустимо, то пользователю предлагается повторить ввод. А поскольку меню должно отобразиться хотя бы один раз, то цикл do-while оказывается идеальным средством для решения этой задачи.

У данной программы имеется ряд других особенностей. В частности, для ввода символов с клавиатуры вызывается метод `System.in.read()`, выполняющий одну из функций консольного ввода в Java. Более подробно методы консольного ввода-вывода рассматриваются в главе 13, а до тех пор следует заметить, что в данном случае метод `System.in.read()` служит для получения результата выбора, сделанного пользователем. Этот метод вводит символы из стандартного потока ввода, откуда они возвращаются в виде целочисленных значений. Именно поэтому тип возвращаемого значения приводится к типу `char`. По умолчанию данные из стандартного потока ввода размещаются в буфере построчно, поэтому нужно нажать клавишу `<Enter>`, чтобы любые введенные символы были переданы программе.

Консольный ввод в Java может вызывать некоторые затруднения. Более того, большинство реальных программ на Java разрабатываются графическими и ориентированными на работу в оконном режиме. Поэтому в данной книге консольному вводу уделяется не очень много внимания. Но в данном случае он удобен. Следует также иметь в виду, что данная программа должна содержать выражение `throws java.io.IOException`, поскольку в ней используется метод `System.in.read()`. Это выражение необходимо для обработки ошибок ввода и является составной частью системы обработки исключений в Java, которая рассматривается в главе 10.

Цикл `for`

Простая форма оператора цикла `for` была представлена в главе 2, но, как будет показано ниже, это довольно эффективная и универсальная языковая конструкция. Начиная с версии JDK 5, в Java имеются две формы оператора цикла `for`. Первая форма считается традиционной и появилась еще в исходной версии Java, а вторая — более новая форма цикла в стиле `for each`. В этом разделе рассматриваются обе формы оператора цикла `for`, начиная с традиционной.

Общая форма традиционной разновидности оператора цикла `for` выглядит следующим образом:

```
for (инициализация; условие; итерация) {  
    // тело  
}
```

Если в цикле повторяется выполнение только одного оператора, то фигурные скобки можно опустить.

Цикл `for` действует следующим образом. Когда цикл начинается, выполняется его *инициализация*. В общем случае это выражение, устанавливающее значение *переменной управления циклом*, которая действует в качестве счетчика, управляющего циклом. Важно понимать, что первая часть цикла `for`, содержащая инициализирующее выражение, выполняется только один раз. Затем вычисляется заданное *условие*, которое должно быть логическим выражением. Как правило, в этом выражении значение управляющей переменной сравнивается с целевым значением. Если результат этого сравнения истинный, то выполняется тело цикла. А если он ложный, то цикл завершается. И наконец, выполняется третья часть цикла `for` — *итерация*. Обычно эта часть цикла содержит выражение, в котором увеличивается или уменьшается значение переменной управления циклом. Затем цикл повторяется, и на каждом его шаге сначала вычисляется условное выражение, затем выполняется тело цикла, а после этого вычисляется итерационное выражение. Этот процесс повторяется до тех пор, пока результат вычисления итерационного выражения не станет ложным.

Ниже приведена версия программы подсчета “тактов”, в которой применяется оператор цикла `for`.

```
// Продемонстрировать применение оператора цикла for  
class ForTick {  
    public static void main(String args[]) {  
        int n;  
  
        for(n=10; n>0; n--)  
            System.out.println("такт " + n);  
    }  
}
```

Объявление переменных, управляющих циклом `for`

Зачастую переменная, управляющая циклом `for`, требуется только для него и нигде больше не используется. В таком случае переменную управления циклом можно объявить в инициализирующей части оператора `for`. Например, предыдущую программу можно переписать, объявив управляющую переменную `n` типа `int` в самом цикле `for`.

```
// Объявить переменную управления циклом в самом цикле for
class ForTick {
    public static void main(String args[]) {

        // здесь переменная n объявляется в самом цикле for
        for(int n=10; n>0; n--)
            System.out.println("такт " + n);
    }
}
```

Объявляя переменную управления циклом `for` в самом цикле, не следует забывать, что область и срок действия этой переменной полностью совпадают с областью и сроком действия оператора цикла `for`. Это означает, что область действия переменной управления циклом `for` ограничивается пределами самого цикла. А за пределами цикла `for` эта переменная прекращает свое существование. Если же переменную управления циклом `for` требуется использовать в других частях программы, ее нельзя объявлять в самом цикле.

В тех случаях, когда переменная управления циклом `for` нигде больше не требуется, большинство программирующих на Java предпочитают объявлять ее в самом операторе цикла `for`. В качестве примера ниже приведена простая программа, в которой проверяется, является ли число простым. Обратите внимание на то, что переменная `i` управления циклом объявлена в самом цикле `for`, поскольку она нигде больше не требуется.

```
// Проверить на простые числа
class FindPrime {
    public static void main(String args[]) {
        int num;
        boolean isPrime;

        num = 14;

        if(num < 2) isPrime = false;
        else isPrime = true;

        for(int i=2; i <= num/i; i++) {
            if((num % i) == 0) {
                isPrime = false;
                break;
            }
        }
        if(isPrime) System.out.println("Простое число");
        else System.out.println("Не простое число");
    }
}
```

Использование запятой

В ряде случаев требуется указать несколько операторов в инициализирующей и итерационной частях оператора цикла `for`. Рассмотрим в качестве примера цикл в следующей программе:

```
class Sample {
    public static void main(String args[]) {
        int a, b;

        b = 4;
```

```

        for(a=1; a<b; a++) {
            System.out.println("a = " + a);
            System.out.println("b = " + b);
            b--;
        }
    }
}

```

Как видите, управление этим циклом осуществляется одновременно двумя переменными. А поскольку цикл управляется двумя переменными, то их желательно включить в сам оператор цикла `for`, а не выполнять обработку переменной `b` вручную. Правда, для решения этой задачи в Java предоставляется специальная возможность. Для того чтобы две переменные или больше могли управлять циклом `for`, в Java допускается указывать несколько операторов как в инициализирующей, так и в итерационной части оператора цикла `for`, разделяя их запятыми.

Используя запятую, предыдущий цикл `for` можно организовать более рационально:

```

// Использование запятой в операторе цикла for
class Comma {
    public static void main(String args[]) {
        int a, b;

        for(a=1, b=4; a<b; a++, b--) {
            System.out.println("a = " + a);
            System.out.println("b = " + b);
        }
    }
}

```

В данном примере программы начальные значения обеих переменных, `a` и `b`, управления циклом `for` устанавливаются в инициализирующей части цикла. Оба разделяемых запятой оператора в итерационной части цикла выполняются при каждом повторении цикла. Эта программа выводит следующий результат:

```

a = 1
b = 4
a = 2
b = 3

```

На заметку! Тем, у кого имеется опыт программирования на C/C++, должно быть известно, что в этих языках запятая обозначает операцию, которую можно использовать в любом допустимом выражении. А в Java запятая служит лишь в качестве разделителя.

Разновидности цикла `for`

У оператора цикла `for` имеется несколько разновидностей, расширяющих возможности его применения. Гибкость этого цикла объясняется тем, что три его части (инициализацию, проверку условий и итерацию) совсем не обязательно использовать только по прямому назначению. По существу, каждую часть оператора цикла `for` можно применять в любых требующихся целях. Рассмотрим несколько примеров такого применения.

В одной из наиболее часто встречающихся разновидностей цикла `for` предполагается употребление условного выражения. В частности, в этом выражении

совсем не обязательно сравнивать переменную управления циклом с некоторым целевым значением. По существу, условием, управляющим циклом `for`, может быть любое логическое выражение. Рассмотрим в качестве примера следующий фрагмент кода:

```
boolean done = false;

for(int i=1; !done; i++) {
    // ...
    if(interrupted()) done = true;
}
```

В этом примере выполнение цикла `for` продолжается до тех пор, пока в переменной `done` не установится логическое значение `true`. В этой разновидности цикла `for` не выполняется проверка значения в переменной `i` управления циклом.

Рассмотрим еще одну интересную разновидность цикла `for`. Инициализирующее или итерационное выражения или оба вместе могут отсутствовать в операторе цикла `for`, как показано ниже.

```
// Отдельные части оператора цикла for могут отсутствовать
class ForVar {
    public static void main(String args[]) {
        int i;
        boolean done = false;

        i = 0;
        for( ; !done; ) {
            System.out.println("i равно " + i);
            if(i == 10) done = true;
            i++;
        }
    }
}
```

В данном примере инициализирующее и итерационное выражения вынесены за пределы цикла `for`. В итоге соответствующие части оператора цикла `for` оказываются пустыми. В данном очень простом примере, демонстрирующем далеко не самый изящный стиль программирования, это не так важно, но иногда такой подход имеет смысл. Так, если начальное условие определяется сложным выражением где-то в другом месте программы или значение переменной управления циклом изменяется случайным образом в зависимости от действий, выполняемых в теле цикла, то эти части оператора цикла `for` имеет смысл оставить пустыми.

Приведем еще одну разновидность цикла `for`. Оставляя все три части оператора пустыми, можно умышленно создать бесконечный цикл, т.е. такой цикл, который никогда не завершается:

```
for( ; ; ) {
    // ...
}
```

Этот цикл может выполняться бесконечно, поскольку отсутствует условие, по которому он мог бы завершиться. Если в некоторых программах вроде командного процессора операционной системы требуется наличие бесконечного цикла, то в большинстве случаев “бесконечные” циклы на самом деле являются лишь циклами с особыми условиями прерывания. Как будет показано ниже, существует

способ прервать цикл (даже бесконечный, как в приведенном выше примере), не требующий указывать обычное условное выражение в цикле.

Разновидность цикла `for` в стиле `for each`

Начиная с версии JDK 5, в Java можно использовать вторую форму цикла `for`, реализующую цикл в стиле `for each`. Вам, вероятно, известно, что в современной теории языков программирования все большее применение находит понятие циклов в стиле `for each`, которые постепенно становятся стандартными средствами во многих языках программирования. Цикл в стиле `for each` предназначен для строго последовательного выполнения повторяющихся действий над коллекцией объектов вроде массива. В отличие от некоторых языков, подобных C#, где для реализации циклов в стиле `for each` используется ключевое слово `foreach`, в Java возможность организации такого цикла реализована путем усовершенствования цикла `for`. Преимущество такого подхода состоит в том, что для его реализации не требуется дополнительное ключевое слово, а уже существующий код не нарушается. Цикл `for` в стиле `for each` называется также *усовершенствованным* циклом `for`. Общая форма разновидности цикла `for` в стиле `for each` имеет следующий вид:

`for(тип итерационная_переменная : коллекция) блок_операторов`

где *тип* обозначает конкретный тип данных; *итерационная_переменная* — имя *итерационной переменной*, которая последовательно принимает значения из коллекции: от первого и до последнего; *коллекция* — перебираемую в цикле коллекцию. В цикле `for` можно перебирать разные типы коллекций, но здесь для этой цели будут использоваться только массивы. (Другие типы коллекций, которые можно перебирать в цикле `for`, в том числе и те, что определены в каркасе коллекций Collection Framework, рассматриваются в последующих главах.) На каждом шаге цикла из коллекции извлекается очередной элемент, который сохраняется в указанной *итерационной_переменной*. Цикл выполняется до тех пор, пока из коллекции не будут извлечены все элементы.

Поскольку итерационная переменная получает значения из коллекции, *тип* должен совпадать (или быть совместимым) с типом элементов, хранящихся в коллекции. Таким образом, при переборе массива *тип* должен быть совместим с типом элемента массива.

Чтобы стали понятнее побудительные причины для применения циклов в стиле `for each`, рассмотрим разновидность цикла `for`, для замены которого этот стиль предназначен. В следующем фрагменте кода для вычисления суммы значений элементов массива применяется традиционный цикл `for`:

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;

for(int i=0; i < 10; i++) sum += nums[i];
```

Чтобы вычислить сумму, значения каждого элемента последовательно извлекаются из массива `nums`. Таким образом, чтение всего массива выполняется в строгой последовательности. Это достигается благодаря индексации массива `nums` вручную по переменной `i` управления циклом.

Цикл `for` в стиле `for each` позволяет автоматизировать этот процесс. В частности, применяя такой цикл, можно не устанавливать значение счетчика цикла, указывать его начальное и конечное значения, а также индексировать массив вручную. Вместо этого цикл выполняется автоматически по всему массиву, последовательно получая значения каждого его элементов: от первого до последнего. Например, предыдущий фрагмент кода можно переписать, используя вариант цикла `for` в стиле `for each` следующим образом:

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;

for(int x: nums) sum += x;
```

На каждом шаге цикла переменной `x` автоматически присваивается значение следующего элемента массива `nums`. Таким образом, на первом шаге цикла переменная `x` содержит значение 1, на втором шаге — значение 2 и т.д. Такая разновидность цикла `for` не только упрощается синтаксис, но и исключает возможность ошибок, связанных с выходом за пределы массива.

Ниже приведен полноценный пример программы, демонстрирующий применение описанной выше разновидности цикла `for` в стиле `for each`.

```
// Применение цикла for в стиле for each
class ForEach {
    public static void main(String args[]) {
        int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        int sum = 0;

        // использовать цикл в стиле for each для вывода и
        // суммирования значений
        for(int x : nums) {
            System.out.println("Значение равно: " + x);
            sum += x;
        }

        System.out.println("Сумма равна: " + sum);
    }
}
```

Эта программа выводит следующий результат:

```
Значение равно: 1
Значение равно: 2
Значение равно: 3
Значение равно: 4
Значение равно: 5
Значение равно: 6
Значение равно: 7
Значение равно: 8
Значение равно: 9
Значение равно: 10
Сумма равна: 55
```

Как следует из этого результата, оператор цикла `for` в стиле `for each` автоматически перебирает элементы массива, от наименьшего индекса к наибольшему. Повторение цикла `for` в стиле `for each` выполняется до тех пор, пока не будут перебраны все элементы массива, но этот цикл можно прервать и раньше, используя оператор `break`. В следующем примере программы суммируются значения пяти первых элементов массива `nums`:

```
// Применение оператора break в цикле for в стиле for each
class ForEach2 {
    public static void main(String args[]) {
        int sum = 0;
        int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

        // использовать цикл for в стиле for each для
        // вывода и суммирования значений из части массива
        for(int x : nums) {
            System.out.println("Значение равно: " + x);
            sum += x;
            if(x == 5) break; // прервать цикл после
                           // получения 5 значений
        }
        System.out.println("Сумма пяти первых элементов равна: " + sum);
    }
}
```

Эта программа выводит следующий результат:

```
Значение равно: 1
Значение равно: 2
Значение равно: 3
Значение равно: 4
Значение равно: 5
Сумма пяти первых элементов равна: 15
```

Как видите, выполнение цикла прерывается после того, как будет получено значение пятого элемента массива. Оператор `break` можно использовать и в других операторах цикла, доступных в Java. Подробнее оператор `break` рассматривается далее в этой главе.

Применяя цикл `for` в стиле `for each`, не следует забывать о том, что его итерационная переменная доступна “только для чтения”, поскольку она связана только с исходным массивом. Присваивание значения итерационной переменной не оказывает никакого влияния на исходный массив. Иначе говоря, содержимое массива нельзя изменить, присваивая новое значение итерационной переменной. Рассмотрим в качестве примера следующую программу:

```
// Переменная цикла в стиле for each доступна только для чтения
class NoChange {
    public static void main(String args[]) {
        int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

        for(int x : nums) {
            System.out.print(x + " ");
            x = x * 10; // этот оператор не оказывает никакого
                     // влияния на массив nums
        }

        System.out.println();

        for(int x : nums)
            System.out.print(x + " ");
        System.out.println();
    }
}
```

В первом цикле `for` значение итерационной переменной увеличивается на 10. Но это присваивание не оказывает никакого влияния на исходный массив `nums`,

как видно из результата выполнения второго оператора `for`. Следующий результат, выводимый данной программой, подтверждает сказанное:

```
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
```

Итерация в многомерных массивах

Усовершенствованная разновидность цикла `for` распространяется и на многомерные массивы. Но не следует забывать, что в Java многомерные массивы представляют собой *массивы массивов*. Например, двумерный массив — это массив одномерных массивов. Это обстоятельство важно иметь в виду при переборе многомерного массива, поскольку результатом каждой итерации оказывается *следующий массив*, а не отдельный элемент. Более того, тип итерационной переменной цикла `for` должен быть совместим с типом получаемого массива. Например, в двумерном массиве итерационная переменная должна быть ссылкой на одномерный массив. В общем случае при использовании цикла в стиле `for each` для перебора массива размерностью N получаемые в итоге объекты будут массивами размерностью $N-1$. Чтобы стало понятнее, что из этого следует, рассмотрим следующий пример программы, где вложенные циклы `for` служат для получения упорядоченных по строкам элементов двумерного массива: от первого до последнего:

```
// Применение цикла for в стиле for each для
// обращения к двумерному массиву
class ForEach3 {
    public static void main(String args[]) {
        int sum = 0;
        int nums[][] = new int[3][5];

        // присвоить значение элементам массива nums
        for(int i = 0; i < 3; i++)
            for(int j = 0; j < 5; j++)
                nums[i][j] = (i+1)*(j+1);

        // использовать цикл for в стиле for each для
        // вывода и суммирования значений
        for(int x[] : nums) {
            for(int y : x) {
                System.out.println("Значение равно: " + y);
                sum += y;
            }
        }
        System.out.println("Сумма: " + sum);
    }
}
```

Эта программа выводит следующий результат:

```
Значение равно: 1
Значение равно: 2
Значение равно: 3
Значение равно: 4
Значение равно: 5
Значение равно: 2
Значение равно: 4
Значение равно: 6
Значение равно: 8
Значение равно: 10
```



```

Значение равно: 3
Значение равно: 6
Значение равно: 9
Значение равно: 12
Значение равно: 15
Сумма: 90

```

Следующая строка кода из данной программы заслуживает особого внимания:

```
for(int x[] : nums) {
```

Обратите внимание на порядок объявления переменной *x*. Эта переменная является ссылкой на одномерный массив целочисленных значений. Это необходимо, потому что результатом выполнения каждого шага цикла `for` является следующий *массив* в массиве `nums`, начиная с массива, обозначаемого элементом `nums[0]`. Затем каждый из этих массивов перебирается во внутреннем цикле `for`, где выводится значение каждого элемента.

Применение усовершенствованного цикла `for`

Каждый оператор цикла `for` в стиле `for each` позволяет перебирать элементы массива только по очереди, начиная с первого и оканчивая последним, и поэтому может показаться, что его применение ограничено. Но на самом деле это не так. Ведь именно такой механизм требуется во многих алгоритмах. Одним из наиболее часто применяемых является алгоритм поиска. В приведенном ниже примере программы цикл `for` использует для поиска значения в неупорядоченном массиве. Поиск прекращается после обнаружения искомого значения.

```

// Поиск в массиве с применением цикла for в стиле for each
class Search {
    public static void main(String args[]) {
        int nums[] = { 6, 8, 3, 7, 5, 6, 1, 4 };
        int val = 5;
        boolean found = false;

        // использовать цикл for в стиле for each для
        // поиска значения переменной val в массиве nums
        for(int x : nums) {
            if(x == val) {
                found = true;
                break;
            }
        }

        if(found)
            System.out.println("Значение найдено!");
    }
}

```

В данном случае выбор стиля `for each` для организации цикла `for` полностью оправдан, поскольку поиск в неупорядоченном массиве предполагает последовательный просмотр каждого элемента. (Безусловно, если бы массив был упорядоченным, можно было бы организовать двоичный поиск, реализация которого потребовала бы применения цикла другого стиля.) К другим примерам, где применение циклов в стиле `for each` дает заметные преимущества, относятся вычисление среднего значения, отыскание минимального или максимального значения в множестве, поиск дубликатов и т.п.

Несмотря на то что в примерах, приведенных ранее в этой главе, были использованы массивы, цикл `for` в стиле `for each` особенно удобен при обращении с коллекциями, определенными в каркасе `Collections Framework`, описываемом в части II данной книги. В более общем случае оператор `for` позволяет перебирать элементы любой коллекции объектов, если эта коллекция удовлетворяет определенному ряду ограничений, описываемых в главе 18.

Вложенные циклы

Как и в других языках программирования, в Java допускается применение вложенных циклов. Это означает, что один цикл может выполняться в другом. В следующем примере программы используются вложенные циклы `for`:

```
// Циклы могут быть вложенными
class Nested {
    public static void main(String args[]) {
        int i, j;

        for(i=0; i<10; i++) {
            for(j=i; j<10; j++)
                System.out.print(".");
            System.out.println();
        }
    }
}
```

Эта программа выводит следующий результат:

```
.....
.....
.....
.....
.....
.....
....
...
..
.
```

Операторы перехода

В языке Java определены три оператора перехода: `break`, `continue` и `return`. Они передают непосредственное управление другой части программы. Рассмотрим каждый из них в отдельности.

На заметку! Кроме операторов перехода, рассматриваемых в этом разделе, в Java поддерживает еще один способ изменения порядка выполнения инструкций программы, который состоит в обработке исключений. Обработка исключений предоставляет структурированный механизм, с помощью которого можно обнаруживать и обрабатывать ошибки во время выполнения программы. Для поддержки этого механизма служат ключевые слова `try`, `catch`, `throws` и `finally`. По существу, механизм обработки исключений позволяет выполнять нелокальные ветви программы. Впрочем, тема обработки исключений настолько обширна, что она рассматривается отдельно в специально посвященной ей главе 10.

Применение оператора `break`

В языке Java оператор `break` находит три применения. Во-первых, как было показано ранее, он завершает последовательность операторов в операторе `switch`. Во-вторых, его можно использовать для выхода из цикла. И в-третьих, этот оператор можно применять в качестве “цивилизованной” формы оператора безусловного перехода `goto`. Рассмотрим два последних применения данного оператора.

Использование оператора `break` для выхода из цикла

Используя оператор `break`, можно вызвать немедленное завершение цикла, пропуская условное выражение и любой остальной код в теле цикла. Когда в теле цикла встречается оператор `break`, выполнение цикла прекращается и управление передается оператору, следующему за циклом. Ниже приведен простой тому пример.

```
// Применение оператора break для выхода из цикла
class BreakLoop {
    public static void main(String args[]) {
        for(int i=0; i<100; i++) {
            if(i == 10) break; // выход из цикла, если значение
                               // переменной i равно 10
            System.out.println("i: " + i);
        }
        System.out.println("Цикл завершен.");
    }
}
```

Эта программа выводит следующий результат:

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Цикл завершен.
```

Как видите, оператор `break` приводит к более раннему выходу из цикла `for`, когда значение переменной `i` становится равным 10, хотя цикл `for` должен был бы выполняться при значениях переменной управления цикла в пределах от 0 до 99.

Оператор `break` можно использовать в любых циклах, доступных в Java, включая преднамеренно бесконечные циклы. Так, ниже приведен переделанный вариант предыдущего примера программы где применяется цикл `while`. В этом варианте программа выводит такой же результат, как и в предыдущем.

```
// Применение оператора break для выхода из цикла while
class BreakLoop2 {
    public static void main(String args[]) {
        int i = 0;
```

```

while(i < 100) {
    if(i == 10) break; // выход из цикла, если значение
                      // переменной i равно 10
    System.out.println("i: " + i);
    i++;
}
System.out.println("Цикл завершен.");
}
}

```

Если в программе применяется ряд вложенных циклов, то оператор `break` осуществляет выход только из самого внутреннего цикла. Ниже приведен характерный тому пример.

```

// Применение оператора break во вложенных циклах
class BreakLoop3 {
    public static void main(String args[]) {
        for(int i=0; i<3; i++) {
            System.out.print("Проход " + i + ": ");
            for(int j=0; j<100; j++) {
                if(j == 10) break; // выход из цикла, значение
                                // переменной j равно 10
                System.out.print(j + " ");
            }
            System.out.println();
        }
        System.out.println("Циклы завершены.");
    }
}

```

Эта программа выводит следующий результат:

```

Проход 0: 0 1 2 3 4 5 6 7 8 9
Проход 1: 0 1 2 3 4 5 6 7 8 9
Проход 2: 0 1 2 3 4 5 6 7 8 9
Циклы завершены.

```

Как видите, оператор `break` во внутреннем цикле может приводить к выходу только из этого цикла. А на внешний цикл он не оказывает никакого влияния.

Используя оператор `break`, необходимо помнить следующее. Во-первых, в цикле можно использовать больше одного оператора `break`. Но при этом следует соблюдать осторожность. Как правило, применение слишком большого количества операторов `break` приводит к нарушению структуры кода. Во-вторых, оператор `break`, завершающий последовательность операторов, выполняемых в операторе `switch`, оказывает влияние только на данный оператор `switch`, но не на любые содержащие его циклы.

Помните! Оператор `break` не предназначен в качестве обычного средства выхода из цикла. Для этого служит условное выражение в цикле. Этот оператор следует использовать для выхода из цикла только в особых случаях.

Применение оператора `break` в качестве формы оператора `goto`

Оператор `break` можно применять не только в операторах `switch` и циклах, но и самостоятельно в качестве “цивилизованной” формы оператора безусловного

перехода `goto`. В языке Java оператор `goto` отсутствует, поскольку он позволяет выполнять ветвление программ произвольным и неструктурированным образом. Как правило, код, который управляется оператором `goto`, труден для понимания и сопровождения. Кроме того, этот оператор исключает возможность оптимизировать код для определенного компилятора. Но в некоторых случаях оператор `goto` оказывается удобным и вполне допустимым средством для управления потоком исполнения команд. Например, оператор `goto` может оказаться полезным при выходе из ряда глубоко вложенных циклов. Для подобных случаев в Java определена расширенная форма оператора `break`. Используя эту форму, можно, например, организовать выход из одного или нескольких блоков кода. Они совсем не обязательно должны быть частью цикла или оператора `switch`, но могут быть любыми блоками кода. Более того, можно точно указать оператор, с которого будет продолжено выполнение программы, поскольку данная форма оператора `break` наделена метками. Как будет показано ниже, оператор `break` с меткой предоставляет все преимущества оператора `goto`, не порождая присущие ему недостатки. Общая форма оператора `break` с меткой имеет следующий вид:

break метка;

Чаще всего *метка* — это имя метки, обозначающее блок кода. Им может быть как самостоятельный блок кода, так и целевой блок другого оператора. При выполнении этой формы оператора `break` управление передается блоку кода, помеченному меткой. Такой блок кода должен содержать оператор `break`, но он не обязательно должен быть непосредственно объемлющим его блоком. В частности, это означает, что оператор `break` с меткой можно применять для выхода из ряда вложенных блоков. Но его нельзя использовать для передачи управления внешнему блоку кода, который не содержит данный оператор `break`.

Чтобы пометить блок, достаточно поместить в его начале метку. *Метка* — это любой допустимый в Java идентификатор с двоеточием. Как только блок помечен, его метку можно использовать в качестве адресата для оператора `break`. В итоге выполнение программы будет продолжено с *конца* помеченного блока. Так, в приведенном ниже примере программа содержит три вложенных блока, каждый из которых помечен отдельной меткой. Оператор `break` осуществляет переход в конец блока с меткой `second`, пропуская два вызова метода `println()`.

```
// Применение оператора break в качестве цивилизованной
// формы оператора goto
class Break {
    public static void main(String args[]) {
        boolean t = true;

        first: {
            second: {
                third: {
                    System.out.println("Предшествует оператору break.");
                    if(t) break second; // выход из блока second
                    System.out.println("Этот оператор не будет выполняться");
                }
                System.out.println("Этот оператор не будет выполняться");
            }
            System.out.println("Этот оператор следует за блоком second.");
        }
    }
}
```

```

    }
}

```

Эта программа выводит следующий результат:

Предшествует оператору `break`.
Этот оператор следует за блоком `second`.

Одним из наиболее распространенных применений оператора `break` с меткой служит выход из вложенных циклов. Так, в следующем примере программы внешний цикл выполняется только один раз:

```

// Применение оператора break для выхода из вложенных циклов
class BreakLoop4 {
    public static void main(String args[]) {
        outer: for(int i=0; i<3; i++) {
            System.out.print("Проход " + i + ": ");
            for(int j=0; j<100; j++) {
                if(j == 10) break outer; // выход из обоих циклов
                System.out.print(j + " ");
            }
            System.out.println("Эта строка не будет выводиться");
        }
        System.out.println("Циклы завершены.");
    }
}

```

Эта программа выводит следующий результат:

Проход 0: 0 1 2 3 4 5 6 7 8 9 Циклы завершены.

Как видите, когда выполняется выход из внутреннего цикла во внешний цикл, это приводит к завершению обоих циклов. Следует иметь в виду, что нельзя выполнить переход к метке, если она не определена для объемлющего блока кода. Так, в следующем примере программа содержит ошибку, и поэтому скомпилировать ее нельзя:

```

// Эта программа содержит ошибку
class BreakErr {
    public static void main(String args[]) {
        one: for(int i=0; i<3; i++) {
            System.out.print("Проход " + i + ": ");
        }

        for(int j=0; j<100; j++) {
            if(j == 10) break one; // ОШИБКА!
            System.out.print(j + " ");
        }
    }
}

```

Блок кода, помеченный меткой `one`, не содержит оператор `break`, и поэтому передача управления этому внешнему блоку невозможна.

Применение оператора `continue`

Иногда требуется, чтобы повторение цикла осуществлялось с более раннего оператора в его теле. Это означает, что на данном конкретном шаге может воз-

никнуть потребность продолжить выполнение цикла без выполнения остального кода в его теле. По существу, это означает переход в теле цикла к его окончанию. Для выполнения этого действия служит оператор `continue`. В циклах `while` и `do-while` оператор `continue` вызывает передачу управления непосредственно условному выражению, управляющему циклом. В цикле `for` управление передается в начале итерационной части цикла `for`, а затем условному выражению. Во всех трех видах циклов любой промежуточный код пропускается.

Ниже приведен пример программы, в которой оператор `continue` используется для вывода двух чисел в каждой строке.

```
// Продемонстрировать применение оператора continue
class Continue {
    public static void main(String args[]) {
        for(int i=0; i<10; i++) {
            System.out.print(i + " ");
            if (i%2 == 0) continue;
            System.out.println("");
        }
    }
}
```

В данном примере кода оператор `%` служит для проверки четности значения переменной `i`. Если это четное значение, то выполнение цикла продолжается без перехода к новой строке. Эта программа выводит следующий результат:

```
0 1
2 3
4 5
6 7
8 9
```

Как и оператор `break`, оператор `continue` может содержать метку объемлющего цикла, который нужно продолжить. Ниже приведен пример программы, в которой оператор `continue` применяется для вывода треугольной таблицы умножения чисел от 0 до 9.

```
// Применение оператора continue с меткой
class ContinueLabel {
    public static void main(String args[]) {
        outer: for (int i=0; i<10; i++) {
            for(int j=0; j<10; j++) {
                if(j > i) {
                    System.out.println();
                    continue outer;
                }
                System.out.print(" " + (i * j));
            }
            System.out.println();
        }
    }
}
```

В данном примере оператор `continue` прерывает цикл подсчета значений переменной `j` и продолжает его со следующего шага цикла, в котором подсчитываются значения переменной `i`. Эта программа выводит следующий результат:

```
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81
```

Удачные примеры применения оператора `continue` встречаются редко. Это объясняется, в частности, тем, что в Java предлагается широкий выбор операторов цикла, удовлетворяющих требованиям большинства приложений. Но в тех случаях, когда требуется более раннее начало нового шага цикла, оператор `continue` предоставляет структурированный способ решения данной задачи.

Оператор `return`

Последним из рассматриваемых здесь управляющих операторов является `return`. Он служит для выполнения явного выхода из метода, т.е. снова передает управление объекту, который вызвал данный метод. Как таковой, этот оператор относится к операторам перехода. Полное описание оператора `return` придется отложить до рассмотрения методов в главе 6, тем не менее, здесь уместно представить хотя бы вкратце его особенности.

Оператор `return` можно использовать в любом месте метода для возврата управления тому объекту, который вызвал данный метод. Следовательно, оператор `return` немедленно прекращает выполнение метода, в теле которого он находится, что и демонстрирует приведенный ниже пример. В данном примере выполнение оператора `return` приводит к возврату управления исполняющей системе Java, поскольку именно она вызывает метод `main()`.

```
// Продемонстрировать применение оператора return
class Return {
    public static void main(String args[]) {
        boolean t = true;

        System.out.println("До возврата.");

        if(t) return; // возврат в вызывающий код

        System.out.println("Этот оператор выполняться не будет.");
    }
}
```

Эта программа выводит следующий результат:

До возврата.

Как видите, заключительный вызов метода `println()` не выполняется. Сразу после выполнения оператора `return` управление возвращается вызывающему объекту.

И наконец, в приведенном выше примере программы наличие оператора `if(t)` обязательно. Без него компилятор Java сигнализировал бы об ошибке типа `"unreachable code"` (недостижимый код), поскольку он выяснил бы, что последний вызов метода `println()` вообще не будет выполняться. Во избежание подобной ошибки в данном примере пришлось прибегнуть к помощи оператора `if`, чтобы ввести компилятор в заблуждение.

Класс — это элемент, составляющий основу Java. А поскольку класс определяет форму и сущность объекта, то он является той логической конструкцией, на основе которой построен весь язык Java. Как таковой, класс образует основу объектно-ориентированного программирования на Java. Любое понятие, которое требуется реализовать в программе на Java, должно быть инкапсулировано в пределах класса.

В связи с тем, что класс играет такую основополагающую роль в Java, эта и несколько последующих глав посвящены исключительно классам. В этой главе представлены основные элементы класса и поясняется, как пользоваться классом для создания объектов. Здесь также представлены методы, конструкторы и ключевое слово `this`.

Основы классов

Классы употреблялись в примерах программ едва ли не с самого начала книги. Но в приведенных до сих пор примерах демонстрировалась только самая примитивная форма класса. Классы в этих примерах служили только в качестве контейнеров для метода `main()`, который предназначался главным образом для ознакомления с основами синтаксиса Java. Как станет ясно в дальнейшем, классы предоставляют значительно больше возможностей, чем те, которые использовались в рассматривавшихся до сих пор примерах.

Вероятно, наиболее важная особенность класса состоит в том, что он определяет новый тип данных. Как только этот новый тип данных будет определен, им можно воспользоваться для создания объектов данного типа. Таким образом, класс — это *шаблон* для создания объекта, а объект — это *экземпляр* класса. А поскольку объект является экземпляром класса, то понятия *объект* и *экземпляр* употребляются как синонимы.

Общая форма класса

При определении класса объявляется его конкретная форма и сущность. Для этого указываются данные, которые он содержит, а также код, воздействующий на эти данные. И хотя очень простые классы могут содержать только код или только данные, большинство классов, применяемых в реальных программах, со-

держит оба компонента. Как будет показано далее, код класса определяет интерфейс с его данными.

Для объявления класса служит ключевое слово `class`. Упомянувшиеся до сих пор классы на самом деле представляли собой очень ограниченные примеры полной формы их объявления. Классы могут быть (и обычно являются) значительно более сложными. Упрощенная общая форма определения класса имеет следующий вид:

```
class имя_класса {
    тип переменная_экземпляра1;
    тип переменная_экземпляра2;
    // ...
    тип переменная_экземпляраN;
    тип имя_метода1(список_параметров) {
        // тело метода
    }
    тип имя_метода2(список_параметров) {
        // тело метода
    }
    // ...
    тип имя_методаN(список_параметров) {
        // тело метода
    }
}
```

Данные, или переменные, определенные в классе, называются *переменными экземпляра*. Код содержится в теле *методов*. Вместе с переменными экземпляра методы, определенные в классе, называются *членами* класса. В большинстве классов действия над переменными экземпляра и доступ к ним осуществляют методы, определенные в этом классе. Таким образом, именно методы, как правило, определяют порядок использования данных класса.

Как упоминалось выше, переменные, определенные в классе, называются переменными экземпляра, поскольку каждый экземпляр класса (т.е. каждый объект класса) содержит собственные копии этих переменных. Таким образом, данные одного объекта отделены и отличаются от данных другого объекта. Мы еще вернемся к рассмотрению этого понятия, но оно настолько важное, что его следовало ввести как можно раньше и пояснить хотя бы кратко.

Все методы имеют ту же общую форму, что и метод `main()`, который употреблялся в рассматривавшихся до сих пор примерах. Но большинство методов редко объявляются как `static` или `public`. Обратите внимание на то, что в общей форме класса отсутствует определение метода `main()`. Классы Java могут и не содержать этот метод. Его обязательно указывать только в тех случаях, когда данный класс служит отправной точкой для выполнения программы. Более того, в некоторых видах приложений Java вроде апплетов метод `main()` вообще не требуется.

Простой класс

Итак, начнем рассмотрение классов с простого примера. Ниже приведен код класса `Box` (Параллелепипед), который определяет три переменные экземпляра: `width` (ширина), `height` (высота) и `depth` (глубина). В настоящий момент класс `Box` не содержит никаких методов (но в дальнейшем они будут введены в него).

```
class Box {  
    double width;  
    double height;  
    double depth;  
}
```

Как пояснялось выше, класс определяет новый тип данных. В данном случае новый тип данных называется `Box`. Это имя будет использоваться для объявления объектов типа `Box`. Не следует забывать, что объявление `class` создает только шаблон, но не конкретный объект. Таким образом, приведенный выше код не приводит к появлению каких-нибудь объектов типа `Box`.

Чтобы действительно создать объект класса `Box`, нужно воспользоваться оператором наподобие следующего:

```
Box mybox = new Box(); // создать объект mybox класса Box
```

После выполнения этого оператора объект `mybox` станет экземпляром класса `Box`. Таким образом, он обретет “физическое” существование. Оставим пока что без внимания особенности выполнения этого оператора.

Напомним, что всякий раз, когда получается экземпляр класса, создается объект, который содержит собственную копию каждой переменной экземпляра, определенной в данном классе. Таким образом, каждый объект класса `Box` будет содержать собственные копии переменных экземпляра `width`, `height` и `depth`. Для доступа к этим переменным служит *операция-точка* (`.`). Эта операция связывает имя объекта с именем переменной экземпляра. Например, чтобы присвоить переменной `width` экземпляра `mybox` значение 100, нужно выполнить следующий оператор:

```
mybox.width = 100;
```

Этот оператор предписывает компилятору, что копии переменной `width`, хранящейся в объекте `mybox`, требуется присвоить значение 100. В общем, операция-точка служит для доступа как к переменным экземпляра, так и к методам в пределах объекта. Следует также иметь в виду, что в формальной спецификации языка Java точка (`.`) относится к категории разделителей, несмотря на то, что она обозначает операцию-точку. Но поскольку термин *операция-точка* широко распространен, то он употребляется и в этой книге.

Ниже приведен полноценный пример программы, в которой используется класс `Box`.

```
/* Программа, использующая класс Box  
  
Присвоить исходному файлу имя BoxDemo.java  
*/  
class Box {  
    double width;  
    double height;  
    double depth;  
}  
  
// В этом классе объявляется объект типа Box  
class BoxDemo {  
    public static void main(String args[]) {  
        Box mybox = new Box();
```

```

double vol;
    // присвоить значение переменным экземпляра mybox
mybox.width = 10;
mybox.height = 20;
mybox.depth = 15;

    // рассчитать объем параллелепипеда
vol = mybox.width * mybox.height * mybox.depth;
System.out.println("Объем равен " + vol);
}
}

```

Файлу этой программы следует присвоить имя `BoxDemo.java`, поскольку метод `main()` определен в классе `BoxDemo`, а не `Box`. После компиляции этой программы вы обнаружите, что в конечном итоге были созданы два файла с расширением `.class`: один — для класса `Box`, другой — для класса `BoxDemo`. Компилятор Java автоматически размещает каждый класс в отдельном файле с расширением `.class`. На самом деле классы `Box` и `BoxDemo` совсем не обязательно должны быть объявлены в одном и том же исходном файле. Каждый из этих классов можно было бы разместить в отдельном файле под именем `Box.java` и `BoxDemo.java` соответственно. Чтобы запустить эту программу на выполнение, следует выполнить файл `BoxDemo.class`. В итоге будет получен следующий результат:

Объем равен 3000.0

Как пояснялось ранее, каждый объект содержит собственные копии переменных экземпляра. Это означает, что при наличии двух объектов класса `Box` каждый из них будет содержать собственные копии переменных `depth`, `width` и `height`. Следует, однако, иметь в виду, что изменения в переменных экземпляра одного объекта не влияют на переменные экземпляра другого. Например, в следующей программе объявлены два объекта класса `Box`:

```

// В этой программе объявляются два объекта класса Box

class Box {
    double width;
    double height;
    double depth;
}

class BoxDemo2 {
    public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        // присвоить значения переменным экземплярам mybox1
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;

        /* присвоить другие значения переменным
           экземплярам mybox2 */
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;
    }
}

```

```
// рассчитать объем первого параллелепипеда
vol = mybox1.width * mybox1.height * mybox1.depth;
System.out.println("Объем равен " + vol);

// рассчитать объем второго параллелепипеда
vol = mybox2.width * mybox2.height * mybox2.depth;
System.out.println("Объем равен " + vol);
}
}
```

Эта программа выводит следующий результат:

```
Объем равен 3000.0
Объем равен 162.0
```

Как видите, данные из объекта `mybox1` полностью изолированы от данных, содержащихся в объекте `mybox2`.

Объявление объектов

Как отмечалось ранее, при создании класса создается новый тип данных, который можно использовать для объявления объектов данного типа. Но создание объектов класса представляет собой двухэтапный процесс. Сначала следует объявить переменную типа класса. Эта переменная не определяет объект. Она является лишь переменной, которая может *ссылаться* на объект. Затем нужно получить конкретную, физическую копию объекта и присвоить ее этой переменной. Это можно сделать с помощью оператора `new`. Этот оператор динамически (т.е. во время выполнения) резервирует память для объекта и возвращает ссылку на него. В общих чертах эта ссылка представляет собой адрес объекта в памяти, зарезервированной оператором `new`. Затем эта ссылка сохраняется в переменной. Таким образом, оперативная память должна динамически выделяться для объектов всех классов в Java. Рассмотрим эту процедуру более подробно.

В приведенном ранее примере программы строка кода, аналогичная приведенной ниже, служит для объявления объекта типа `Box`.

```
Box mybox = new Box();
```

В этой строке кода объединяются оба этапа только что описанного процесса. Чтобы каждый этап данного процесса стал более очевидным, приведенную выше строку кода можно переписать следующим образом:

```
Box mybox; // объявить ссылку на объект
mybox = new Box(); // выделить память для объекта Box
```

В первой строке приведенного выше фрагмента кода переменная `mybox` объявляется как ссылка на объект типа `Box`. В данный момент переменная `mybox` пока еще не ссылается на конкретный объект. В следующей строке кода выделяется память для конкретного объекта, а переменной `mybox` присваивается ссылка на этот объект. После выполнения второй строки кода переменную `mybox` можно использовать так, как если бы она была объектом типа `Box`. Но в действительности переменная `mybox` просто содержит адрес памяти конкретного объекта типа `Box`. Результат выполнения этих двух строк кода показан на рис. 6.1.

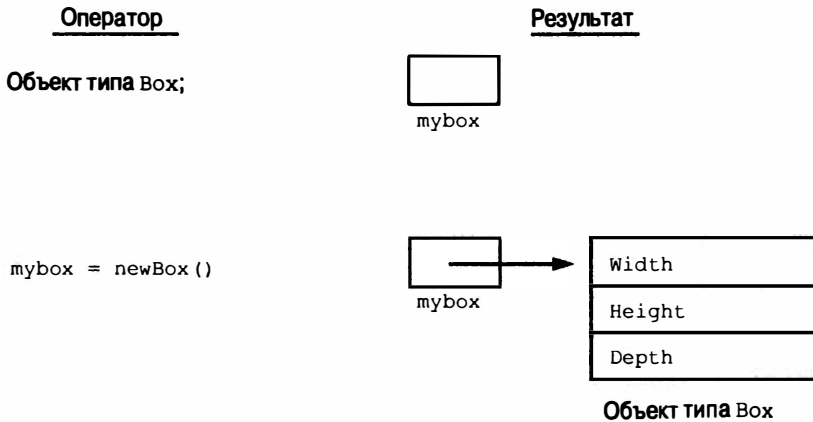


Рис 6.1. Объявление объекта типа Box

На заметку! Тем, у кого имеется опыт программирования на C/C++, вероятно, известно, что ссылки на объекты подобны указателям. В целом это верно. Ссылка на объект похожа на указатель в памяти. Основное их отличие и в то же время главное средство, обеспечивающее безопасность программ на Java, состоит в том, что ссылками нельзя манипулировать, как настоящими указателями. В частности, ссылка на объект не может указывать на произвольную ячейку памяти, и ею нельзя манипулировать как целочисленным значением.

Подробное рассмотрение оператора `new`

Как отмечалось ранее, оператор `new` динамически выделяет оперативную память для объекта. Общая форма этого оператора имеет следующий вид:

```
переменная_класса = new имя_класса ();
```

где *переменная_класса* обозначает переменную создаваемого класса, а *имя_класса* — конкретное имя класса, экземпляр которого получается. Имя класса, за которым следуют круглые скобки, обозначает *конструктор* данного класса. Конструктор определяет действия, выполняемые при создании объекта класса. Конструкторы являются важной частью всех классов и обладают множеством важных свойств. В большинстве классов, используемых в реальных программах, явно объявляются свои конструкторы в пределах определения класса. Но если ни один из явных конструкторов не указан, то в Java будет автоматически предоставлен конструктор по умолчанию. Именно это и происходит при создании объекта класса `Box`. В рассматриваемых здесь примерах будет пока еще употребляться конструктор по умолчанию, а в дальнейшем будет показано, как определять собственные конструкторы.

В связи с изложенным выше может возникнуть вопрос: почему оператор `new` не требуется для таких типов данных, как целочисленные или символьные? Это объясняется тем, что примитивные типы данных в Java реализованы не в виде объектов, а как "обычные" переменные. И сделано это ради повышения эффективности. Как будет показано далее, объекты обладают многими свойствами и сред-

ствами, которые требуют, чтобы они трактовались в Java иначе, чем примитивные типы. Отсутствие таких же издержек, как и на обращение с объектами, позволяет эффективнее реализовать примитивные типы данных. В дальнейшем будут представлены объектные версии примитивных типов, которые могут пригодиться в тех случаях, когда требуются полноценные объекты этих типов.

Однако оператор `new` выделяет оперативную память для объекта во время выполнения. Преимущество такого подхода состоит в том, что в программе можно создать ровно столько объектов, сколько требуется во время ее выполнения. Но поскольку объем оперативной памяти ограничен, возможны случаи, когда оператор `new` не в состоянии выделить память для объекта из-за ее нехватки. В таком случае возникает исключение времени выполнения. (Более подробно обработка исключений рассматривается в главе 10.) В примерах программ, приведенных в этой книге, недостатка в объеме памяти не возникает, но в реальных программах такую возможность придется учитывать.

Еще раз рассмотрим отличие класса от объекта. Класс создает новый тип данных, который можно использовать для создания объектов. Это означает, что класс создает логический каркас, определяющий взаимосвязь между его членами. При объявлении объекта класса создается экземпляр этого класса. Таким образом, класс — это логическая конструкция, а объект имеет физическую сущность, т.е. он занимает конкретную область оперативной памяти. Об этом отличии важно помнить.

Присваивание переменным ссылок на объекты

При присваивании переменные ссылок на объекты действуют иначе, чем можно было бы предположить. Например, какие действия выполняет приведенный ниже фрагмент кода?

```
Box b1 = new Box();  
Box b2 = b1;
```

На первый взгляд, переменной `b2` присваивается ссылка на копию объекта, на которую ссылается переменная `b1`. Таким образом, может показаться, что переменные `b1` и `b2` ссылаются на совершенно разные объекты, но это совсем не так. После выполнения данного фрагмента кода обе переменные, `b1` и `b2`, будут ссылаться на *один и тот же* объект. Присваивание переменной `b1` значения переменной `b2` не привело к выделению области памяти или копированию какой-нибудь части исходного объекта. Такое присваивание приводит лишь к тому, что переменная `b2` ссылается на тот же объект, что и переменная `b1`. Таким образом, любые изменения, внесенные в объекте по ссылке в переменной `b2`, окажут влияние на объект, на который ссылается переменная `b1`, поскольку это один и тот же объект. Это положение наглядно иллюстрирует рис. 6.2.

Помимо того, что переменные `b1` и `b2` ссылаются на один и тот же объект, они не связаны никак иначе. Так, в приведенном ниже примере кода присваивание пустого значения переменной `b1` просто *разрывает связь* переменной `b1` с исходным объектом, не оказывая никакого влияния на сам объект или переменную `b2`. В дан-

ном примере в переменной `b1` устанавливается пустое значение `null`, но переменная `b2` по-прежнему указывает на исходный объект.

```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

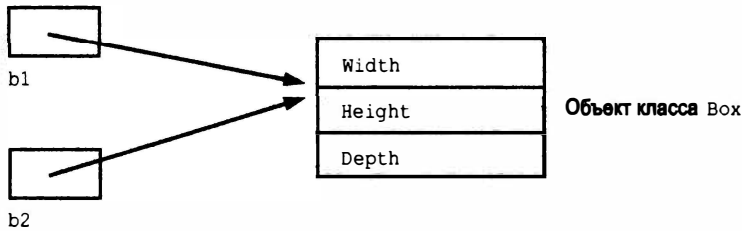


Рис. 6.2. Присваивание переменным ссылок на один и тот же объект

Помните! Присваивание одной переменной ссылки на объект другой приводит к созданию только копии ссылки на объект, а не копии самого объекта.

Введение в методы

Как упоминалось в начале этой главы, классы обычно состоят из двух компонентов: переменных экземпляра и методов. Тема методов довольно обширна, поскольку в Java они получают немалые возможности и удобства. По существу, многие из последующих глав данной книги посвящены методам. Но для того чтобы вводить методы в классы, нужно знать хотя бы самые основные их особенности. Общая форма объявления метода выглядит следующим образом:

```
тип имя(список_параметров) {
    // тело метода
}
```

где `тип` обозначает конкретный тип данных, возвращаемых методом. Он может быть любым допустимым типом данных, в том числе и типом созданного класса. Если метод не возвращает значение, то его возвращаемым типом должен быть `void`. Для указания имени метода служит идентификатор `имя`. Это может быть любой допустимый идентификатор, кроме тех, которые уже используются другими элементами кода в текущей области действия. А `список_параметров` обозначает последовательность пар “тип-идентификатор”, разделенных запятыми. По существу, параметры — это переменные, которые принимают значения аргументов, передаваемых методу во время его вызова. Если у метода отсутствуют параметры, то `список_параметров` оказывается пустым.

Методы, возвращаемый тип которых отличается от `void`, возвращают значение вызывающей части программы в соответствии со следующей формой оператора `return`:

```
return значение;
```

где параметр *значение* обозначает возвращаемое методом значение. В последующих разделах будет показано, каким образом создаются различные типы методов, в том числе принимающие параметры и возвращающие значения.

Ввод метода в класс Box

Было бы очень удобно создать класс, содержащий только данные, но в реальных программах подобное встречается крайне редко. В большинстве случаев для доступа к переменным экземпляра, определенным в классе, приходится пользоваться методами. По существу, методы определяют интерфейс для большинства классов. Это позволяет тому, кто реализует класс, скрывать конкретное расположение внутренних структур данных за более понятными абстракциями методов. Кроме методов, обеспечивающих доступ к данным, можно определить и методы, применяемые в самом классе.

Итак, приступим к вводу метода в класс Box. Просматривая предыдущие примеры программ, нетрудно прийти к выводу, что класс Box мог бы лучше справиться с расчетом объема параллелепипеда, чем класс BoxDemo. В конце концов, было бы логично, если бы такой расчет выполнялся в классе Box, поскольку объем параллелепипеда зависит от его размеров. Для этого в класс Box следует ввести метод, как показано ниже.

// В этой программе применяется метод, введенный в класс **Box**

```
class Box {
    double width;
    double height;
    double depth;

    // вывести объем параллелепипеда
    void volume() {
        System.out.print("Объем равен ");
        System.out.println(width * height * depth);
    }
}

class BoxDemo3 {
    public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();

        // присвоить значение переменным экземпляра mybox1
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;

        /* присвоить другие значения переменным экземпляра mybox2
           экземпляра mybox2 */
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;

        // вывести объем первого параллелепипеда
        mybox1.volume();

        // вывести объем второго параллелепипеда
```

```

        mybox2.volume();
    }
}

```

Эта программа выводит следующий результат, совпадающий с результатом в предыдущей версии:

```

Объем равен 3000.0
Объем равен 162.0

```

Внимательно рассмотрите две следующие строки кода:

```

mybox1.volume();
mybox2.volume();

```

В первой строке кода вызывается метод `volume()` для объекта `mybox1`. Следовательно, метод `volume()` вызывается по отношению к объекту `mybox1`, для чего было указано имя объекта, а вслед за ним — операция-точка. Таким образом, в результате вызова метода `mybox1.volume()` выводится объем параллелепипеда, определяемого объектом `mybox1`, а в результате вызова метода `mybox2.volume()` — объем параллелепипеда, определяемого объектом `mybox2`. При каждом вызове метод `volume()` выводит объем указанного параллелепипеда.

Пояснения, приведенные в следующих абзацах, облегчат понимание принципа вызова методов. При вызове метода `mybox1.volume()` исполняющая система Java передает управление коду, определенному в теле метода `volume()`. По окончании выполнения всех операторов в теле метода управление возвращается вызывающей части программы и далее ее выполнение продолжается со строки кода, следующей за вызовом метода. В самом общем смысле метод — это способ реализации подпрограмм в Java.

В методе `volume()` следует обратить внимание на еще одну очень важную особенность: ссылка на переменные экземпляра `width`, `height` и `depth` делается непосредственно без указания перед ними имени объекта или операции-точки. Когда в методе используется переменная экземпляра, определенная в его же классе, это делается непосредственно, без указания явной ссылки на объект и применения операции-точки. Это становится понятным, если немного подумать. Метод всегда вызывается по отношению к какому-то объекту его класса. Как только этот вызов сделан, объект известен. Таким образом, в теле метода вторичное указание объекта совершенно излишне. Это означает, что переменные экземпляра `width`, `height` и `depth` неявно ссылаются на копии этих переменных, хранящиеся в объекте, который вызывает метод `volume()`.

Подведем краткие итоги. Когда доступ к переменной экземпляра осуществляется из кода, *не* входящего в класс, где определена переменная экземпляра, следует непременно указать объект с помощью операции-точки. Но когда такой доступ осуществляется из кода, входящего в класс, где определена переменная экземпляра, ссылка на переменную может делаться непосредственно. Эти же правила относятся и к методам.

Возврат значений

Несмотря на то что реализация метода `volume()` переносит расчет объема параллелепипеда в пределы класса `Box`, которому принадлежит этот метод, такой

способ расчета все же не является наилучшим. Например, что делать, если в другой части программы требуется знать объем параллелепипеда без его вывода? Более рациональный способ реализации метода `volume()` состоит в том, чтобы рассчитать объем параллелепипеда и вернуть результат вызывающему коду. Именно эта задача решается в приведенном ниже примере усовершенствованной версии предыдущей программы.

// Теперь метод `volume()` возвращает объем параллелепипеда

```
class Box {
    double width;
    double height;
    double depth;

    // рассчитать и вернуть объем
    double volume() {
        return width * height * depth;
    }
}

class BoxDemo4 {
    public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        // присвоить значения переменным экземпляра mybox1
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;

        /* присвоить другие значения переменным экземпляра mybox2 */
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;

        // получить объем первого параллелепипеда
        vol = mybox1.volume();
        System.out.println("Объем равен " + vol);

        // получить объем второго параллелепипеда
        vol = mybox2.volume();
        System.out.println("Объем равен " + vol);
    }
}
```

Как видите, вызов метода `volume()` выполняется в правой части операции присваивания. В левой части этой операции находится переменная, в данном случае `vol`, которая будет принимать значение, возвращаемое методом `volume()`. Таким образом, после выполнения операции

```
vol = mybox1.volume();
```

метод `mybox1.volume()` возвращает значение 3000, и это значение рассчитанного объема сохраняется в переменной `vol`. При обращении с возвращаемыми значениями следует принимать во внимание два важных обстоятельства.

- Тип данных, возвращаемых методом, должен быть совместим с возвращаемым типом, указанным в методе. Так, если какой-нибудь метод должен воз-

вращать логический тип `boolean`, то вернуть из него целочисленное значение нельзя.

- Переменная, принимающая возвращаемое методом значение (например, `vol`), также должна быть совместима с возвращаемым типом, указанным для метода.

И еще одна особенность: приведенную выше программу можно было бы написать и в более эффективной форме, поскольку переменная `vol` в действительности совсем не нужна. Метод `volume()` можно было бы вызвать непосредственно в операторе с вызовом метода `println()`, как в следующей строке кода:

```
System.out.println("Объем равен" + mybox1.volume());
```

В этом случае метод `mybox1.volume()` будет вызываться автоматически при выполнении оператора с вызовом метода `println()`, а возвращаемое им значение будет передаваться методу `println()`.

Ввод метода, принимающего параметры

Хотя некоторые методы не нуждаются в параметрах, большинство из них все же требует их передачи. Параметры позволяют обобщить метод. Это означает, что метод с параметрами может обрабатывать различные данные и/или применяться в самых разных случаях. В качестве иллюстрации рассмотрим очень простой пример. Ниже приведен метод, возвращающий квадрат числа 10.

```
int square()
{
    return 10 * 10;
}
```

Хотя этот метод действительно возвращает значение 10^2 , его применение очень ограничено. Но если метод `square()` изменить таким образом, чтобы он принимал параметр, как показано ниже, то пользы от него будет много больше.

```
int square(int i)
{
    return i * i;
}
```

Теперь метод `square()` будет возвращать квадрат любого значения, с которым он вызывается. Таким образом, метод `square()` становится теперь методом общего назначения, способным вычислять квадрат любого целочисленного значения, а не только числа 10. Ниже приведены некоторые примеры применения метода `square()`.

```
int x, y;
x = square(5); // x равно 25
x = square(9); // x равно 81
y = 2;
x = square(y); // x равно 4
```

При первом вызове метода `square()` значение 5 передается параметру `i`. При втором вызове этого метода параметр `i` принимает значение 9, а при третьем вызове метода `square()` ему передается значение переменной `y`, которое в данном

примере равно 2. Как следует из этих примеров, метод `square()` способен возвращать квадрат любых передаваемых ему числовых значений.

В отношении методов важно различать два термина: *параметр* и *аргумент*. *Параметр* — это определенная в методе переменная, которая принимает заданное значение при вызове метода. Например, в методе `square()` параметром является `i`. *Аргумент* — это значение, передаваемое методу при его вызове. Например, при вызове `square(100)` в качестве аргумента этому методу передается значение 100. Это значение получает параметр `i` в теле метода `square()`.

Методом с параметрами можно воспользоваться для усовершенствования класса `Box`. В предшествующих примерах размеры каждого параллелепипеда нужно было устанавливать отдельно, используя последовательность операторов вроде следующей:

```
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
```

Несмотря на то что приведенный выше код вполне работоспособен, он не очень удобен по двум причинам. Во-первых, он громоздок и чреват ошибками. Так, можно вполне забыть определить один из размеров параллелепипеда. И во-вторых, в правильно написанных программах на Java доступ к переменным экземпляра должен осуществляться только через методы, определенные в их классе. В дальнейшем поведение метода можно изменить, но нельзя изменить поведение предоставляемой переменной экземпляра.

Следовательно, более рациональный способ установки размеров параллелепипеда состоит в создании метода, который принимает размеры параллелепипеда в виде своих параметров и соответствующим образом устанавливает значение каждой переменной экземпляра. Именно этот принцип и был реализован в приведенном ниже примере программы.

// В этой программе применяется метод с параметрами

```
class Box {
    double width;
    double height;
    double depth;

    // рассчитать и вернуть объем
    double volume() {
        return width * height * depth;
    }
    // установить размеры параллелепипеда
    void setDim(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
}

class BoxDemo5 {
    public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;
```

```
// инициализировать каждый экземпляр класса Вох
mybox1.setDim(10, 20, 15);
mybox2.setDim(3, 6, 9);

// получить объем первого параллелепипеда
vol = mybox1.volume();
System.out.println("Объем равен " + vol);

// получить объем второго параллелепипеда
vol = mybox2.volume();
System.out.println("Объем равен " + vol);
}
}
```

Как видите, метод `setDim()` использован для установки размеров каждого параллелепипеда. Например, при выполнении следующей строки кода:

```
mybox1.setDim(10, 20, 15);
```

значение **10** копируется в параметр *w*, значение **20** — в параметр *h*, а значение **15** — в параметр *d*. Затем в теле метода `setDim()` значения параметров *w*, *h* и *d* присваиваются переменным `width`, `height` и `depth` соответственно.

Понятия, представленные в этих разделах, вероятно, знакомы многим читателям. Но если вы еще не знакомы с такими понятиями, как вызовы методов, аргументы и параметры, немного поэкспериментируйте с ними, прежде чем продолжить изучение материала, изложенного в последующих разделах. Понятия вызова метода, параметров и возвращаемых значений являются основополагающими в программировании на языке Java.

Конструкторы

Инициализация всех переменных класса при каждом создании его экземпляра может оказаться утомительным процессом. Даже при добавлении служебных функций вроде метода `setDim()` было бы проще и удобнее, если бы все действия по установке значений переменных выполнялись при первом создании объекта. В связи с тем что потребность в инициализации возникает очень часто, объектам в Java разрешается выполнять собственную инициализацию при их создании. И эта автоматическая инициализация осуществляется с помощью конструктора.

Конструктор инициализирует объект непосредственно во время его создания. Его имя совпадает с именем класса, в котором он находится, а синтаксис аналогичен синтаксису метода. Как только конструктор определен, он автоматически вызывается при создании объекта перед окончанием выполнения оператора `new`. Конструкторы выглядят не совсем привычно, поскольку они не имеют возвращаемого типа — даже типа `void`. Это объясняется тем, что неявно заданным возвращаемым типом конструктора класса является тип самого класса. Именно конструктор инициализирует внутреннее состояние объекта таким образом, чтобы код, создающий экземпляр, с самого начала содержал полностью инициализированный, пригодный к употреблению объект.

Пример класса `Box` можно переделать, чтобы значения размеров параллелепипеда присваивались при конструировании объекта. Для этого придется заменить метод `setDim()` конструктором. Сначала в приведенной ниже новой версии про-

граммы определяется простой конструктор, устанавливающий одинаковые значения размеров всех параллелепипедов.

```
/* В данном примере программы для инициализации размеров
   параллелепипеда в классе Box применяется конструктор
*/
class Box {
    double width;
    double height;
    double depth;

    // Это конструктор класса Box
    Box() {
        System.out.println("Конструирование объекта Box");
        width = 10;
        height = 10;
        depth = 10;
    }

    // рассчитать и вернуть объем
    double volume() {
        return width * height * depth;
    }
}

class BoxDemo6 {
    public static void main(String args[]) {
        // объявить, выделить память и инициализировать объекты типа Box
        Box mybox1 = new Box();
        Box mybox2 = new Box();

        double vol;

        // получить объем первого параллелепипеда
        vol = mybox1.volume();
        System.out.println("Объем равен " + vol);

        // получить объем второго параллелепипеда
        vol = mybox2.volume();
        System.out.println("Объем равен " + vol);
    }
}
```

Эта программа выводит следующий результат:

```
Конструирование объекта Box
Конструирование объекта Box
Объем равен 1000.0
Объем равен 1000.0
```

Как видите, оба объекта, `mybox1` и `mybox2`, инициализированы конструктором `Box()` при их создании. Конструктор присваивает всем параллелепипедам одинаковые размеры $10 \times 10 \times 10$, поэтому объекты `mybox1` и `mybox2` будут иметь одинаковый объем. Вызов метода `println()` в конструкторе `Box()` делается исключительно ради иллюстрации. Но большинство конструкторов не выводят никаких данных, а лишь выполняют инициализацию объекта.

Прежде чем продолжить дальше, вернемся к рассмотрению оператора `new`. Как вам, должно быть уже известно, при выделении памяти для объекта используется следующая общая форма:

```
переменная_класса = new имя_класса();
```


Теперь вам должно быть ясно, почему после имени класса требуется указывать круглые скобки. В действительности оператор `new` вызывает конструктор класса. Таким образом, в следующей строке кода:

```
Box mybox1 = new Box();
```

оператор `new Box()` вызывает конструктор `Box()`. Если конструктор класса не определен явно, то в Java для класса создается конструктор по умолчанию. Именно поэтому приведенная выше строка кода была вполне работоспособной в предыдущих версиях класса `Box`, где конструктор не был определен. Конструктор по умолчанию инициализирует все переменные экземпляра устанавливаемыми по умолчанию значениями, которые могут быть нулевыми, пустыми (`null`) и логическими (`false`) для числовых, ссылочных и логических (`boolean`) типов соответственно. Зачастую конструктора по умолчанию оказывается достаточно для простых классов, чего, как правило, нельзя сказать о более сложных классах. Как только в классе будет определен собственный конструктор, конструктор по умолчанию больше не используется.

Параметризованные конструкторы

В предыдущем примере конструктор `Box()` инициализирует объект класса `Box`, но пользы от такого конструктора немного, так как все параллелепипеды получают одинаковые размеры. Следовательно, требуется найти способ создавать объекты класса `Box` с разными размерами. В качестве простейшего решения достаточно ввести в конструктор параметры. Нетрудно догадаться, такое решение делает конструктор более полезным. Например, в приведенной ниже версии класса `Box` определяется параметризованный конструктор, устанавливающий размеры параллелепипеда по значениям параметров конструктора. Обратите особое внимание на порядок создания объектов класса `Box`.

```
/* В данном примере программы для инициализации размеров
   параллелепипеда в классе Box применяется параметризованный
   конструктор
*/
class Box {
    double width;
    double height;
    double depth;

    // Это конструктор класса Box
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // рассчитать и вернуть объем
    double volume() {
        return width * height * depth;
    }
}

class BoxDemo7 {
    public static void main(String args[]) {
        // объявить, выделить память и инициализировать объекты типа Box
```

```
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box(3, 6, 9);

double vol;

// получить объем первого параллелепипеда
vol = mybox1.volume();
System.out.println("Объем равен " + vol);

// получить объем второго параллелепипеда
vol = mybox2.volume();
System.out.println("Объем равен " + vol);
}
}
```

Вывод этой программы имеет следующий вид:

```
Объем равен 3000.0
Объем равен 162.0
```

Как видите, инициализация каждого объекта выполняется в соответствии с заданными значениями параметров его конструктора. Например, в следующей строке кода:

```
Box mybox1 = new Box(10, 20, 15);
```

значения **10**, **20** и **15** передаются конструктору `Box()` при создании объекта с помощью оператора `new`. Таким образом, копии переменных `width`, `height` и `depth` будут содержать значения **10**, **20** и **15** соответственно.

Ключевое слово `this`

Иногда требуется, чтобы метод ссылался на вызвавший его объект. Для этой цели в Java определено ключевое слово `this`. Им можно пользоваться в теле любого метода для ссылки на текущий объект. Это означает, что ключевое слово `this` всегда служит ссылкой на объект, для которого был вызван метод. Ключевое слово `this` можно использовать везде, где допускается ссылка на объект типа текущего класса. Для того чтобы стало понятнее назначение ключевого слова `this`, рассмотрим следующую версию конструктора `Box()`:

```
// Избыточное применение ключевого слова this
Box(double w, double h, double d) {
    this.width = w;
    this.height = h;
    this.depth = d;
}
```

Эта версия конструктора `Box()` действует точно так же, как и предыдущая. Ключевое слово `this` применяется в данном случае избыточно, хотя и верно. В теле конструктора `Box()` ключевое слово `this` будет всегда ссылаться на вызывающий объект. И хотя здесь это совершенно излишне, в других случаях, один из которых рассмотрен в следующем разделе, пользоваться ключевым словом `this` очень удобно.

Соккрытие переменной экземпляра

Как известно, в Java не допускается объявление двух локальных переменных с одним и тем же именем в той же самой или в объемлющей областях действия. Тем не менее допускается существование локальных переменных, включая и формальные параметры методов, имена которых совпадают с именами переменных экземпляра класса. Но когда имя локальной переменной совпадает с именем переменной экземпляра, локальная переменная *скрывает* переменную экземпляра. Именно поэтому именами `width`, `height` и `depth` в классе `Box` не были названы параметры конструктора `Box()`. В противном случае имя `width`, например, обозначало бы формальный параметр, скрывая переменную экземпляра `width`. И хотя было бы проще выбрать разные имена, существует и другой способ выхода из данного положения. Ключевое слово `this` позволяет ссылаться непосредственно на объект, и поэтому его можно применять для разрешения любых конфликтов, которые могут возникать между переменными экземпляра и локальными переменными в пространстве имен. В качестве примера ниже приведена еще одна версия конструктора `Box()`, в которой имена `width`, `height` и `depth` служат для обозначения параметров, а ключевое слово `this` — для обращения к переменным экземпляра по этим же именам.

```
// В этом коде разрешаются конфликты в пространстве имен
Box(double width, double height, double depth) {
    this.width = width;
    this.height = height;
    this.depth = depth;
}
```

Тем не менее иногда подобное применение ключевого слова `this` может приводить к недоразумениям, и некоторые программисты стараются не применять имена локальных переменных и параметров, скрывающие переменные экземпляра. Безусловно, есть программисты, которые придерживаются иного мнения, считая целесообразным использовать одни и те же имена для большей ясности программ, а ключевое слово `this` — для предотвращения сокращения переменных экземпляра. Впрочем, выбор любого из этих двух подходов зависит от личных предпочтений.

Сборка “мусора”

В связи с тем что выделение оперативной памяти для объектов осуществляется динамически с помощью оператора `new`, невольно может возникнуть вопрос: каким образом объекты уничтожаются и как занимаемая ими память освобождается для повторного выделения под другие объекты. В некоторых языках, подобных C++, объекты, динамически размещаемые в оперативной памяти, приходится освобождать из нее вручную с помощью оператора `delete`. А в Java применяется другой подход: освобождение оперативной памяти осуществляется автоматически. Этот процесс называется *сборкой “мусора”* и происходит следующим образом: в отсутствие любых ссылок на объект считается, что этот объект больше не нужен и занимаемую им память можно освободить. В Java не нужно уничтожать объекты

явным образом, как это требуется в C++. Во время выполнения программы сборка “мусора” выполняется только изредка, если вообще требуется. Она не будет выполняться лишь потому, что один или несколько объектов существуют и больше не используются. Более того, в различных реализациях исполняющей системы Java могут применяться разные подходы к сборке “мусора”, но, как правило, об этом можно не беспокоиться при написании программ.

Метод `finalize()`

Иногда уничтожаемый объект должен выполнять некоторое действие. Так, если объект содержит какой-нибудь ресурс, отличающийся от ресурса Java (вроде файлового дескриптора или шрифта), такой ресурс, скорее всего, придется освободить перед уничтожением объекта. Для разрешения подобных ситуаций в Java предоставляется механизм, называемый *полным завершением*. Применяя механизм полного завершения, можно определить конкретные действия, которые будут выполняться непосредственно перед удалением объекта сборщиком “мусора”.

Чтобы ввести в класс средство полного завершения, достаточно определить метод `finalize()`, который вызывается в исполняющей среде Java непосредственно перед удалением объекта данного класса. В теле метода `finalize()` нужно указать те действия, которые должны быть выполнены перед уничтожением объекта. Сборщик “мусора” запускается периодически, проверяя наличие объектов, на которые отсутствуют ссылки как непосредственно из любого рабочего состояния, так и косвенно через другие доступные по ссылке объекты. Непосредственно перед освобождением ресурсов метод `finalize()` вызывается из исполняющей среды Java для удаляемого объекта. Общая форма метода `finalize()` имеет следующий вид:

```
protected void finalize() {  
    // здесь должен следовать код полного завершения  
}
```

В этой синтаксической конструкции ключевое слово `protected` служит модификатором, предотвращающим доступ к методу `finalize()`. Этот и другие модификаторы доступа подробно описываются в главе 7.

Следует иметь в виду, что метод `finalize()` вызывается лишь непосредственно перед сборкой “мусора”. Например, он не вызывается при выходе объекта из области действия. Это означает, что заранее неизвестно, когда будет (и будет ли вообще) выполняться метод `finalize()`. Поэтому в программе должны быть предоставлены другие средства для освобождения используемых объектом системных ресурсов и т.п. Нормальная работа программы не должна зависеть от метода `finalize()`.

На заметку! Тем, у кого имеется опыт программирования на C++, должно быть известно, что в этом языке допускается определять деструктор класса, который вызывается при выходе объекта из области действия. А в Java такая возможность не поддерживается, поэтому применение деструкторов не допускается. По своему действию метод `finalize()` лишь отдаленно напоминает деструктор. Постепенно приобретая опыт программирования на Java, вы сами убедитесь, что благодаря наличию подсистемы сборки “мусора” потребность в функциях деструктора возникает крайне редко.

Класс Stack

Несмотря на то что класс `Box` удобен для демонстрации основных элементов класса, его практическая ценность невелика. Поэтому в завершение этой главы рассмотрим более сложный пример, демонстрирующий истинный потенциал классов. Как пояснялось при изложении основ объектно-ориентированного программирования (ООП) в главе 2, одно из главных преимуществ ООП дает инкапсуляция данных и кода, который манипулирует этими данными. А в этой главе пояснялось, что в Java класс реализует механизм инкапсуляции. Вместе с классом создается новый тип данных, который определяет характер обрабатываемых данных, а также используемые для этой цели процедуры. Далее методы задают согласованный и управляемый интерфейс с данными класса. Таким образом, классом можно пользоваться через его методы, не обращая внимания на особенности его реализации или порядок управления данными в классе. В каком-то смысле класс действует подобно механизму данных. Для того чтобы привести такой механизм в действие с помощью его элементов управления, совсем не обязательно знать его внутреннее устройство. А поскольку подробности реализации этого механизма скрыты, то его внутреннее устройство можно изменить по мере необходимости. Изменения можно вносить во внутреннее устройство класса, не вызывая побочных эффектов за его пределами, при условии, что класс используется в прикладном коде через его методы.

Для практического применения приведенных выше теоретических положений рассмотрим организацию стека как один из типичных примеров инкапсуляции. Данные хранятся в стеке по принципу “первым пришел, последним обслужен”. В этом отношении стек подобен стопке тарелок на столе: тарелка, поставленная на стол первой, будет использована последней. Для управления стеком служат две операции, традиционно называемые *размещением* (в стеке) и *извлечением* (из стека). Для того чтобы расположить элемент на вершине стека, выполняется операция размещения. А для того чтобы изъять элемент из стека, выполняется операция извлечения. Как будет показано ниже, инкапсуляция всего механизма действия стека не представляет никакой сложности.

Ниже приведен класс `Stack`, реализующий стек емкостью до десяти целочисленных значений.

```
// В этом классе определяется целочисленный стек, в котором
// можно хранить до 10 целочисленных значений
class Stack {
    int stck[] = new int[10];
    int tos;

    // инициализировать вершину стека
    Stack() {
        tos = -1;
    }

    // разместить элемент в стеке
    void push(int item) {
        if(tos==9)
            System.out.println("Стек заполнен.");
        else
```

```

        stck[++tos] = item;
    }

    // извлечь элемент из стека
    int pop() {
        if(tos < 0) {
            System.out.println("Стек не загружен.");
            return 0;
        }
        else
            return stck[tos--];
    }
}

```

Как видите, в классе `Stack` определены два элемента данных и три метода. Стек целочисленных значений хранится в массиве `stck`. Этот массив индексируется по переменной `tos`, которая всегда содержит индекс вершины стека. В конструкторе `Stack()` переменная `tos` инициализируется значением `-1`, обозначающим пустой стек. Метод `push()` размещает элемент в стеке. Чтобы извлечь элемент из стека, следует вызвать метод `pop()`. А поскольку доступ к стеку осуществляется с помощью методов `push()` и `pop()`, то для обращения со стеком на самом деле не имеет никакого значения, что стек хранится в массиве. Стек можно было бы с тем же успехом хранить и в более сложной структуре данных вроде связанного списка, но интерфейс, определяемый методами `push()` и `pop()`, оставался бы без изменения.

Применение класса `Stack` демонстрируется в приведенном ниже примере класса `TestStack`. В этом классе организуются два целочисленных стека, в каждом из которых сначала размещаются, а затем извлекаются некоторые значения.

```

class TestStack {
    public static void main(String args[]) {
        Stack mystack1 = new Stack();
        Stack mystack2 = new Stack();

        // разместить числа в стеке
        for(int i=0; i<10; i++) mystack1.push(i);
        for(int i=10; i<20; i++) mystack2.push(i);

        // извлечь эти числа из стека
        System.out.println("Содержимое стека mystack1:");
        for(int i=0; i<10; i++)
            System.out.println(mystack1.pop());

        System.out.println("Содержимое стека mystack2:");
        for(int i=0; i<10; i++)
            System.out.println(mystack2.pop());
    }
}

```

Эта программа выводит следующий результат:

Содержимое стека mystack1:

```

9
8
7
6
5
4
3

```

```
2
1
0
Содержимое стека mystack2:
19
18
17
16
15
14
13
12
11
10
```

Как видите, содержимое обоих стеков отличается.

И последнее замечание по поводу класса `Stack`. В том виде, в каком он реализован, массив `stck`, содержащий стек, может быть изменен из кода, определенного за пределами класса `Stack`. Это делает класс `Stack` уязвимым к злоупотреблениям и повреждениям. В следующей главе будет показано, как исправить этот недостаток.

Подробное рассмотрение классов и методов

В этой главе будет продолжено более подробное рассмотрение методов и классов, начатое в предыдущей главе. Вначале мы обсудим ряд вопросов, связанных с методами, в том числе перегрузку, передачу параметров и рекурсию. Затем еще раз обратимся к классам и рассмотрим управление доступом, использование ключевого слова `static` и `String` — одного из самых важных классов, встроенных в Java.

Перегрузка методов

В Java разрешается в одном и том же классе определять два или более метода с одним именем, если только объявления их параметров отличаются. В этом случае методы называют *перегружаемыми*, а сам процесс — *перегрузкой методов*. Перегрузка методов является одним из способов поддержки полиморфизма в Java. Тем, кто никогда не программировал на языке, допускающем перегрузку методов, это понятие может сначала показаться странным. Но, как станет ясно в дальнейшем, перегрузка методов — одна из наиболее замечательных и полезных функциональных возможностей Java.

При вызове перегружаемого метода для определения нужного его варианта в Java используется тип и/или количество аргументов метода. Следовательно, перегружаемые методы должны отличаться по типу и/или количеству их параметров. Возвращаемые типы перегружаемых методов могут отличаться, но самого возвращаемого типа явно недостаточно для того, чтобы отличать два разных варианта метода. Когда в исполняющей среде Java встречается вызов перегружаемого метода, в ней просто выполняется тот его вариант, параметры которого соответствуют аргументам, указанным в вызове.

В следующем простом примере демонстрируется перегрузка метода:

```
// Продемонстрировать перегрузку методов
class OverloadDemo {
    void test() {
        System.out.println("Параметры отсутствуют");
    }

    // Перегружаемый метод, проверяющий наличие
    // одного целочисленного параметра
    void test(int a) {
        System.out.println("a: " + a);
    }

    // Перегружаемый метод, проверяющий наличие
```



```

// двух целочисленных параметров
void test(int a, int b) {
    System.out.println("a и b: " + a + " " + b);
}

// Перегружаемый метод, проверяющий наличие
// параметра типа double
double test(double a) {
    System.out.println("double a: " + a);
    return a*a;
}
}
class Overload {
    public static void main(String args[]) {
        OverloadDemo ob = new OverloadDemo();
        double result;

        // вызвать все варианты метода test()
        ob.test();
        ob.test(10);
        ob.test(10, 20);
        result = ob.test(123.25);
        System.out.println("Результат вызова ob.test(123.25): " + result);
    }
}

```

Эта программа выводит следующий результат:

```

Параметры отсутствуют
a: 10
a и b: 10 20
double a: 123.25
Результат вызова ob.test(123.25): 15190.5625

```

Как видите, метод `test()` перегружается четыре раза. Первый его вариант вообще не принимает параметров, второй принимает один целочисленный параметр, третий вариант — два целочисленных параметра, а четвертый — один параметр типа `double`. Тот факт, что четвертый вариант метода `test()` еще и возвращает значение, не имеет никакого значения для перегрузки, поскольку возвращаемый тип ни коим образом не влияет на поиск перегружаемого варианта метода.

При вызове перегружаемого метода в Java обнаруживается соответствие аргументов, использованных для вызова метода, и параметров метода. Но это соответствие совсем не обязательно должно быть полным. Иногда важную роль в разрешении перегрузки может играть автоматическое преобразование типов в Java. Рассмотрим в качестве примера следующую программу:

```

// Применить автоматическое преобразование типов к перегрузке
class OverloadDemo {
    void test() {
        System.out.println("Параметры отсутствуют");
    }

    // Перегружаемый метод, проверяющий наличие
    // двух целочисленных параметров
    void test(int a, int b) {
        System.out.println("a и b: " + a + " " + b);
    }

    // Перегружаемый метод, проверяющий наличие
    // параметра типа double
    void test(double a) {

```

```
System.out.println(
    "Внутреннее преобразование при вызове test(double) a: " + a);
}
class Overload {
    public static void main(String args[]) {
        OverloadDemo ob = new OverloadDemo();
        int i = 88;

        ob.test();
        ob.test(10, 20);

        ob.test(i); // здесь вызывается вариант метода test(double)
        ob.test(123.2); // а здесь вызывается вариант метода test(double)
    }
}
```

Эта программа выводит следующий результат:

Параметры отсутствуют

a и b: 10 20

Внутреннее преобразование при вызове test(double) a: 88

Внутреннее преобразование при вызове test(double) a: 123.2

Как видите, в данной версии класса OverloadDemo перегружаемый вариант метода `test(int)` не определяется. Поэтому при вызове метода `test()` с целочисленным аргументом в классе Overload отсутствует соответствующий метод. Но в Java может быть автоматически выполнено преобразование типа `integer` в тип `double`, чтобы разрешить вызов нужного варианта данного метода. Так, если вариант метода `test(int)` не обнаружен, тип переменной `i` автоматически продвигается в Java к типу `double`, а затем вызывается вариант метода `test(double)`. Безусловно, если бы вариант метода `test(int)` был определен, то был бы вызван именно он. Автоматическое преобразование типов в Java выполняется только в том случае, если не обнаружено полное соответствие.

Перегрузка методов поддерживает полиморфизм, поскольку это один из способов реализации в Java принципа “один интерфейс, несколько методов”. Разъясним это положение подробнее, чтобы оно стало понятнее. В тех языках программирования, где перегрузка методов не поддерживается, каждому методу должно быть присвоено однозначное имя. Но зачастую требуется реализовать, по существу, один и тот же метод для разных типов данных. Рассмотрим в качестве примера функцию, вычисляющую абсолютное значение. Обычно в тех языках программирования, где перегрузка методов не поддерживается, существуют три или больше варианта этой функции с несколько отличающимися именами. Например, в языке C функция `abs()` возвращает абсолютное значение типа `integer`, функция `labs()` — абсолютное значение типа `long integer`, а функция `fabs()` — абсолютное значение с плавающей точкой. В языке C перегрузка не поддерживается, и поэтому у каждой из этих функций должно быть свое имя, несмотря на то, что все три функции выполняют, по существу, одно и то же действие. В итоге ситуация становится принципиально более сложной, чем она есть на самом деле. И хотя каждая функция построена по одному и тому же принципу, программирующему на C приходится помнить три разных имени одной и той же функции. В Java подобная ситуация не возникает, поскольку все методы вычисления абсолютного значения могут называться одинаково. И действительно, в состав стандартной библиотеки классов Java входит метод вычисления абсолютного значения, называемый `abs()`.

Перегружаемые варианты этого метода для обработки всех числовых типов данных определены во встроенном в Java классе `Math`. И в зависимости от типа аргумента в Java вызывается нужный вариант метода `abs()`.

Перегрузка методов ценна тем, что позволяет обращаться к похожим методам по общему имени. Следовательно, имя `abs` представляет *общее действие*, которое должно выполняться. Выбор *подходящего* варианта метода для конкретной ситуации входит в обязанности компилятора, а программисту нужно лишь запомнить общее выполняемое действие. Полиморфизм позволяет свести несколько имен к одному. Приведенный выше пример довольно прост, но если расширить продемонстрированный в нем принцип, то нетрудно убедиться, что перегрузка позволяет упростить решение более сложных задач.

При перегрузке метода каждый его вариант может выполнять любые требующиеся действия. Не существует правила, согласно которому перегружаемые методы должны быть связаны друг с другом. Но со стилистической точки зрения перегрузка методов предполагает определенную их связь. И хотя одно и то же имя можно было бы употребить для перегрузки несвязанных методов, делать этого все же не следует. Например, имя `sqrt` можно было бы употребить для создания методов, возвращающих *квадрат* целочисленного значения и *квадратный корень* числового значения с плавающей точкой. Но эти две операции принципиально различны. Такое применение перегрузки методов противоречит ее первоначальному назначению. Поэтому на практике перегружать следует только тесно связанные операции.

Перегрузка конструкторов

Наряду с перегрузкой обычных методов можно также выполнять перегрузку методов-конструкторов. По существу, перегружаемые конструкторы станут скорее нормой, а не исключением для большинстве классов, которые вам придется создавать на практике. Чтобы стали понятнее причины этого, вернемся к классу `Box`, разработанному в предыдущей главе. Ниже приведена самая последняя его версия.

```
class Box {
    double width;
    double height;
    double depth;

    // Это конструктор класса Box
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // рассчитать и вернуть объем
    double volume() {
        return width * height * depth;
    }
}
```

Как видите, конструктору `Box()` требуется передать три параметра. Это означает, что во всех объявлениях объектов класса `Box` конструктору `Box()` должны передаваться три аргумента. Например, следующая строка кода недопустима:

```
Box ob = new Box();
```

Вызов конструктора `Box()` без аргументов приводит к ошибке, поскольку ему следует непременно передать три аргумента. В связи с этим возникают следующие важные вопросы: что, если требуется просто определить параллелепипед, а его начальные размеры не имеют значения или вообще неизвестны, и можно ли инициализировать куб, указав только одно значение для всех трех измерений? Текущее определение класса `Box` не дает ответы на эти вопросы, поскольку запрашиваемые дополнительные возможности в нем отсутствуют.

Правда, разрешить подобные вопросы совсем не трудно: достаточно перегрузить конструктор `Box()`, чтобы принять во внимание упомянутые выше требования. Ниже приведен пример программы с усовершенствованной версией класса `Box`, где решается подобная задача.

```
/* В данном примере конструкторы определяются в классе Box для
   инициализации размеров параллелепипеда тремя разными способами
*/
class Box {
    double width;
    double height;
    double depth;

    // конструктор, используемый при указании всех размеров
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // конструктор, используемый, когда ни один из размеров не указан
    Box() {
        width = -1; // использовать значение -1 для обозначения
        height = -1; // неинициализированного
        depth = -1; // параллелепипеда
    }

    // конструктор, используемый при создании куба
    Box(double len) {
        width = height = depth = len;
    }

    // рассчитать и вернуть объем
    double volume() {
        return width * height * depth;
    }
}

class OverloadCons {
    public static void main(String args[]) {
        // создать параллелепипеды, используя разные
        // конструкторы
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);

        double vol;

        // получить объем первого параллелепипеда
        vol = mybox1.volume();
        System.out.println("Объем mybox1 равен " + vol);

        // получить объем второго параллелепипеда
```

```

        vol = mybox2.volume();
        System.out.println("Объем mybox2 равен " + vol);

        // получить объем куба
        vol = mycube.volume();
        System.out.println("Объем mycube равен " + vol);
    }
}

```

Эта программа выводит следующий результат:

```

Объем mybox1 равен 3000.0
Объем mybox2 равен -1.0
Объем mycube равен 343.0

```

Как видите, соответствующий перегружаемый конструктор вызывается в зависимости от параметров, указываемых при выполнении оператора `new`.

Применение объектов в качестве параметров

В представленных до сих пор примерах программ в качестве параметров методов употреблялись только простые типы данных. Но передача методам объектов не только вполне допустима, но и довольно распространена. Рассмотрим в качестве примера следующую короткую программу:

```

// Объекты допускается передавать методам в качестве параметров
class Test {
    int a, b;

    Test(int i, int j) {
        a = i;
        b = j;
    }

    // вернуть логическое значение true, если в качестве
    // параметра o указан вызывающий объект
    boolean equals(Test o) {
        if(o.a == a && o.b == b) return true;
        else return false;
    }
}

class PassOb {
    public static void main(String args[]) {
        Test ob1 = new Test(100, 22);
        Test ob2 = new Test(100, 22);
        Test ob3 = new Test(-1, -1);

        System.out.println("ob1 == ob2: " + ob1.equals(ob2));
        System.out.println("ob1 == ob3: " + ob1.equals(ob3));
    }
}

```

Эта программа выводит следующий результат:

```

ob1 == ob2: true
ob1 == ob3: false

```

Как видите, метод `equals()` проверяет в классе `Test` на равенство два объекта и возвращает получаемый результат. Таким образом, он сравнивает вызывающий объект с тем, который был ему передан. Если оба объекта содержат одинаковые зна-

чения, метод `equals()` возвращает логическое значение `true`, а иначе — логическое значение `false`. Обратите внимание на то, что в качестве типа параметра `o` в методе `equals()` указывается класс `Test`. И хотя `Test` обозначает тип класса, создаваемого в программе, он употребляется точно так же, как и типы данных, встроенные в Java.

В качестве параметров объекты чаще всего употребляются в конструкторах. Нередко новый объект приходится создавать таким образом, чтобы он первоначально ничем не отличался от уже существующего объекта. Для этого придется определить конструктор, принимающий в качестве параметра объект своего класса. Например, приведенная ниже очередная версия класса `Box` позволяет инициализировать один объект другим.

```
// В этой версии класса Box один объект допускается
// инициализировать другим объектом
class Box {
    double width;
    double height;
    double depth;

    // Обратите внимание на этот конструктор.
    // В качестве параметра в нем используется объект типа Box
    Box(Box ob) { // передать объект конструктору
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // конструктор, используемый при указании всех размеров
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // конструктор, используемый, если ни один из размеров не указан
    Box() {
        width = -1; // использовать значение -1 для обозначения
        height = -1; // неинициализированного
        depth = -1; // параллелепипеда
    }

    // конструктор, используемый при создании куба
    Box(double len) {
        width = height = depth = len;
    }

    // рассчитать и вернуть объем
    double volume() {
        return width * height * depth;
    }
}

class OverloadCons2 {
    public static void main(String args[]) {
        // создать параллелепипеды, используя
        // разные конструкторы
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);

        Box myclone = new Box(mybox1);
    }
}
```

```

// создать копию объекта mybox1
double vol;

// получить объем первого параллелепипеда
vol = mybox1.volume();
System.out.println("Объем mybox1 равен " + vol);

// получить объем второго параллелепипеда
vol = mybox2.volume();
System.out.println("Объем mybox2 равен " + vol);

// получить объем куба
vol = mycube.volume();
System.out.println("Объем куба равен " + vol);

// получить объем клона
vol = myclone.volume();
System.out.println("Объем клона равен " + vol);
}
}

```

Приступив к разработке своих классов, вы быстро обнаружите потребность иметь в своем распоряжении многие формы конструкторов. Они позволят вам удобно и эффективно создавать нужные объекты.

Подробное рассмотрение особенностей передачи аргументов

В общем, для передачи аргументов подпрограмме в языках программирования имеются два способа. Первым способом является *вызов по значению*. В этом случае *значение* аргумента копируется в формальный параметр подпрограммы. Следовательно, изменения, вносимые в параметр подпрограммы, не оказывают никакого влияния на аргумент. Вторым способом передачи аргумента является *вызов по ссылке*. В этом случае параметру передается ссылка на аргумент, а не его значение. В теле подпрограммы эта ссылка служит для обращения к конкретному аргументу, указанному в вызове. Это означает, что изменения, вносимые в параметр подпрограммы, будут оказывать влияние на аргумент, используемый при ее вызове. Как будет показано далее, все аргументы в Java передаются при вызове по значению, но конкретный результат зависит от того, какой именно тип данных передается: примитивный или ссылочный.

Когда методу передается аргумент примитивного типа, его передача происходит по значению. В итоге создается копия аргумента, и все, что происходит с параметром, принимающим этот аргумент, не оказывает никакого влияния за пределами вызываемого метода. Рассмотрим в качестве примера следующую программу:

```

// Аргументы примитивных типов передаются по значению
class Test {
    void meth(int i, int j) {
        i *= 2;
        j /= 2;
    }
}

```

```
class CallByValue {
    public static void main(String args[]) {
        Test ob = new Test();

        int a = 15, b = 20;

        System.out.println("a и b до вызова: " +
                           a + " " + b);

        ob.meth(a, b);

        System.out.println("a и b после вызова: " +
                           a + " " + b);
    }
}
```

Эта программа выводит следующий результат:

```
a и b до вызова: 15 20
a и b после вызова: 15 20
```

Как видите, операции, выполняемые в теле метода `meth()`, не оказывают никакого влияния на значения переменных `a` и `b`, используемых при вызове этого метода. Их значения *не* изменились на 30 и 10 соответственно, но остались прежними.

При передаче объекта в качестве аргумента методу ситуация меняется коренным образом, поскольку объекты, по существу, передаются при вызове по ссылке. Не следует, однако, забывать, что при объявлении переменной типа класса создается лишь ссылка на объект этого класса. Таким образом, при передаче этой ссылки методу принимающий ее параметр будет ссылаться на тот же самый объект, на который ссылается и аргумент. По существу, это означает, что объекты действуют так, как будто они передаются методам по ссылке. Но изменения объекта в теле метода оказывают *влияние* на объект, указываемый в качестве аргумента. Рассмотрим в качестве примера следующую программу:

// Объекты передаются по ссылке на них

```
class Test {
    int a, b;

    Test(int i, int j) {
        a = i;
        b = j;
    }

    // передать объект
    void meth(Test o) {
        o.a *= 2;
        o.b /= 2;
    }
}

class PassObjRe {
    public static void main(String args[]) {
        Test ob = new Test(15, 20);
        System.out.println("ob.a и ob.b до вызова: " +
                           ob.a + " " + ob.b);

        ob.meth(ob);

        System.out.println("ob.a и ob.b после вызова: " +
```



```

        ob.a + " " + ob.b);
    }
}

```

Эта программа выводит следующий результат:

```

ob.a и ob.b до вызова: 15 20
ob.a и ob.b после вызова: 30 10

```

Как видите, в данном случае действия, выполняемые в теле метода `meth()`, оказывают влияние на объект, указываемый в качестве аргумента.

Помните! Когда методу передается ссылка на объект, сама ссылка передается способом вызова по значению. Но поскольку передаваемое значение ссылается на объект, то копия этого значения все равно будет ссылаться на тот же самый объект, что и соответствующий аргумент.

Возврат объектов

Метод может возвращать любой тип данных, в том числе созданные типы классов. Так, в приведенном ниже примере программы метод `incrByTen()` возвращает объект, в котором значение переменной `a` на 10 больше значения этой переменной в вызывающем объекте.

```

// Возврат объекта
class Test {
    int a;

    Test(int i) {
        a = i;
    }

    Test incrByTen() {
        Test temp = new Test(a+10);
        return temp;
    }
}

class RetOb {
    public static void main(String args[]) {
        Test ob1 = new Test(2);
        Test ob2;

        ob2 = ob1.incrByTen();
        System.out.println("ob1.a: " + ob1.a);
        System.out.println("ob2.a: " + ob2.a);

        ob2 = ob2.incrByTen();
        System.out.println("ob2.a после второго увеличения значения: "
                           + ob2.a);
    }
}

```

Эта программа выводит следующий результат:

```

ob1.a: 2
ob2.a: 12
ob2.a после второго увеличения значения: 22

```

Как видите, при каждом вызове метода `incrByTen()` в данной программе создается новый объект, а ссылка на него возвращается вызывающей части программы. В данной программе демонстрируется еще один важный момент: память выделяется для всех объектов динамически с помощью оператора `new`, а следовательно, программисту не нужно принимать никаких мер, чтобы объект не вышел за пределы области своего действия, поскольку выполнение метода, в котором он был создан, прекращается. Объект будет существовать до тех пор, пока будет существовать ссылка на него в каком-нибудь другом месте программы. В отсутствие любых ссылок на объект он будет уничтожен при последующей сборке “мусора”.

Рекурсия

В языке Java поддерживается *рекурсия* — процесс определения чего-либо относительно самого себя. Применительно к программированию на Java рекурсия — это средство, которое позволяет методу вызывать самого себя. Такой метод называется *рекурсивным*.

Классическим примером рекурсии служит вычисление факториала числа. Факториал числа N — это произведение всех целых чисел от 1 до N . Например, факториал числа 3 равен $1 \times 2 \times 3$, т.е. 6. Ниже показано, как вычислить факториал, используя рекурсивный метод.

```
// Простой пример рекурсии
class Factorial {
    // это рекурсивный метод
    int fact(int n) {
        int result;

        if(n==1) return 1;
        result = fact(n-1) * n;
        return result;
    }
}

class Recursion {
    public static void main(String args[]) {
        Factorial f = new Factorial();

        System.out.println("Факториал 3 равен " + f.fact(3));
        System.out.println("Факториал 4 равен " + f.fact(4));
        System.out.println("Факториал 5 равен " + f.fact(5));
    }
}
```

Ниже приведен результат, выводимый этой программой.

```
Факториал 3 равен 6
Факториал 4 равен 24
Факториал 5 равен 120
```

Тем, кто незнаком с рекурсивными методами, принцип действия метода `fact()` может быть не совсем понятным. Метод `fact()` действует следующим образом. Когда этот метод вызывается со значением 1 своего аргумента, возвращается значение 1. В противном случае возвращается произведение `fact(n-1) * n`. Для вы-

числения этого выражения метод `fact()` вызывается со значением $n-1$ своего аргумента. Этот процесс повторяется до тех пор, пока n не станет равным 1, после чего начнется возврат из последовательных вызовов метода `fact()`.

Для того чтобы стал понятнее принцип действия рекурсивного метода `fact()`, обратимся к небольшому примеру. Для расчета факториала числа 3 вслед за первым вызовом метода `fact()` происходит второй вызов этого метода со значением 2 его аргумента. Это, в свою очередь, приводит к третьему вызову метода `fact()` со значением 1 его аргумента. Возвращаемое из этого вызова значение 1 затем умножается на 2 (т.е. значение n при втором вызове метода). Полученный результат, равный 2, возвращается далее исходному вызову метода `fact()` и умножается на 3 (исходное значение n). В конечном итоге получается искомый результат, равный 6. В метод `fact()` можно было бы ввести вызовы метода `println()`, чтобы отображать уровень каждого вызова и промежуточные результаты вычисления факториала заданного числа.

Когда рекурсивный метод вызывает самого себя, новым локальным переменным и параметрам выделяется место в стеке и код метода выполняется с этими новыми исходными значениями. При каждом возврате из вызова рекурсивного метода прежние локальные переменные и параметры удаляются из стека, а выполнение продолжается с точки вызова в самом методе. Рекурсивные методы выполняют действия, которые можно сравнить с раскладыванием и складыванием телескопа.

Вследствие издержек на дополнительные вызовы рекурсивные варианты многих процедур могут выполняться медленнее их итерационных аналогов. Слишком большое количество вызовов рекурсивного метода может привести к переполнению стека, поскольку параметры и локальные переменные сохраняются в стеке, а при каждом новом вызове создаются новые копии этих значений. В таком случае в исполняющей системе Java возникнет исключение. Но подобная ситуация не возникнет, если не выпустить рекурсивный метод из-под контроля.

Главное преимущество рекурсивных методов заключается в том, что их можно применять для реализации более простых и понятных вариантов некоторых алгоритмов, чем их итерационные аналоги. Например, алгоритм быстрой сортировки очень трудно реализовать итерационным способом. А некоторые виды алгоритмов, связанных с искусственным интеллектом, легче всего реализовать с помощью рекурсивных решений.

При написании рекурсивных методов следует позаботиться о том, чтобы в каком-нибудь другом месте программы присутствовал условный оператор `if`, осуществляющий возврат из метода без его рекурсивного вызова. В противном случае возврата из рекурсивно вызываемого метода так и не произойдет. Подобная ошибка очень часто встречается при организации рекурсии. Поэтому на стадии разработки рекурсивных методов рекомендуется как можно чаще делать вызовы метода `println()`, чтобы следить за происходящим и прерывать выполнение при обнаружении ошибки.

Рассмотрим еще один пример организации рекурсии. В данном примере рекурсивный метод `printArray()` выводит первые i элементов из массива `values`.

```
// Еще один пример рекурсии
```

```
class RecTest {
    int values[];
```

```
RecTest(int i) {
    values = new int[i];
}

// вывести рекурсивно элементы массива
void printArray(int i) {
    if(i==0) return;
    else printArray(i-1);
    System.out.println "[" + (i-1) + " ] " + values[i-1]);
}

class Recursion2 {
    public static void main(String args[]) {
        RecTest ob = new RecTest(10);
        int i;

        for(i=0; i<10; i++) ob.values[i] = i;

        ob.printArray(10);
    }
}
```

Эта программа выводит следующий результат:

```
[0] 0
[1] 1
[2] 2
[3] 3
[4] 4
[5] 5
[6] 6
[7] 7
[8] 8
[9] 9
```

Введение в управление доступом

Как вам, должно быть уже известно, инкапсуляция связывает данные с манипулирующим ими кодом. Но инкапсуляция предоставляет еще одно важное средство: *управление доступом*. Инкапсуляция позволяет управлять доступом к членам класса из отдельных частей программы, а следовательно, предотвращать злоупотребления. Например, предоставляя доступ к данным только с помощью вполне определенного ряда методов, можно предотвратить злоупотребление этими данными. Так, если класс реализован правильно, он создает своего рода “черный ящик”, которым можно пользоваться, но внутренний механизм которого защищен от повреждений. Но представленные ранее классы не полностью удовлетворяют этому требованию. Рассмотрим, например, класс `Stack`, представленный в конце главы 6. Несмотря на то что методы `push()` и `pop()` действительно предоставляют управляемый интерфейс со стеком, придерживаться этого интерфейса совсем не обязательно. Это означает, что другая часть программы может обойти эти методы и обратиться к стеку непосредственно. Ясно, что злоумышленник не преминет воспользоваться такой возможностью. В этом разделе представлен механизм, с помощью которого можно точно управлять доступом к разным членам класса.

Способ доступа к члену класса определяется *модификатором доступа*, присутствующим в его объявлении. В Java определен обширный ряд модификаторов доступа. Некоторые аспекты управления доступом связаны главным образом с наследованием и пакетами. (*Пакет* — это, по существу, группировка классов.) Эти составляющие механизма управления доступом в Java будут рассмотрены в последующих разделах, а сейчас рассмотрим управление доступом применительно к отдельному классу. Когда основы управления доступом станут вам понятны, освоение других особенностей данного процесса не составит особого труда.

В Java определяются следующие модификаторы доступа: `public` (открытый), `private` (закрытый) и `protected` (защищенный), а также уровень доступа, предоставляемый по умолчанию. Модификатор `protected` применяется только при наследовании. Остальные модификаторы доступа описаны далее в этой главе.

Начнем с определения модификаторов `public` и `private`. Когда член объявляется с модификатором доступа `public`, он становится доступным из любого другого кода. А когда член класса объявляется с модификатором доступа `private`, он доступен только другим членам этого же класса. Теперь становится понятно, почему в объявлении метода `main()` всегда присутствует модификатор `public`. Этот метод вызывается из кода, находящегося за пределами данной программы, т.е. из исполняющей системы Java. В отсутствие модификатора доступа по умолчанию член класса считается открытым в своем пакете, но недоступным для кода, находящегося за пределами этого пакета. (Пакеты рассматриваются в следующей главе.)

В рассмотренных ранее примерах классов все члены класса действовали в выбираемом по умолчанию режиме доступа. Но этот режим доступа зачастую не соответствует практическим требованиям. Как правило, доступ к данным класса приходится ограничивать, предоставляя доступ к ним только через методы. А в ряде случаев в классе придется определять и закрытые методы.

Модификатор доступа предшествует остальной спецификации типа члена. Это означает, что оператор объявления члена должен начинаться с модификатора доступа, как показано ниже.

```
public int i;  
private double j;  
  
private int myMethod(int a, char b) { // ...
```

Чтобы стало понятнее влияние, которое оказывает организация открытого и закрытого доступа, рассмотрим следующий пример программы:

```
/* В этой программе демонстрируется отличие  
модификаторов public и private.  
*/  
class Test {  
    int a;           // доступ, определяемый по умолчанию  
    public int b;     // открытый доступ  
    private int c;    // закрытый доступ  
  
    // методы доступа к члену с данного класса  
    void setc(int i) { // установить значение члена с данного класса  
        c = i;  
    }  
    int getc() {      // получить значение члена с данного класса  
        return c;  
    }  
}
```

```

    }
}

class AccessTest {
    public static void main(String args[]) {
        Test ob = new Test();

        // Эти операторы правильны, поэтому члены a и b
        // данного класса доступны непосредственно
        ob.a = 10;
        ob.b = 20;

        // Этот оператор неверен и может вызвать ошибку
        // ob.c = 100;    // ОШИБКА!

        // Доступ к члену c данного класса должен осуществляться
        // с помощью методов ее класса
        ob.setc(100); // ВЕРНО!
        System.out.println("a, b, и c: " + ob.a + " " + ob.b +
                           " " + ob.getc());
    }
}

```

Как видите, в классе `Test` используется режим доступа, выбираемый по умолчанию, что в данном примере равносильно указанию модификатора доступа `public`. Член `b` данного класса явно указан как `public`, тогда как член `c` указан как закрытый. Это означает, что он недоступен из кода за пределами его класса. Поэтому в самом классе `AccessTest` член `c` не может применяться непосредственно. Доступ к нему должен осуществляться с помощью открытых методов `setc()` и `getc()` данного класса. Удаление символов комментариев в начале следующей строки кода:

```
// ob.c = 100; // ОШИБКА!
```

сделало бы компиляцию данной программы невозможной из-за нарушений правил доступа.

В качестве более реального примера организации управления доступом рассмотрим следующую усовершенствованную версию класса `Stack`, код которого был приведен в конце главы 6:

```

// В этом классе определяется целочисленный стек, который может
// содержать 10 значений
class Stack {
    /* Теперь переменные stck и tos являются закрытыми.
       Это означает, что они не могут быть случайно или
       намеренно изменены таким образом, чтобы нарушить стек.
    */
    private int stck[] = new int[10];
    private int tos;

    // инициализировать вершину стека
    Stack() {
        tos = -1;
    }

    // разместить элемент в стеке
    void push(int item) {
        if(tos==9)
            System.out.println("Стек заполнен.");
        else

```

```

        stck[++tos] = item;
    }

    // извлечь элемент из стека
    int pop() {
        if(tos < 0) {
            System.out.println("Стек не загружен.");
            return 0;
        }
        else
            return stck[tos--];
    }
}

```

Как видите, теперь как `private` объявлены обе переменные: `stck`, содержащая стек, а также `tos`, содержащая индекс вершины стека. Это означает, что обращение к ним или их изменение может осуществляться только через методы `push()` и `pop()`. Например, объявление переменной `tos` закрытой препятствует случайной установке ее значения за границами массива `stck` из других частей программы.

В следующем примере представлена усовершенствованная версия класса `Stack`. Чтобы убедиться в том, что члены класса `stck` и `tos` действительно недоступны, попытайтесь удалить символы комментариев из закомментированных строк кода.

```

class TestStack {
    public static void main(String args[]) {
        Stack mystack1 = new Stack();
        Stack mystack2 = new Stack();

        // разместить числа в стеке
        for(int i=0; i<10; i++) mystack1.push(i);
        for(int i=10; i<20; i++) mystack2.push(i);

        // извлечь эти числа из стека
        System.out.println("Стек в mystack1:");
        for(int i=0; i<10; i++)
            System.out.println(mystack1.pop());

        System.out.println("Стек в mystack2:");

        for(int i=0; i<10; i++)
            System.out.println(mystack2.pop());

        // эти операторы недопустимы
        // mystack1.tos = -2;
        // mystack2.stck[3] = 100;
    }
}

```

Обычно методы будут обеспечивать доступ к данным, которые определены в классе, хотя это и не обязательно. Переменная экземпляра вполне может быть открытой, если на то имеются веские основания. Так, ради простоты переменные экземпляра в большинстве несложных классов, представленных в данной книге, определены как открытые. Но в большинстве классов, применяемых в реальных программах, манипулирование данными должно осуществляться только с помощью методов. Мы еще вернемся в следующей главе к теме управления доступом, и у вас будет возможность убедиться в том, что управление доступом особенно важно при наследовании.

Ключевое слово `static`

Иногда желательно определить член класса, который будет использоваться независимо от любого объекта этого класса. Как правило, обращение к члену класса должно осуществляться только в сочетании с объектом его класса. Но можно создать член класса, чтобы пользоваться им отдельно, не ссылаясь на конкретный экземпляр. Чтобы создать такой член, в начале его объявления нужно разместить ключевое слово `static`. Когда член класса объявлен как `static` (статический), он доступен до создания любых объектов его класса и без ссылки на какой-нибудь объект. Статическими могут быть объявлены как методы, так и переменные. Наиболее распространенным примером статического члена служит метод `main()`, который объявляется как `static`, поскольку он должен быть объявлен до создания любых объектов.

Переменные экземпляра, объявленные как `static`, по существу, являются глобальными. При объявлении объектов класса этих переменных их копии не создаются. Вместо этого все экземпляры класса совместно используют одну и ту же статическую переменную.

На методы, объявленные как `static`, накладывается следующие ограничения:

- Они могут непосредственно вызывать только другие статические методы.
- Им непосредственно доступны только статические переменные.
- Они никоим образом не могут делать ссылки типа `this` или `super`. (Ключевое слово `super` связано с наследованием и описывается в следующей главе.)

Если для инициализации статических переменных требуется произвести вычисления, то для этой цели достаточно объявить статический блок, который будет выполняться только один раз при первой загрузке класса. В приведенном ниже примере демонстрируется класс, который содержит статический метод, несколько статических переменных и статический блок инициализации.

```
// Продемонстрировать статические переменные, методы и блоки кода
class UseStatic {
    static int a = 3;
    static int b;

    static void meth(int x) {
        System.out.println("x = " + x);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
    }

    static {
        System.out.println("Статический блок инициализирован.");
        b = a * 4;
    }

    public static void main(String args[]) {
        meth(42);
    }
}
```

Как только загружается класс `UseStatic`, выполняются все статические операторы. Сначала в переменной `a` устанавливается значение **3**, затем выполняется статический блок кода, в котором выводится сообщение, а переменная `b` иници-

ализируется значением $a * 4$, т.е. 12. После этого вызывается метод `main()`, который, в свою очередь, вызывает метод `meth()`, передавая параметру `x` значение 42. В трех вызовах метода `println()` делаются ссылки на две статические переменные, `a` и `b`, а также на локальную переменную `x`.

Результат, выводимый этой программой, выглядит следующим образом:

Статический блок инициализирован.

```
x = 42
a = 3
b = 12
```

За пределами класса, в котором определены статические методы и переменные, ими можно пользоваться независимо от любого объекта. Для этого достаточно указать имя их класса через операцию-точку непосредственно перед их именами. Так, если требуется вызвать статический метод за пределами его класса, это можно сделать, используя следующую общую форму:

имя_класса.метод()

где `имя_класса` обозначает имя того класса, в котором объявлен статический метод. Как видите, эта форма аналогична той, что применяется для вызова нестатических методов через переменные ссылки на объекты. Аналогично для доступа к статической переменной ее имя следует предварить именем ее класса через операцию-точку. Именно так в Java реализованы управляемые версии глобальных методов и переменных.

Обратимся к конкретному примеру. В теле метода `main()` обращение к статическому методу `callme()` и статической переменной `b` осуществляется по имени их класса `StaticDemo`, как показано ниже.

```
class StaticDemo {
    static int a = 42;
    static int b = 99;

    static void callme() {
        System.out.println("a = " + a);
    }
}

class StaticByName {
    public static void main(String args[]) {
        StaticDemo.callme();
        System.out.println("b = " + StaticDemo.b);
    }
}
```

Эта программа выводит следующий результат:

```
a = 42
b = 99
```

Ключевое слово **final**

Поле может быть объявлено как **final** (завершенное). Это позволяет предотвратить изменение содержимого переменной, сделав ее, по существу, константой. Следовательно, завершенное поле должно быть инициализировано во время его

объявления. Значение такому полю можно присвоить и в пределах конструктора, но первый подход более распространен. Ниже приведен ряд примеров объявления завершенных полей.

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

Теперь во всех последующих частях программы можно пользоваться полем `FILE_OPEN` и прочими полями таким образом, как если бы они были константами, без риска изменить их значения. В практике программирования на Java идентификаторы всех завершенных полей принято обозначать прописными буквами, как в приведенном выше примере.

Кроме полей, объявленными как `final` могут быть параметры метода и локальные переменные. Объявление параметра как `final` препятствует его изменению в пределах метода, тогда как аналогичное объявление локальной переменной — присвоению ей значения больше одного раза.

Ключевое слово `final` можно указывать и в объявлении методов, но в этом случае оно имеет совсем иное назначение, чем в переменных. Это дополнительное применение ключевого слова `final` более подробно описано в следующей главе, посвященной наследованию.

Еще раз о массивах

Массивы были представлены ранее в этой книге еще до рассмотрения классов. Теперь, имея представление о классах, можно сделать следующий важный вывод относительно массивов: все они реализованы как объекты. В связи с этим для массивов существует специальное средство, которым выгодно воспользоваться при программировании на Java. В частности, размер массива, т.е. количество элементов, которые может содержать массив, хранится в его переменной экземпляра `length`. Все массивы обладают этой переменной, которая всегда будет содержать размер массива. Ниже приведен пример программы, которая демонстрирует это средство.

```
// В этой программе демонстрируется применение члена длины массива
class Length {
    public static void main(String args[]) {
        int a1[] = new int[10];
        int a2[] = {3, 5, 7, 1, 8, 99, 44, -10};
        int a3[] = {4, 3, 2, 1};

        System.out.println("длина a1 равна " + a1.length);
        System.out.println("длина a2 равна " + a2.length);
        System.out.println("длина a3 равна " + a3.length);
    }
}
```

Эта программа выводит следующий результат:

```
длина a1 равна 10
длина a2 равна 8
длина a3 равна 4
```

Как видите, в данной программе выводится размер каждого массива. Имейте в виду, что значение переменной `length` никак не связано с количеством действительно используемых элементов массива. Оно отражает лишь то количество элементов, которое может содержать массив.

Члену `length` можно найти применение во многих случаях. В качестве примера ниже приведена усовершенствованная версия класса `Stack`. Напомним, что в предшествующих версиях этого класса всегда создавался 10-элементный стек. А новая версия класса `Stack` позволяет создавать стеки любого размера. Значение `stack.length` служит для того, чтобы предотвратить переполнение стека.

```
// Усовершенствованный класс Stack, в котором
// использован член длины массива
class Stack {
    private int stck[];
    private int tos;

    // выделить память под стек и инициализировать его
    Stack(int size) {
        stck = new int[size];
        tos = -1;
    }

    // разметить элемент в стеке
    void push(int item) {
        if(tos==stck.length-1) // использовать член длины массива
            System.out.println("Стек заполнен.");
        else
            stck[++tos] = item;
    }

    // извлечь элемент из стека
    int pop() {
        if(tos < 0) {
            System.out.println("Стек не загружен.");
            return 0;
        }
        else
            return stck[tos--];
    }
}

class TestStack2 {
    public static void main(String args[]) {
        Stack mystack1 = new Stack(5);
        Stack mystack2 = new Stack(8);

        // разместить числа в стеке
        for(int i=0; i<5; i++) mystack1.push(i);
        for(int i=0; i<8; i++) mystack2.push(i);

        // извлечь эти числа из стека
        System.out.println("Стек в mystack1:");
        for(int i=0; i<5; i++)
            System.out.println(mystack1.pop());

        System.out.println("Стек в mystack2:");
        for(int i=0; i<8; i++)
            System.out.println(mystack2.pop());
    }
}
```

Обратите внимание на то, что в данной программе создаются два стека: один глубиной в пять элементов, другой — в шесть. Как видите, тот факт, что в массивах поддерживается информация об их длине, упрощает организацию стеков любой величины.

Вложенные и внутренние классы

В языке Java допускается определять один класс в другом классе. Такие классы называются *вложенными*. Область действия вложенного класса ограничена областью действия внешнего класса. Так, если класс В определен в классе А, то класс В не может существовать независимо от класса А. Вложенный класс имеет доступ к членам (в том числе закрытым) того класса, в который он вложен. Но внешний класс не имеет доступа к членам вложенного класса. Вложенный класс, объявленный непосредственно в области действия своего внешнего класса, считается его членом. Можно также объявлять вложенные классы, являющиеся локальными для блока кода.

Существуют два типа вложенных классов: *статические* и *нестатические*. Статическим называется такой вложенный класс, который объявляется с модификатором `static`. А поскольку он является статическим, то должен обращаться к нестатическим членам своего внешнего класса посредством объекта. Это означает, что вложенный статический класс не может непосредственно ссылаться на нестатические члены своего внешнего класса. В силу этого ограничения статические вложенные классы применяются редко.

Наиболее важным типом вложенного класса является *внутренний* класс. Внутренний класс — это нестатический вложенный класс. Он имеет доступ ко всем переменным и методам своего внешнего класса и может непосредственно ссылаться на них таким же образом, как это делают остальные нестатические члены внешнего класса.

В приведенном ниже примере программы демонстрируется определение и использование внутреннего класса. В классе `Outer` содержится одна переменная экземпляра `outer_x`, один метод экземпляра `test()` и определяется один внутренний класс `Inner`.

```
// Продемонстрировать применение внутреннего класса
class Outer {
    int outer_x = 100;

    void test() {
        Inner inner = new Inner();
        inner.display();
    }

    // это внутренний класс
    class Inner {
        void display() {
            System.out.println("вывод: outer_x = " + outer_x);
        }
    }
}
```

```

class InnerClassDemo {
    public static void main(String args[]) {
        Outer outer = new Outer();
        outer.test();
    }
}

```

Эта программа выводит следующий результат:

вывод: outer_x = 100

В данной программе внутренний класс Inner определен в области действия класса Outer. Поэтому любой код из класса Inner может непосредственно обращаться к переменной outer_x. Метод экземпляра display() определен в классе Inner. Этот метод выводит значение переменной outer_x в стандартный поток вывода. В методе main() из класса InnerClassDemo создается экземпляр класса Outer и вызывается его метод test(). А в этом методе создается экземпляр класса Inner и вызывается метод display().

Следует иметь в виду, что экземпляр класса Inner может быть создан только в контексте класса Outer. В противном случае компилятор Java выдаст сообщение об ошибке. В общем, экземпляр внутреннего класса нередко создается в коде, находящемся в объемлющей области действия, как демонстрирует рассматриваемый здесь пример.

Как пояснялось выше, внутренний класс имеет доступ ко всем элементам своего внешнего класса, но не наоборот. Члены внутреннего класса доступны только в области действия внутреннего класса и не могут быть использованы внешним классом. Как показано в приведенном ниже примере программы, переменная y объявлена как переменная экземпляра класса Inner. Поэтому она недоступна за пределами этого класса и не может использоваться в методе showy().

```

// Эта программа не подлежит компиляции
class Outer {
    int outer_x = 100;

    void test() {
        Inner inner = new Inner();
        inner.display();
    }

    // это внутренний класс
    class Inner {
        int y = 10; // y - локальная переменная класса Inner

        void display() {
            System.out.println("вывод: outer_x = " + outer_x);
        }
    }

    void showy() {
        System.out.println(y); // ошибка, здесь переменная
                               // y недоступна!
    }
}

class InnerClassDemo {
    public static void main(String args[]) {

```

```
        Outer outer = new Outer();
        outer.test();
    }
}
```

Выше основное внимание было уделено внутренним классам, определенным в качестве членов в области действия внешнего класса. Но внутренние классы можно определять и в области действия любого блока кода. Например, вложенный класс можно определить в блоке кода, относящегося к методу, или даже в теле цикла `for`:

```
// Определить внутренний класс в цикле for
class Outer {
    int outer_x = 100;

    void test() {
        for(int i=0; i<10; i++) {
            class Inner {
                void display() {
                    System.out.println("вывод: outer_x = " + outer_x);
                }
            }
            Inner inner = new Inner();
            inner.display();
        }
    }
}

class InnerClassDemo {
    public static void main(String args[]) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

Ниже приведен результат, выводимый этой версией программы.

```
вывод: outer_x = 100
вывод: outer_x = 100
вывод: outer_x = 100
вывод: outer_x = 100
вывод: outer_x = 100
вывод: outer_x = 100
вывод: outer_x = 100
вывод: outer_x = 100
вывод: outer_x = 100
вывод: outer_x = 100
```

Вложенные классы можно применять не во всех случаях. Тем не менее они особенно удобны для обработки событий. Мы еще вернемся к теме вложенных классов в главе 24, где представлены внутренние классы, которые можно использовать для упрощения кода, предназначенного для обработки определенных типов событий. Там же будут представлены и *анонимные* (т.е. безымянные) внутренние классы.

И последнее замечание: в первоначальной спецификации языка Java, относящейся к версии 1.0, вложенные классы не допускались. Они появились лишь в версии Java 1.1.

Краткий обзор класса String

Более подробно класс String будет рассмотрен в части II этой книги, а до тех пор уместно рассмотреть его хотя бы вкратце, поскольку символьные строки будут использоваться в ряде последующих примеров из части I. Класс String относится к числу наиболее часто употребляемых в библиотеке классов Java. И это, очевидно, происходит потому, что символьные строки являются очень важным средством программирования.

Во-первых, следует уяснить, что любая создаваемая символьная строка на самом деле является объектом класса String. И даже строковые константы в действительности являются объектами класса String. Например, в следующем операторе:

```
System.out.println("Это также объект String");
```

символьная строка "Это также объект String" является объектом класса String.

Во-вторых, объекты класса String являются неизменяемыми. Как только такой объект будет создан, его содержимое не подлежит изменению. На первый взгляд это может показаться серьезным ограничением, но на самом деле это не так по следующим причинам.

- Если требуется изменить символьную строку, то всегда можно создать новую символьную строку, содержащую все требующиеся изменения.
- В Java определены классы StringBuffer и StringBuilder, равноправные классу String и допускающие изменение символьных строк, что позволяет выполнять в Java все обычные операции с символьными строками. (Классы StringBuffer и StringBuilder будут описаны в части II.)

Символьные строки можно создавать разными способами. Самый простой из них — воспользоваться оператором вроде следующего:

```
String myString = "Это тестовая строка";
```

Как только объект класса String будет создан, им можно пользоваться везде, где допускаются символьные строки. Например, в следующей строке кода выводится содержимое объекта myString:

```
System.out.println(myString);
```

Для объектов класса String в Java определена одна операция +, предназначенная для сцепления двух символьных строк. Например, в результате приведенной ниже операции переменной myString присваивается символьная строка "Мне нравится Java".

```
String myString = "Мне" + " нравится " + "Java.";
```

Все упомянутые выше понятия демонстрируются в следующем примере программы:

```
// Продемонстрировать применение символьных строк
class StringDemo {
```

```
public static void main(String args[]) {  
    String strOb1 = "Первая строка";  
    String strOb2 = "Вторая строка";  
    String strOb3 = strOb1 + " и " + strOb2;  
  
    System.out.println(strOb1);  
    System.out.println(strOb2);  
    System.out.println(strOb3);  
}  
}
```

Эта программа выводит следующий результат:

```
Первая строка  
Вторая строка  
Первая строка и вторая строка
```

В классе `String` содержится ряд методов, которыми можно пользоваться, программируя на Java. Обсудим вкратце некоторые из них. Так, с помощью метода `equals()` можно проверить две символьные строки на равенство, а метод `length()` позволяет выяснить длину символьной строки. Вызывая метод `charAt()`, можно получить символ по заданному индексу. Ниже приведены общие формы этих трех методов.

```
boolean equals(вторая_строка)  
int length()  
char charAt(индекс)
```

Применение этих методов демонстрируется в следующем примере программы:

```
// Продемонстрировать некоторые методы из класса String  
class StringDemo2 {  
    public static void main(String args[]) {  
        String strOb1 = "Первая строка";  
        String strOb2 = "Вторая строка";  
        String strOb3 = strOb1;  
  
        System.out.println("Длина строки strOb1: " +  
                           strOb1.length());  
  
        System.out.println("Символ по индексу 3 в строке strOb1: " +  
                           strOb1.charAt(3));  
  
        if(strOb1.equals(strOb2))  
            System.out.println("strOb1 == strOb2");  
        else  
            System.out.println("strOb1 != strOb2");  
  
        if(strOb1.equals(strOb3))  
            System.out.println("strOb1 == strOb3");  
        else  
            System.out.println("strOb1 != strOb3");  
    }  
}
```

Эта программа выводит следующий результат:

```
Длина строки strOb1: 12  
Символ по индексу 3 в строке strOb1: s  
strOb1 != strOb2  
strOb1 == strOb3
```


Безусловно, подобно существованию массивов объектов любого другого типа, могут существовать и массивы символьных строк. Ниже приведен характерный тому пример.

```
// Продемонстрировать применение массивов объектов типа String
class StringDemo3 {
    public static void main(String args[]) {
        String str[] = { "один", "два", "три" };

        for(int i=0; i<str.length; i++)
            System.out.println("str[" + i + "]: " +
                               str[i]);
    }
}
```

Результат, выводимый этой программой, выглядит так:

```
str[0]: один
str[1]: два
str[2]: три
```

Как будет показано в следующем разделе, строковые массивы играют важную роль во многих программах на Java.

Применение аргументов командной строки

Иногда определенную информацию требуется передать программе во время ее запуска. Для этой цели служат аргументы командной строки для метода `main()`. *Аргумент командной строки* — это информация, которая во время запуска программы указывается в командной строке непосредственно после ее имени. Доступ к аргументам командной строки в программе на Java не представляет особого труда, поскольку они хранятся в виде символьных строк в массиве типа `String`, передаваемого методу `main()`. Первый аргумент командной строки хранится в элементе массива `args[0]`, второй — в элементе `args[1]` и т.д. В следующем примере программы выводятся все аргументы командной строки, с которыми она вызывается:

```
// Вывести все аргументы командной строки
class CommandLine {
    public static void main(String args[]) {
        for(int i=0; i<args.length; i++)
            System.out.println("args[" + i + "]: " +
                               args[i]);
    }
}
```

Попробуйте выполнить эту программу, введя следующую команду в командной строке:

```
java CommandLine this is a test 100 -1
```

В итоге будет выведен следующий результат:

```
args[0]: this
args[1]: is
args[2]: a
args[3]: test
```

```
args[4]: 100  
args[5]: -1
```

Помните! Все аргументы командной строки передаются в виде символьных строк. Числовые значения следует вручную преобразовать в их внутренние представления, как поясняется в главе 17.

Аргументы переменной длины

В версии JDK 5 было введено новое языковое средство, упрощающее создание методов, принимающих переменное количество аргументов. Оно получило название *varargs* (*variable-length arguments* — аргументы переменной длины). Метод, который принимает переменное количество аргументов, называется *методом с аргументами переменной длины*.

Ситуации, когда методу требуется передать переменное количество аргументов, встречаются не так уж редко. Например, метод, устанавливающий соединение с Интернетом, может принимать имя пользователя, пароль, имя файла, сетевой протокол и прочие данные, но в то же время выбирать значения, задаваемые по умолчанию, если какие-нибудь из этих данных опущены. В подобной ситуации было бы удобнее передать только те аргументы, для которых неприменимы значения, задаваемые по умолчанию. Еще одним примером служит метод `printf()`, входящий в состав библиотеки ввода-вывода в Java. Как будет показано в главе 20, этот метод принимает переменное количество аргументов, которые форматируются, а затем выводятся.

До версии J2SE 5 обработка аргументов переменной длины могла выполняться двумя способами, ни один из которых не был особенно удобным. Во-первых, если максимальное количество аргументов было небольшим и известным, можно было создавать перегружаемые варианты метода — по одному для каждого из возможных способов вызова метода. И хотя такой способ вполне работоспособен, он пригоден только в редких случаях.

Во-вторых, когда максимальное количество возможных аргументов было большим или неизвестным, применялся подход, при котором аргументы сначала размещались в массиве, а затем массив передавался методу. Такой подход демонстрируется в следующей программе:

```
// Использовать массив для передачи методу переменного  
// количества аргументов. Это старый подход к обработке  
// аргументов переменной длины  
class PassArray {  
    static void vaTest(int v[]) {  
        System.out.print("Количество аргументов: " + v.length +  
                           " Содержимое: ");  
  
        for(int x : v)  
            System.out.print(x + " ");  
        System.out.println();  
    }  
  
    public static void main(String args[])  
    {  
        // Обратите внимание на порядок создания массива
```

```

        // для хранения аргументов.
        int n1[] = { 10 };
        int n2[] = { 1, 2, 3 };
        int n3[] = { };
        vaTest(n1); // 1 аргумент
        vaTest(n2); // 3 аргумента
        vaTest(n3); // без аргументов
    }
}

```

Эта программа выводит следующий результат:

```

Количество аргументов: 1 Содержимое: 10
Количество аргументов: 3 Содержимое: 1 2 3
Количество аргументов: 0 Содержимое:

```

В данной программе аргументы передаются методу `vaTest()` через массив `v`. Этот старый подход к обработке аргументов переменной длины позволяет методу `vaTest()` принимать любое количество аргументов. Но он требует, чтобы эти аргументы были вручную размещены в массиве до вызова метода `vaTest()`. Создание массива при каждом вызове метода `vaTest()` — задача не только трудоемкая, но и чреватая ошибками. Методы с аргументами переменной длины обеспечивают более простой и эффективный подход к обработке таких аргументов.

Для указания аргументов переменной длины служат три точки (`...`). В приведенном ниже примере показано, каким образом метод `vaTest()` можно объявить с аргументами переменной длины.

```
static void vaTest(int ... v) {
```

В этой синтаксической конструкции компилятору предписывается, что метод `vaTest()` может вызываться без аргументов или с несколькими аргументами. В итоге массив `v` неявно объявляется как массив типа `int[]`. Таким образом, в теле метода `vaTest()` доступ к массиву `v` осуществляется с помощью синтаксиса обычного массива. Ниже приведена переделанная версия предыдущего примера программы, где применяется метод с аргументами переменной длины. По результату своего выполнения она ничем не отличается от предыдущей версии.

```

// Продемонстрировать применение аргументов переменной длины
class VarArgs {
    // теперь метод vaTest() объявляется с аргументами переменной длины
    static void vaTest(int ... v) {
        System.out.print("Количество аргументов: " + v.length +
            " Содержимое: ");

        for(int x : v)
            System.out.print(x + " ");

        System.out.println();
    }

    public static void main(String args[])
    {
        // Обратите внимание на возможные способы вызова
        // метода vaTest() с аргументами переменной длины
        vaTest(10);           // 1 аргумент
        vaTest(1, 2, 3);      // 3 аргумента
    }
}

```

```

    vaTest();          // без аргументов
}
}

```

Отметим две важные особенности этой версии программы. Во-первых, как отмечалось ранее, в теле метода `vaTest()` переменная `v` действует как массив, поскольку она действительно *является* массивом. Синтаксическая конструкция `...` просто указывает компилятору, что в данном методе предполагается использовать переменное количество аргументов и что эти аргументы будут храниться в массиве, на который ссылается переменная `v`. Во-вторых, метод `vaTest()` вызывается в методе `main()` с разным количеством аргументов, в том числе и совсем без них. Аргументы автоматически размещаются в массиве и передаются переменной `v`. Если же аргументы отсутствуют, длина этого массива равна нулю.

Наряду с параметром переменной длины у метода могут быть и “обычные” параметры. Но параметр переменной длины должен быть последним среди всех параметров, объявляемых в методе. Например, следующее объявление метода вполне допустимо:

```
int doIt(int a, int b, double c, int ... vals) {
```

В данном случае первые три аргумента, указанные в объявлении метода `doIt()`, соответствуют первым трем параметрам. А все остальные аргументы считаются принадлежащими параметру `vals`. Напомним, что параметр с переменным количеством аргументов должен быть последним. Например, следующее объявление сделано неправильно:

```
int doIt(int a, int b, double c,
        int ... vals, boolean stopFlag) { // ОШИБКА!
```

В данном примере предпринимается попытка объявить обычный параметр после параметра с переменным количеством аргументов, что недопустимо. Существует еще одно ограничение, о котором следует знать: метод должен содержать только один параметр с переменным количеством аргументов. Например, приведенное ниже объявление также неверно.

```
int doIt(int a, int b, double c,
        int ... vals, double ... morevals) { // ОШИБКА!
```

Попытка объявить второй параметр с переменным количеством аргументов недопустима. Ниже приведена измененная версия метода `vaTest()`, который принимает как обычный аргумент, так и аргументы переменной длины.

```

// Использовать аргументы переменной длины вместе
// со стандартными аргументами
class VarArgs2 {

    // В данном примере msg – обычный параметр,
    // а v – параметр переменной длины
    static void vaTest(String msg, int ... v) {
        System.out.print(msg + v.length +
            " Содержимое: ");

        for(int x : v)
            System.out.print(x + " ");

        System.out.println();
    }
}

```

```

    }
    public static void main(String args[])
    {
        vaTest("Один параметр переменной длины: ", 10);
        vaTest("Три параметра переменной длины: ", 1, 2, 3);
        vaTest("Без параметров переменной длины: ");
    }
}

```

Результат, выводимый этой программой, выглядит следующим образом:

```

Один параметр переменной длины: 1 Содержимое: 10
Три параметра переменной длины: 3 Содержимое: 1 2 3
Без параметров переменной длины: 0 Содержимое:

```

Перегрузка методов с аргументами переменной длины

Метод, принимающий аргументы переменной длины, можно перегружать. В приведенном ниже примере программы метод `vaTest()` перегружается трижды.

```

// Аргументы переменной длины и перегрузка
class VarArgs3 {

    static void vaTest(int ... v) {
        System.out.print("vaTest(int ...): " +
            "Количество аргументов: " + v.length +
            " Содержимое: ");

        for(int x : v)
            System.out.print(x + " ");

        System.out.println();
    }

    static void vaTest(boolean ... v) {
        System.out.print("vaTest(boolean ...) " +
            "Количество аргументов: " + v.length +
            " Содержимое: ");

        for(boolean x : v)
            System.out.print(x + " ");

        System.out.println();
    }

    static void vaTest(String msg, int ... v) {
        System.out.print("vaTest(String, int ...): " +
            msg + v.length +
            " Содержимое: ");

        for(int x : v)
            System.out.print(x + " ");

        System.out.println();
    }

    public static void main(String args[])
    {
        vaTest(1, 2, 3);
        vaTest("Проверка: ", 10, 20);
    }
}

```

```

        vaTest(true, false, false);
    }
}

```

Эта программа выводит следующий результат:

```

vaTest(int ...): Количество аргументов: 3 Содержимое: 1 2 3
vaTest(String, int ...): Проверка: 2 Содержимое: 10 20
vaTest(boolean ...) Количество аргументов: 3 Содержимое: true false false

```

В приведенном выше примере программы демонстрируются два возможных способа перегрузки метода с аргументами переменной длины. Первый способ состоит в том, что у параметра данного метода с переменным количеством аргументов могут быть разные типы. Именно это имеет место в вариантах метода `vaRest(int...)` и `vaTest(boolean...)`. Напомним, что языковая конструкция `...` вынуждает компилятор обрабатывать параметр как массив заданного типа. Поэтому, используя разные типы аргументов переменной длины, можно выполнять перегрузку методов с переменным количеством аргументов таким же образом, как и обычных методов с массивом разнотипных параметров. В этом случае исполняющая система Java использует отличие в типах аргументов для выбора нужного варианта перегружаемого метода.

Второй способ перегрузки метода с аргументами переменной длины состоит в том, чтобы добавить один или несколько обычных параметров. Именно это и было сделано при объявлении метода `vaTest(String, int...)`. В данном случае для выбора нужного варианта метода исполняющая система Java использует не только количество аргументов, но и их тип.

На заметку! Метод, поддерживающий переменное количество аргументов, может быть также перегружен методом, который не поддерживает такую возможность. Так, в приведенном выше примере программы метод `vaTest()` может быть перегружен методом `vaTest(int x)`. Этот специализированный вариант вызывается только при наличии аргумента `int`. Если же передаются два или более аргумента типа `int`, то будет выбран вариант метода `vaTest(int...v)` с аргументами переменной длины.

Аргументы переменной длины и неоднозначность

При перегрузке метода, принимающего аргументы переменной длины, могут происходить непредвиденные ошибки. Они связаны с неоднозначностью, которая может возникать при вызове перегружаемого метода с аргументами переменной длины. Рассмотрим в качестве примера следующую программу:

```

// Аргументы переменной длины, перегрузка и неоднозначность
//
// Эта программа содержит ошибку, и поэтому не может быть скомпилирована!
class VarArgs4 {

    static void vaTest(int ... v) {
        System.out.print("vaTest(int ...): " +
            "Количество аргументов: " + v.length +
            " Содержимое: ");

        for(int x : v)

```

```

        System.out.print(x + " ");

        System.out.println();
    }

    static void vaTest(boolean ... v) {
        System.out.print("vaTest(boolean ...) " +
            "Количество аргументов: " + v.length +
            " Содержимое: ");

        for(boolean x : v)
            System.out.print(x + " ");

        System.out.println();
    }

    public static void main(String args[])
    {
        vaTest(1, 2, 3);           // Верно!
        vaTest(true, false, false); // Верно!
        vaTest();                  // Ошибка: неоднозначность!
    }
}

```

В этой программе перегрузка метода `vaTest()` задается вполне корректно. Тем не менее скомпилировать ее не удастся из-за следующего вызова:

```
vaTest(); // ОШИБКА: неоднозначность!
```

Параметр с переменным количеством аргументов может быть пустым, поэтому этот вызов может быть преобразован в вызов метода `vaTest(int...)` или `vaTest(boolean...)`. А поскольку вполне допустимы оба варианта, то данный вызов принципиально неоднозначен.

Рассмотрим еще один пример неоднозначности. Приведенные ниже перегружаемые варианты метода `vaTest()` изначально неоднозначны, несмотря на то, что один из них принимает обычный параметр.

```
static void vaTest(int ... v) { // ...
static void vaTest(int n, int ... v) { // ...

```

Несмотря на то что оба списка параметров метода `vaTest()` отличаются, компилятор не в состоянии разрешить следующий вызов:

```
vaTest(1)
```

Должен ли он быть преобразован в вызов метода `vaTest(int...)` с аргументами переменной длины или же в вызов метода `vaTest(int, int...)` без аргументов переменной длины? Компилятор не в состоянии разрешить этот вопрос. Таким образом, возникает неоднозначная ситуация.

Из-за ошибок неоднозначности, подобных описанным выше, иногда приходится отказываться от перегрузки и просто использовать один и тот же метод под двумя разными именами. Кроме того, ошибки неоднозначности порой служат признаком принципиальных изъянов в программе, которые можно устранить, тщательно проработав решения поставленной задачи.

Одним из основополагающих принципов объектно-ориентированного программирования является наследование, поскольку оно позволяет создавать иерархические классификации. Используя наследование, можно создать класс, который определяет характеристики, общие для набора связанных элементов. Затем этот общий класс может наследоваться другими, более специализированными классами, каждый из которых будет добавлять свои особые характеристики. В терминологии Java наследуемый класс называется *суперклассом*, а наследующий класс — *подклассом*. Следовательно, подкласс — это специализированная версия суперкласса. Он наследует все члены, определенные в суперклассе, добавляя к ним собственные, особые элементы.

Основы наследования

Чтобы наследовать класс, достаточно ввести определение одного класса в другой, используя ключевое слово `extends`. Для иллюстрации принципа наследования обратимся к краткому примеру. В приведенной ниже программе создаются суперкласс А и подкласс В. Обратите внимание на использование ключевого слова `extends` для создания подкласса, производного от класса А.

```
// Простой пример наследования

// создать суперкласс
class A {
    int i, j;

    void showij() {
        System.out.println("i и j: " + i + " " + j);
    }
}

// создать подкласс путем расширения класса А
class B extends A {
    int k;

    void showk() {
        System.out.println("k: " + k);
    }
    void sum() {
```



```

        System.out.println("i+j+k: " + (i+j+k));
    }
}

class SimpleInheritance {
    public static void main(String args[]) {
        A superOb = new A();
        B subOb = new B();

        // суперкласс может использоваться самостоятельно
        superOb.i = 10;
        superOb.j = 20;
        System.out.println("Содержимое объекта superOb: ");
        superOb.showij();
        System.out.println();

        /* Подкласс имеет доступ ко всем открытым членам
           своего суперкласса. */
        subOb.i = 7;
        subOb.j = 8;
        subOb.k = 9;
        System.out.println("Содержимое объекта subOb: ");
        subOb.showij();
        subOb.showk();

        System.out.println();
        System.out.println("Сумма i, j и k в объекте subOb:");
        subOb.sum();
    }
}

```

Эта программа выводит следующий результат:

```

Содержимое объекта superOb:
i и j: 10 20
Содержимое объекта subOb:
i и j: 7 8
k: 9
Сумма i, j и k в объекте subOb:
i+j+k: 24

```

Как видите, подкласс B включает в себя все члены своего суперкласса A. Именно поэтому объект subOb имеет доступ к переменным i и j и может вызывать метод showij(). Кроме того, в методе sum() возможна непосредственная ссылка на переменные i и j, как если бы они были частью класса B.

Несмотря на то что класс A является суперклассом для класса B, он в то же время остается полностью независимым и самостоятельным классом. То, что один класс является суперклассом для другого класса, совсем не исключает возможность его самостоятельного использования. Более того, один подкласс может быть суперклассом другого подкласса.

Ниже приведена общая форма объявления класса, который наследуется от суперкласса.

```

class имя_подкласса extends имя_суперкласса {
    // тело класса
}

```

Для каждого создаваемого подкласса можно указать только один суперкласс. В Java не поддерживается наследование нескольких суперклассов в одном подклассе.

се. Как отмечалось ранее, можно создать иерархию наследования, в которой один подкласс становится суперклассом другого подкласса. Но ни один из классов не может стать суперклассом для самого себя.

Доступ к членам класса и наследование

Несмотря на то что подкласс включает в себя все члены своего суперкласса, он не может иметь доступ к тем членам суперкласса, которые объявлены как `private`. Рассмотрим в качестве примера следующую простую иерархию классов:

```
/* В иерархии классов закрытые члены остаются закрытыми
   в пределах своего класса.

   Эта программа содержит ошибку, и поэтому
   скомпилировать ее не удастся.
*/

// создать суперкласс
class A {
    int i;           // этот член открыт по умолчанию,
    private int j;   // а этот член закрыт в классе A

    void setij(int x, int y) {
        i = x;
        j = y;
    }
}

// Член j класса A в этом классе недоступен
class B extends A {
    int total;

    void sum() {
        total = i + j; // ОШИБКА: член j в этом классе недоступен
    }
}

class Access {
    public static void main(String args[]) {
        B subOb = new B();

        subOb.setij(10, 12);

        subOb.sum();
        System.out.println("Сумма равна " + subOb.total);
    }
}
```

Скомпилировать эту программу не удастся, потому что использование переменной `j` из класса `A` в методе `sum()` из класса `B` приводит к нарушению правил доступа. Но поскольку переменная `j` объявлена в классе `A` как `private`, то она доступна только другим членам ее собственного класса. А подклассы не имеют к ней доступа.

Помните! Член класса, который объявлен закрытым, останется закрытым в пределах своего класса. Он недоступен для любого кода за пределами его класса, в том числе для подклассов.

Более практический пример

Рассмотрим более практический пример, который поможет лучше продемонстрировать возможности наследования. В этом примере последняя версия класса `Box` из предыдущей главы расширена четвертым компонентом `weight` (вес). Таким образом, новая версия класса будет `Box` содержать поля ширины, высоты, глубины и веса параллелепипеда.

```
// В этой программе наследование применяется
// для расширения класса Box
class Box {
    double width;
    double height;
    double depth;

    // сконструировать клон объекта
    Box(Box ob) { // передать объект конструктору
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // конструктор, применяемый при указании всех размеров
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // конструктор, применяемый в отсутствие размеров
    Box() {
        width = -1; // значение -1 служит для обозначения
        height = -1; // неинициализированного
        depth = -1; // параллелепипеда
    }

    // конструктор, применяемый при создании куба
    Box(double len) {
        width = height = depth = len;
    }

    // рассчитать и вернуть объем
    double volume() {
        return width * height * depth;
    }
}

// расширить класс Box, включив в него поле веса
class BoxWeight extends Box {
    double weight; // вес параллелепипеда

    // конструктор BoxWeight()
    BoxWeight(double w, double h, double d, double m) {
        width = w;
        height = h;
        depth = d;
        weight = m;
    }
}

class DemoBoxWeight {
    public static void main(String args[]) {
```

```
BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
double vol;

vol = mybox1.volume();
System.out.println("Объем mybox1 равен " + vol);
System.out.println("Вес mybox1 равен " + mybox1.weight);
System.out.println();

vol = mybox2.volume();
System.out.println("Объем mybox2 равен " + vol);
System.out.println("Вес mybox2 равен " + mybox2.weight);
}
}
```

Эта программа выводит следующий результат:

```
Объем mybox1 равен 3000.0
Вес mybox1 равен 34.3
Объем mybox2 равен 24.0
Вес mybox2 равен 0.076
```

Класс `BoxWeight` наследует все характеристики класса `Box` и добавляет к ним компонент `weight`. Классу `BoxWeight` не нужно воссоздавать все характеристики класса `Box`. Для этого достаточно расширить класс `Box`, исходя из конкретных целей.

Главное преимущество наследования состоит в том, что, как только будет создан суперкласс, определяющий общие свойства ряда объектов, его можно использовать для разработки любого количества более специализированных классов. Каждый подкласс может точно определять свою собственную классификацию. Например, следующий класс наследует характеристики класса `Box` и добавляет к ним свойство цвета параллелепипеда:

```
// Этот класс расширяет класс Box, включая в него свойство цвета
class ColorBox extends Box {
    int color; // цвет параллелепипеда

    ColorBox(double w, double h, double d, int c) {
        width = w;
        height = h;
        depth = d;
        color = c;
    }
}
```

Напомним, как только суперкласс, который определяет общие свойства объекта, будет создан, он может наследоваться для разработки специализированных классов. Каждый подкласс добавляет собственные особые свойства. В этом и состоит сущность наследования.

Переменная суперкласса может ссылаться на объект подкласса

Ссылочной переменной из суперкласса может быть присвоена ссылка на любой подкласс, производный от этого суперкласса. Этот аспект наследования может оказаться удобным во многих случаях. Рассмотрим следующий пример:

```

class RefDemo {
    public static void main(String args[]) {
        BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
        Box plainbox = new Box();
        double vol;

        vol = weightbox.volume();
        System.out.println("Объем weightbox равен " + vol);
        System.out.println("Вес weightbox равен " +
                           weightbox.weight);
        System.out.println();

        // присвоить ссылке на объект BoxWeight ссылки на объект Box
        plainbox = weightbox;
        vol = plainbox.volume(); // Верно, т.к. метод volume()
                                // определен в классе Box
        System.out.println("Объем plainbox равен " + vol);

        /* Следующий оператор ошибочен, поскольку член plainbox
           не определяет член weight. */
        // System.out.println("Вес plainbox равен " + plainbox.weight);
    }
}

```

В данном примере член `weightbox` содержит ссылку на объекты класса `BoxWeight`, а член `rainbox` — ссылку на объекты класса `Box`. А поскольку `BoxWeight` — это подкласс, производный от класса `Box`, то его члену `rainbox` можно присвоить ссылку на объект `weightbox`.

Следует иметь в виду, что доступные члены класса определяются типом ссылочной переменной, а не типом объекта, на который она ссылается. Это означает, что если ссылочной переменной из суперкласса присваивается ссылка на объект подкласса, то доступ предоставляется только к указанным в ней частям объекта, определяемого в суперклассе. Именно поэтому у объекта `plainbox` нет доступа к полю `weight` даже в том случае, когда он ссылается на объект класса `BoxWeight`. Это становится понятным по зрелом размышлении, ведь суперклассу неизвестно, что именно в него добавляет подкласс. Поэтому последняя строка кода в рассматриваемом здесь примере кода закомментирована. Ссылка на объект класса `Box` не предоставляет доступ к полю `weight`, поскольку оно не определено в классе `Box`.

Все сказанное выше может показаться не совсем понятным. Тем не менее ему можно найти ряд практических применений, два из которых рассматриваются в последующих разделах.

Ключевое слово `super`

В предыдущих примерах классы, производные от класса `Box`, были реализованы не столь эффективно и надежно, как следовало бы. Например, конструктор `BoxWeight()` явно инициализирует поля `width`, `height` и `depth` из класса `Box`. Это не только ведет к дублированию кода суперкласса, что весьма неэффективно, но и предполагает наличие у подкласса доступа к этим членам суперкласса. Но иногда приходится создавать суперкласс, подробности реализации которого доступны только для него самого (т.е. класс с закрытыми членами). В этом случае

подкласс не в состоянии непосредственно обращаться к этим переменным или инициализировать их. Инкапсуляция — один из главных принципов ООП, поэтому и не удивительно, что в Java предлагается свое решение этой проблемы. Всякий раз, когда подклассу требуется сослаться на его непосредственный суперкласс, это можно сделать с помощью ключевого слова `super`.

У ключевого слова `super` имеются две общие формы. Первая форма служит для вызова конструктора суперкласса, вторая — для обращения к члену суперкласса, скрываемому членом подкласса. Рассмотрим обе эти формы подробнее.

Вызов конструкторов суперкласса с помощью ключевого слова `super`

Из подкласса можно вызывать конструктор, определенный в его суперклассе, используя следующую форму ключевого слова `super`:

```
super (список_аргументов) ;
```

где `список_аргументов` определяет любые аргументы, требующиеся конструктору в суперклассе. Вызов метода `super()` всегда должен быть первым оператором, выполняемым в конструкторе подкласса.

В качестве примера, демонстрирующего применение метода `super()`, рассмотрим следующую усовершенствованную версию класса `BoxWeight`:

```
// Теперь в классе BoxWeight ключевое слово super используется
// для инициализации собственных свойств объекта типа Box
class BoxWeight extends Box {
    double weight; // вес параллелепипеда

    // инициализировать поля width, height и depth
    // с помощью метода super()
    BoxWeight(double w, double h, double d, double m) {
        super(w, h, d); // вызвать конструктор суперкласса
        weight = m;
    }
}
```

В данном примере метод `super()` вызывается с аргументами `w`, `h` и `d` из метода `BoxWeight()`. Это приводит к вызову конструктора `Box()`, в котором поля `width`, `height` и `depth` инициализируются передаваемыми ему значениями соответствующих параметров. Теперь эти значения не инициализируются в самом классе `BoxWeight`. В нем остается только инициализировать его собственное поле `weight`. В итоге эти поля могут, если требуется, оставаться закрытыми в классе `Box`.

В приведенном выше примере метод `super()` вызывался с тремя аргументами. Конструкторы могут быть перегружаемыми, и поэтому метод `super()` можно вызывать, используя любую форму, определяемую в суперклассе. Выполнен будет тот конструктор, который соответствует указанным аргументам. В качестве примера ниже приведена полная реализация класса `BoxWeight`, предоставляющая конструкторы для разных способов создания параллелепипедов. В каждом случае метод `super()` вызывается с соответствующими аргументами. Обратите внимание на то, что члены `width`, `height` и `depth` объявлены в классе `Box` как закрытые.

```
// Полная реализация класса BoxWeight
class Box {
    private double width;
    private double height;
    private double depth;

    // сконструировать клон объекта
    Box(Box ob) { // передать объект конструктору
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // конструктор, применяемый при указании всех размеров
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // конструктор, применяемый в отсутствие размеров
    Box() {
        width = -1; // значение -1 служит для обозначения
        height = -1; // неинициализированного
        depth = -1; // параллелепипеда
    }

    // конструктор, применяемый при создании куба
    Box(double len) {
        width = height = depth = len;
    }

    // рассчитать и возвратить объем
    double volume() {
        return width * height * depth;
    }
}

// Теперь в классе BoxWeight полностью реализованы все конструкторы
class BoxWeight extends Box {
    double weight; // вес параллелепипеда

    // сконструировать клон объекта
    BoxWeight(BoxWeight ob) { // передать объект конструктору
        super(ob);
        weight = ob.weight;
    }

    // конструктор, применяемый при указании всех параметров
    BoxWeight(double w, double h, double d, double m) {
        super(w, h, d); // вызвать конструктор из суперкласса
        weight = m;
    }

    // конструктор, применяемый по умолчанию
    BoxWeight() {
        super();
        weight = -1;
    }

    // конструктор, применяемый при создании куба
```

```
BoxWeight(double len, double m) {
    super(len);
    weight = m;
}
}
class DemoSuper {
    public static void main(String args[]) {
        BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
        BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
        BoxWeight mybox3 = new BoxWeight(); // по умолчанию
        BoxWeight mycube = new BoxWeight(3, 2);
        BoxWeight myclone = new BoxWeight(mybox1);
        double vol;

        vol = mybox1.volume();
        System.out.println("Объем mybox1 равен " + vol);
        System.out.println("Вес mybox1 равен " + mybox1.weight);
        System.out.println();

        vol = mybox2.volume();
        System.out.println("Объем mybox2 равен " + vol);
        System.out.println("Вес mybox2 равен " + mybox2.weight);
        System.out.println();

        vol = mybox3.volume();
        System.out.println("Объем mybox3 равен " + vol);
        System.out.println("Вес mybox3 равен " + mybox3.weight);
        System.out.println();

        vol = myclone.volume();
        System.out.println("Объем myclone равен " + vol);
        System.out.println("Вес myclone равен " + myclone.weight);
        System.out.println();

        vol = mycube.volume();
        System.out.println("Объем mycube равен " + vol);
        System.out.println("Вес mycube равен " + mycube.weight);
        System.out.println();
    }
}
```

Эта программа выводит следующий результат:

```
Объем mybox1 равен 3000.0
Вес mybox1 равен 34.3
Объем mybox2 равен 24.0
Вес mybox2 равен 0.076
Объем mybox3 равен -1.0
Вес mybox3 равен -1.0
Объем myclone равен 3000.0
Вес myclone равен 34.3
Объем mycube равен 27.0
Вес mycube равен 2.0
```

Обратите особое внимание на следующий конструктор из класса BoxWeight:

```
// сконструировать клон объекта
BoxWeight(BoxWeight ob) { // передать объект конструктору
    super(ob);
    weight = ob.weight;
}
```


Обратите также внимание на то, что методу `super()` передается объект типа `BoxWeight`, а не `Box`. Но это все равно приводит к вызову конструктора `Box (Box . ob)`. Как отмечалось ранее, переменную из суперкласса можно использовать для ссылки на любой объект, унаследованный от этого класса. Таким образом, объект класса `BoxWeight` можно передать конструктору `Box ()`. Разумеется, классу `Box` будут доступны только его собственные члены.

Рассмотрим основные принципы, положенные в основу метода `super()`. Когда метод `super()` вызывается из подкласса, вызывается конструктор его непосредственного суперкласса. Таким образом, метод `super()` всегда обращается к суперклассу, находящемуся в иерархии непосредственно над вызывающим классом. Это справедливо даже для многоуровневой иерархии. Кроме того, вызов метода `super()` должен быть всегда в первом операторе, выполняемом в теле конструктора подкласса.

Другое применение ключевого слова `super`

Вторая форма ключевого слова `super` действует подобно ключевому слову `this`, за исключением того, что ссылка всегда делается на суперкласс того подкласса, в котором используется это ключевое слово. В общем виде эта форма применения ключевого слова `super` выглядит следующим образом:

super. член

где *член* может быть методом или переменной экземпляра. Вторая форма применения ключевого слова `super` наиболее пригодна в тех случаях, когда имена членов подкласса скрывают члены суперкласса с такими же именами. Рассмотрим в качестве примера следующую простую иерархию классов:

```
// Использовать ключевое слово super с целью предотвратить сокрытие имен
class A {
    int i;
}

// создать подкласс путем расширения класса A
class B extends A {
    int i;    // этот член i скрывает член i из класса A

    B(int a, int b) {
        super.i = a; // член i из класса A
        i = b;      // член i из класса B
    }

    void show() {
        System.out.println("Член i в суперклассе: " + super.i);
        System.out.println("Член i в подклассе: " + i);
    }
}

class UseSuper {
    public static void main(String args[]) {
        B subOb = new B(1, 2);

        subOb.show();
    }
}
```

Эта программа выводит следующий результат:

```
Член i в суперклассе: 1
Член i в подклассе: 2
```

Хотя переменная экземпляра *i* из класса *B* скрывает переменную экземпляра *i* из класса *A*, ключевое слово `super` позволяет получить доступ к переменной *i*, определяемой в суперклассе. Как будет показано далее, ключевое слово `super` можно использовать также для вызова методов, которые скрываются в подклассе.

Создание многоуровневой иерархии

В приведенных до сих пор примерах использовались простые иерархии классов, которые состояли только из суперкласса и подкласса. Но ничто не мешает строить иерархии, состоящие из любого количества уровней наследования. Как отмечалось ранее, подкласс вполне допустимо использовать в качестве суперкласса другого подкласса. Например, класс *C* может быть подклассом, производным от класса *B*, который, в свою очередь, является производным от класса *A*. В подобных случаях каждый подкласс наследует все характеристики всех его суперклассов. В приведенном выше примере класс *C* наследует все характеристики классов *B* и *A*. В качестве примера построения многоуровневой иерархии рассмотрим еще одну программу, в которой подкласс `BoxWeight` служит в качестве суперкласса для создания подкласса `Shipment`. Класс `Shipment` наследует все характеристики классов `BoxWeight` и `Box` и добавляет к ним поле `cost`, которое содержит стоимость доставки посылки.

```
// Расширение класса BoxWeight включением в него
// поля стоимости доставки

// создать сначала класс Box
class Box {
    private double width;
    private double height;
    private double depth;

    // сконструировать клон объекта
    Box(Box ob) { // передать объект конструктору
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }

    // конструктор, применяемый при указании всех размеров
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }

    // конструктор, применяемый в отсутствие размеров
    Box() {
        width = -1; // значение -1 служит для обозначения
        height = -1; // неинициализированного
        depth = -1; // параллелепипеда
    }
}
```

```
// конструктор, применяемый при создании куба
Box(double len) {
    width = height = depth = len;
}

// рассчитать и вернуть объем
double volume() {
    return width * height * depth;
}
}

// добавить поле веса
class BoxWeight extends Box {
    double weight; // вес параллелепипеда

    // сконструировать клон объекта
    BoxWeight(BoxWeight ob) { // передать объект конструктору
        super(ob);
        weight = ob.weight;
    }

    // конструктор, применяемый при указании всех параметров
    BoxWeight(double w, double h, double d, double m) {
        super(w, h, d); // вызвать конструктор суперкласса
        weight = m;
    }

    // конструктор, применяемый по умолчанию
    BoxWeight() {
        super();
        weight = -1;
    }

    // конструктор, применяемый при создании куба
    BoxWeight(double len, double m) {
        super(len);
        weight = m;
    }
}

// добавить поле стоимости доставки
class Shipment extends BoxWeight {
    double cost;

    // сконструировать клон объекта
    Shipment(Shipment ob) { // передать объект конструктору
        super(ob);
        cost = ob.cost;
    }

    // конструктор, используемый при указании всех параметров
    Shipment(double w, double h, double d,
        double m, double c) {
        super(w, h, d, m); // вызвать конструктор суперкласса
        cost = c;
    }

    // конструктор, применяемый по умолчанию
    Shipment() {
        super();
        cost = -1;
    }
}
```

```
// конструктор, применяемый при создании куба
Shipment(double len, double m, double c) {
    super(len, m);
    cost = c;
}

class DemoShipment {
    public static void main(String args[]) {
        Shipment shipment1 =
            new Shipment(10, 20, 15, 10, 3.41);
        Shipment shipment2 =
            new Shipment(2, 3, 4, 0.76, 1.28);

        double vol;

        vol = shipment1.volume();
        System.out.println("Объем shipment1 равен " + vol);
        System.out.println("Вес shipment1 равен "
            + shipment1.weight);
        System.out.println("Стоимость доставки: $" + shipment1.cost);
        System.out.println();
        vol = shipment2.volume();
        System.out.println("Объем shipment2 равен " + vol);
        System.out.println("Вес shipment2 равен "
            + shipment2.weight);
        System.out.println("Стоимость доставки: $" + shipment2.cost);
    }
}
```

Результат, выводимый этой программой, выглядит следующим образом:

```
Объем shipment1 равен 3000.0
Вес shipment1 равен 10.0
Стоимость доставки: $3.41
Объем shipment2 равен 24.0
Вес shipment2 равен 0.76
Стоимость доставки: $1.28
```

Благодаря наследованию в классе `Shipment` могут использоваться ранее определенные классы `Box` и `BoxWeight`, добавляя только те дополнительные данные, которые требуются для его собственного специализированного применения. В этом и состоит одно из ценных свойств наследования. Оно позволяет использовать код повторно.

Приведенный выше пример демонстрирует следующий важный аспект: метод `super()` всегда ссылается на конструктор ближайшего по иерархии суперкласса. В методе `super()` из класса `Shipment` вызывается конструктор класса `BoxWeight`. А в методе `super()` из класса `BoxWeight` вызывается конструктор класса `Box`. Если в иерархии классов требуется передать параметры конструктору суперкласса, то все подклассы должны передавать эти параметры вверх по иерархии. Данное утверждение справедливо независимо от того, нуждается ли подкласс в собственных параметрах.

На заметку! В приведенном выше примере программы вся иерархия классов, включая `Box`, `BoxWeight` и `Shipment`, находится в одном файле. Это сделано только ради удобства. В Java все три класса могли бы быть размещены в отдельных файлах и скомпилированы независимо друг от друга. На самом деле использование отдельных файлов при создании иерархий классов считается скорее правилом, а не исключением.

Порядок вызова конструкторов

В каком порядке вызываются конструкторы классов, образующих иерархию при ее создании? Например, какой конструктор вызывается раньше: `A()` или `B()`, если `B` — это подкласс, а `A` — суперкласс? В иерархии классов конструкторы вызываются в порядке наследования, начиная с суперкласса и кончая подклассом. Более того, этот порядок остается неизменным независимо от того, используется форма `super()` или нет, поскольку вызов метода `super()` должен быть в первом операторе, выполняемом в конструкторе подкласса. Если метод `super()` не вызывается, то используется конструктор по умолчанию или же конструктор без параметров из каждого суперкласса. В следующем примере программы демонстрируется порядок вызова и выполнения конструкторов:

```
// Продемонстрировать порядок вызова конструкторов

// создать суперкласс

class A {
    A() {
        System.out.println("В конструкторе A.");
    }
}

// создать подкласс путем расширения класса A
class B extends A {
    B() {
        System.out.println("В конструкторе B.");
    }
}

// создать еще один подкласс путем расширения класса B
class C extends B {
    C() {
        System.out.println("В конструкторе C.");
    }
}

class CallingCons {
    public static void main(String args[]) {
        C c = new C();
    }
}
```

Эта программа выводит следующий результат:

```
Внутри конструктора A
Внутри конструктора B
Внутри конструктора C
```

Как видите, конструкторы вызываются в порядке наследования. По зрелом размышлении становится ясно, что выполнение конструкторов в порядке наследования имеет определенный смысл. Суперклассу ничего неизвестно о своих подклассах, и поэтому любая инициализация должна быть выполнена в нем совершенно независимо от любой инициализации, выполняемой подклассом. Следовательно, она должна выполняться в первую очередь.

Переопределение методов

Если в иерархии классов совпадают имена и сигнатуры типов методов из подкласса и суперкласса, то говорят, что метод из подкласса *переопределяет* метод из суперкласса. Когда переопределенный метод вызывается из своего подкласса, он всегда ссылается на свой вариант, определенный в подклассе. А вариант метода, определенный в суперклассе, будет скрыт. Рассмотрим следующий пример:

```
// Переопределение метода
class A {
    int i, j;
    A(int a, int b) {
        i = a;
        j = b;
    }

    // вывести содержимое переменных i и j
    void show() {
        System.out.println("i и j: " + i + " " + j);
    }
}

class B extends A {
    int k;
    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }

    // вывести содержимое переменной k с помощью метода,
    // переопределяющего метод show() из класса A
    void show() {
        System.out.println("k: " + k);
    }
}

class Override {
    public static void main(String args[]) {
        B subOb = new B(1, 2, 3);
        subOb.show(); // здесь вызывается метод show() из класса B
    }
}
```

Эта программа выводит следующий результат:

k: 3

Когда в данной программе вызывается метод `show()` для объекта типа `B`, выбирается вариант этого метода, определенный в классе `B`. Это означает, что вариант метода `show()`, определенный в классе `B`, переопределяет его вариант, объявленный в классе `A`.

Если требуется получить доступ к варианту переопределенного метода из суперкласса, это можно сделать с помощью ключевого слова `super`. Например, в следующей версии класса `B` вариант метода `show()`, объявленный в суперклассе, вызывается из подкласса. Благодаря этому выводятся все переменные экземпляра.

```
class B extends A {
    int k;
```

```

    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }

    void show() {
        super.show(); // здесь вызывается метод show() из класса A
        System.out.println("k: " + k);
    }
}

```

Подстановка этой версии класса **A** в предыдущую программу приведет к выводу следующего результата:

```

i и j: 1 2
k: 3

```

В этой версии выполнение оператора `super.show()` приводит к вызову варианта метода `show()`, определенного в суперклассе.

Переопределение методов выполняется *только* в том случае, если имена и сигнатуры типов обоих методов одинаковы. В противном случае оба метода считаются перегружаемыми. Рассмотрим следующую измененную версию предыдущего примера:

```

// Методы с отличающимися сигнатурами считаются
// перегружаемыми, а не переопределяемыми
class A {
    int i, j;
    A(int a, int b) {
        i = a;
        j = b;
    }
    // вывести содержимое переменных i и j
    void show() {
        System.out.println("i и j: " + i + " " + j);
    }
}

// создать подкласс путем расширения класса A
class B extends A {
    int k;
    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }
    // переписать метод show()
    void show(String msg) {
        System.out.println(msg + k);
    }
}

class Override {
    public static void main(String args[]) {
        B subOb = new B(1, 2, 3);

        subOb.show("Это k: "); // вызвать метод show() из класса B
        subOb.show();           // вызвать метод show() из класса A
    }
}

```

Эта программа выводит следующий результат:

```
Это k: 3  
i и j: 1 2
```

Вариант метода `show()`, определенный в классе `B`, принимает строковый параметр. В итоге его сигнатура типа отличается от сигнатуры типа метода без параметров из класса `A`. Поэтому никакого переопределения (или сокрытия имени) не происходит. Вместо этого выполняется перегрузка варианта метода `show()`, определенного в классе `A`, вариантом, определенным в классе `B`.

Динамическая диспетчеризация методов

Несмотря на то что приведенные в предыдущем разделе примеры демонстрируют механизм переопределения методов, они не раскрывают весь его потенциал. В самом деле, если бы переопределение методов служило лишь для условного обозначения пространства имен, оно имело бы скорее теоретическое, а не практическое значение. Но в действительности это не так. Переопределение методов служит основой для одного из наиболее эффективных принципов в Java — *динамической диспетчеризации методов*. Динамическая диспетчеризация методов — это механизм, с помощью которого вызов переопределенного метода разрешается во время выполнения, а не компиляции. Динамическая диспетчеризация методов важна потому, что благодаря ей полиморфизм в Java реализуется во время выполнения.

Прежде всего сформулируем еще раз следующий важный принцип: ссылочная переменная из суперкласса может ссылаться на объект подкласса. Этот принцип используется в Java для разрешения вызовов переопределенных методов во время выполнения следующим образом: когда переопределенный метод вызывается по ссылке на суперкласс, нужный вариант этого метода выбирается в Java в зависимости от типа объекта, на который делается ссылка в момент вызова. Следовательно, этот выбор делается во время выполнения. По ссылке на разные типы объектов будут вызываться разные варианты переопределенного метода. Иначе говоря, вариант переопределенного метода выбирается для выполнения в зависимости от *типа объекта, на который делается ссылка*, а не типа ссылочной переменной. Так, если суперкласс содержит метод, переопределяемый в подклассе, то по ссылке на разные типы объектов через ссылочную переменную из суперкласса будут выполняться разные варианты этого метода.

В следующем примере кода демонстрируется динамическая диспетчеризация методов:

```
// Динамическая диспетчеризация методов  
class A {  
    void callme() {  
        System.out.println("В методе callme() из класса A");  
    }  
}  
  
class B extends A {  
    // переопределить метод callme()  
    void callme() {  
        System.out.println("В методе callme() из класса B");  
    }  
}
```



```

    }
}

class C extends A {
    // переопределить метод callme()
    void callme() {
        System.out.println("В методе callme() из класса C");
    }
}

class Dispatch {
    public static void main(String args[]) {
        A a = new A(); // объект класса A
        B b = new B(); // объект класса B
        C c = new C(); // объект класса C

        A r;           // получить ссылку на объект типа A

        r = a;          // переменная r ссылается на объект типа A
        r.callme();      // вызвать вариант метода callme(),
                        // определенный в классе A
        r = b;          // переменная r ссылается на объект типа B
        r.callme();      // вызвать вариант метода callme(),
                        // определенный в классе B
        r = c;          // переменная r ссылается на объект типа C
        r.callme();      // вызвать вариант метода callme(),
                        // определенный в классе C
    }
}

```

Эта программа выводит следующий результат:

```

В методе callme() из класса A
В методе callme() из класса B
В методе callme() из класса C

```

В этой программе создаются один суперкласс A и два его подкласса B и C. В подклассах B и C переопределяется метод `callme()`, объявляемый в классе A. В методе `main()` объявляются объекты классов A, B и C, а также переменная `r` ссылки на объект типа A. Затем переменной `r` присваивается по очереди ссылка на объект каждого из классов A, B и C, и по этой ссылке вызывается метод `callme()`. Как следует из результата, выводимого этой программой, выполняемый вариант метода `callme()` определяется исходя из типа объекта, на который делается ссылка в момент вызова. Если бы выбор делался по типу ссылочной переменной `r`, то выводимый результат отражал бы три вызова одного и того же метода `callme()` из класса A.

На заметку! Тем, у кого имеется опыт программирования на C++ или C#, следует иметь в виду, что переопределенные методы в Java подобны виртуальным функциям в этих языках.

Назначение переопределенных методов

Как пояснялось ранее, переопределенные методы позволяют поддерживать в Java полиморфизм во время выполнения. Особое значение полиморфизма для ООП объясняется следующей причиной: он позволяет определить в общем

классе методы, которые станут общими для всех производных от него классов, а в подклассах — конкретные реализации некоторых или всех этих методов. Переопределенные методы предоставляют еще один способ реализовать в Java принцип полиморфизма “один интерфейс, множество методов”.

Одним из основных условий успешного применения полиморфизма является ясное понимание, что суперклассы и подклассы образуют иерархию по степени увеличения специализации. Если суперкласс применяется правильно, он предоставляет все элементы, которые могут непосредственно использоваться в подклассе. В нем также определяются те методы, которые должны быть реализованы в самом производном классе. Это дает удобную возможность определять в подклассе его собственные методы, сохраняя единообразие интерфейса. Таким образом, сочетая наследование с переопределенными методами, в суперклассе можно определить общую форму для методов, которые будут использоваться во всех его подклассах.

Динамический, реализуемый во время выполнения полиморфизм — один из самых эффективных механизмов объектно-ориентированной архитектуры, обеспечивающих повторное использование и надежность кода. Возможность вызывать из библиотек уже существующего кода методы для экземпляров новых классов, не прибегая к повторной компиляции и в то же время сохраняя ясность абстрактного интерфейса, является сильно действующим средством.

Применение переопределения методов

Рассмотрим более практический пример, в котором применяется переопределение методов. В приведенной ниже программе создается суперкласс `Figure` для хранения размеров двумерного объекта, а также определяется метод `area()` для расчета площади этого объекта. Кроме того, в этой программе создаются два класса, `Rectangle` и `Triangle`, производные от класса `Figure`. Метод `area()` переопределяется в каждом из этих подклассов, чтобы возвращать площадь четырехугольника и треугольника соответственно.

```
// Применение полиморфизма во время выполнения
class Figure {
    double dim1;
    double dim2;

    Figure(double a, double b) {
        dim1 = a;
        dim2 = b;
    }

    double area() {
        System.out.println("Площадь фигуры не определена.");
        return 0;
    }
}

class Rectangle extends Figure {
    Rectangle(double a, double b) {
        super(a, b);
    }

    // переопределить метод area() для четырехугольника
    double area() {
```

```

        System.out.println("В области четырехугольника.");
        return dim1 * dim2;
    }
}

class Triangle extends Figure {
    Triangle(double a, double b) {
        super(a, b);
    }

    // переопределить метод area() для прямоугольного треугольника
    double area() {
        System.out.println("В области треугольника.");
        return dim1 * dim2 / 2;
    }
}

class FindAreas {
    public static void main(String args[]) {
        Figure f = new Figure(10, 10);
        Rectangle r = new Rectangle(9, 5);
        Triangle t = new Triangle(10, 8);
        Figure figref;

        figref = r;
        System.out.println("Площадь равна " + figref.area());

        figref = t;
        System.out.println("Площадь равна " + figref.area());

        figref = f;
        System.out.println("Площадь равна " + figref.area());
    }
}

```

Эта программа выводит следующий результат:

```

В области четырехугольника.
Площадь равна 45
В области треугольника.
Площадь равна 40
Область фигуры не определена.
Площадь равна 0

```

Двойной механизм наследования и полиморфизма во время выполнения позволяет определить единый интерфейс, используемый разнотипными, хотя и связанными вместе классами объектов. Так, если объект относится к классу, производному от класса `Figure`, его площадь можно рассчитать, вызвав метод `area()`. Интерфейс для выполнения этой операции остается неизменным независимо от вида фигуры.

Применение абстрактных классов

Иногда суперкласс требуется определить таким образом, чтобы объявить в нем структуру заданной абстракции, не предоставляя полную реализацию каждого метода. Это означает создать суперкласс, определяющий только обобщенную форму для совместного использования всеми его подклассами, в каждом из которых мо-

гут быть добавлены требующиеся детали. В таком классе определяется характер методов, которые должны быть реализованы в подклассах. Подобная ситуация может, например, возникнуть, когда в суперклассе не удастся полностью реализовать метод. Именно так и было в классе `Figure` из предыдущего примера. Определение метода `area()` в этом классе служит лишь в качестве шаблона, не позволяя рассчитать и вывести площадь объекта какого-нибудь типа.

В процессе создания собственных библиотек классов вы сами убедитесь, что отсутствие полного определения метода в контексте суперкласса — не такая уж и редкая ситуация. Выйти из этой ситуации можно двумя способами. Один из них, как было показано в предыдущем примере, состоит в том, чтобы просто вывести предупреждающее сообщение. Такой способ удобен в определенных случаях, например, при отладке, но, как правило, он не годится. Ведь могут быть и такие методы, которые должны быть переопределены в подклассе, чтобы подкласс имел хотя бы какой-то смысл. Рассмотрим в качестве примера класс `Triangle`. Он лишен всякого смысла, если метод `area()` не определен. В таком случае требуется каким-то образом убедиться, что в подклассе действительно переопределяются все необходимые методы. И для этой цели в Java служит *абстрактный метод*.

Для того чтобы некоторые методы переопределялись в подклассе, достаточно объявить их с модификатором типа `abstract`. Иногда они называются *методами под ответственностью подкласса*, поскольку в суперклассе для них никакой реализации не предусмотрено. Следовательно, эти методы должны быть переопределены в подклассе, где нельзя просто воспользоваться их вариантом, определенным в суперклассе. Для объявления абстрактного метода используется приведенная ниже общая форма. Как видите, в этой форме тело метода отсутствует.

```
abstract тип имя(список_параметров) ;
```

Любой класс, содержащий один или больше абстрактных методов, должен быть также объявлен как абстрактный. Для этого достаточно указать ключевое слово `abstract` перед ключевым словом `class` в начале объявления класса. У абстрактного класса не может быть никаких объектов. Это означает, что экземпляр абстрактного класса не может быть получен непосредственно с помощью оператора `new`. Такие объекты были бы бесполезны, поскольку абстрактный класс определен не полностью. Кроме того, нельзя объявлять абстрактные конструкторы или абстрактные статические методы. Любой подкласс, производный от абстрактного класса, должен реализовать все абстрактные методы из своего суперкласса или же сам быть объявлен абстрактным.

Ниже приведен простой пример класса, содержащего абстрактный метод, и класса, реализующего этот метод.

```
// Простой пример абстракции
abstract class A {
    abstract void callme();

    // абстрактные классы все же могут содержать конкретные методы
    void callmetoo() {
        System.out.println("Это конкретный метод.");
    }
}
```

```

class B extends A {
    void callme() {
        System.out.println("Реализация метода callme() из класса B.");
    }
}

class AbstractDemo {
    public static void main(String args[]) {
        B b = new B();

        b.callme();
        b.callmetoo();
    }
}

```

Обратите внимание на то, что в этой программе объекты класса `A` не объявляются. Как отмечалось ранее, получить экземпляр абстрактного класса нельзя. И еще одно: в классе `A` реализуется конкретный метод `callmetoo()`, что вполне допустимо. В абстрактные классы может быть включена реализация какого угодно количества конкретных методов.

Несмотря на то что абстрактные классы не позволяют получать экземпляры объектов, их все же можно применять для создания ссылок на объекты, поскольку в Java полиморфизм во время выполнения реализован с помощью ссылок на суперкласс. Поэтому должна быть возможность создавать ссылку на абстрактный класс для указания на объект подкласса. В приведенном ниже примере показано, как воспользоваться такой возможностью.

Используя абстрактный класс, можно усовершенствовать созданный ранее класс `Figure`. Понятие площади неприменимо к неопределенной двумерной фигуре, поэтому в приведенной ниже новой версии программы метод `area()` объявляется в классе `Figure` как `abstract`. Это, конечно, означает, что метод `area()` должен быть переопределен во всех классах, производных от класса `Figure`.

```

// Применение абстрактных методов и классов
abstract class Figure {
    double dim1;
    double dim2;
    Figure(double a, double b) {
        dim1 = a;
        dim2 = b;
    }

    // теперь метод area() объявляется абстрактным
    abstract double area();
}

class Rectangle extends Figure {
    Rectangle(double a, double b) {
        super(a, b);
    }

    // переопределить метод area() для четырехугольника
    double area() {
        System.out.println("В области четырехугольника.");
        return dim1 * dim2;
    }
}

```

```
class Triangle extends Figure {
    Triangle(double a, double b) {
        super(a, b);
    }

    // переопределить метод area() для прямоугольного треугольника
    double area() {
        System.out.println("В области треугольника.");
        return dim1 * dim2 / 2;
    }
}

class AbstractAreas {
    public static void main(String args[]) {
        // Figure f = new Figure(10, 10); // теперь недопустимо
        Rectangle r = new Rectangle(9, 5);
        Triangle t = new Triangle(10, 8);
        Figure figref; // верно, но объект не создается

        figref = r;
        System.out.println("Площадь равна " + figref.area());

        figref = t;
        System.out.println("Площадь равна " + figref.area());
    }
}
```

Как следует из комментариев в теле метода `main()`, объявление объектов типа `Figure` больше не допускается, поскольку теперь этот класс является абстрактным. И во всех подклассах, производных от класса `Figure`, должен быть переопределен метод `area()`. Чтобы убедиться в этом, попытайтесь создать подкласс, в котором метод `area()` не переопределяется. Это приведет к ошибке во время компиляции.

Если создать объект типа `Figure` нельзя, то можно хотя бы создать ссылочную переменную типа `Figure`. Переменная `figref` объявлена как ссылка на класс `Figure`, т.е. ее можно использовать для ссылки на объект любого класса, производного от класса `Figure`. Как пояснялось ранее, вызовы переопределенных методов разрешаются во время выполнения с помощью ссылочных переменных из суперкласса.

Ключевое слово `final` в сочетании с наследованием

Ключевое слово `final` можно использовать тремя способами. Первый способ служит для создания эквивалента именованной константы. Такое применение ключевого слова `final` было описано в предыдущей главе. А два других способа его применения относятся к наследованию. Рассмотрим их подробнее.

Предотвращение переопределения с помощью ключевого слова `final`

Несмотря на то что переопределение методов является одним из самых эффективных языковых средств Java, иногда его желательно избегать. Чтобы запретить пере-

определение метода, в начале его объявления следует указать ключевое слово `final`. Методы, объявленные как `final`, переопределяться не могут. Такой способ применения ключевого слова `final` демонстрируется в приведенном ниже фрагменте кода.

```
class A {
    final void meth() {
        System.out.println("Это завершенный метод.");
    }
}

class B extends A {
    void meth() { // ОШИБКА! Этот метод не может быть переопределен.
        System.out.println("Недопустимо!");
    }
}
```

Метод `meth()` объявлен как `final` и поэтому не может быть переопределен в классе `B`. Любая попытка переопределить его приведет к ошибке во время компиляции.

Иногда методы, объявленные как `final`, могут способствовать увеличению производительности программы. Компилятор вправе *встраивать* вызовы этих методов, поскольку ему известно, что они не будут переопределены в подклассе. Нередко при вызове небольшого заверщенного метода компилятор Java может встраивать байт-код для подпрограммы непосредственно в скомпилированный код вызывающего метода, тем самым снижая издержки на вызов метода. Такая возможность встраивания вызовов присуща только завершенным методам. Как правило, вызовы методов разрешаются в Java динамически во время выполнения. Такой способ называется *поздним связыванием*. Но поскольку завершенные методы не могут быть переопределены, их вызовы могут быть разрешены во время компиляции. И такой способ называется *ранним связыванием*.

Предотвращения наследования с помощью ключевого слова `final`

Иногда требуется предотвратить наследование класса. Для этого в начале объявления класса следует указать ключевое слово `final`. Объявление класса завершенным неявно делает завершенными и все его методы. Нетрудно догадаться, что одновременное объявление класса как `abstract` и `final` недопустимо, поскольку абстрактный класс принципиально является незавершенным и только его подклассы предоставляют полную реализацию методов. Ниже приведен пример заверщенного класса.

```
final class A {
    // ...
}

// Следующий класс недопустим.
class B extends A { // ОШИБКА! Класс A не может иметь подклассы
    // ...
}
```

Как следует из комментария к приведенному выше коду, класс `B` не может наследовать от класса `A`, поскольку класс `A` объявлен завершенным.

Класс Object

В Java определен один специальный класс, называемый `Object`. Все остальные классы являются подклассами, производными от этого класса. Это означает, что класс `Object` служит суперклассом для всех остальных классов, и ссылочная переменная из класса `Object` может ссылаться на объект любого другого класса. А поскольку массивы реализованы в виде классов, то ссылочная переменная типа `Object` может ссылаться и на любой массив. В классе `Object` определены методы, перечисленные в табл. 8.1 и доступные для любого объекта.

Таблица 8.1. Методы из класса Object

Метод	Назначение
<code>Object clone()</code>	Создает новый объект, не отличающийся от клонируемого
<code>boolean equals(Object object)</code>	Определяет, равен ли один объект другому
<code>void finalize()</code>	Вызывается перед удалением неиспользуемого объекта
<code>Class<?> getClass()</code>	Получает класс объекта во время выполнения
<code>int hashCode()</code>	Возвращает хеш-код, связанный с вызывающим объектом
<code>void notify()</code>	Возобновляет исполнение потока, ожидающего вызывающего объекта
<code>void notifyAll()</code>	Возобновляет исполнение всех потоков, ожидающих вызывающего объекта
<code>String toString()</code>	Возвращает символьную строку, описывающую объект
<code>void wait()</code>	Ожидает другого потока исполнения
<code>void wait(long миллисекунд)</code>	
<code>void wait(long миллисекунд, int наносекунд)</code>	

Методы `getClass()`, `notify()`, `notifyAll()` и `wait()` объявлены как `final`. Остальные методы можно переопределять (они будут описаны в последующих главах). Обратите, однако, внимание на два метода: `equals()` и `toString()`. Метод `equals()` сравнивает два объекта. Если объекты равны, он возвращает логическое значение `true`, в противном случае — логическое значение `false`. Точное определение равенства зависит от типа сравниваемых объектов. Метод `toString()` возвращает символьную строку с описанием объекта, для которого он вызван. Кроме того, метод `toString()` вызывается автоматически, когда содержимое объекта выводится с помощью метода `println()`. Этот метод переопределяется во многих классах, чтобы приспособливать описание к создаваемым в них конкретным типам объектов.

И последнее замечание: обратите внимание на необычный синтаксис типа, возвращаемого методом `getClass()`. Этот синтаксис имеет отношение к *обобщениям* в Java, которые описываются в главе 14.

Пакеты и интерфейсы

В этой главе рассматриваются два наиболее новаторских языковых средства Java: пакеты и интерфейсы. *Пакеты* являются контейнерами классов. Они служат для разделения пространств имен класса. Например, можно создать класс `List`, чтобы хранить его в отдельном пакете, не беспокоясь о возможных конфликтах с другим классом `List`, хранящимся в каком-нибудь другом месте. Пакеты хранятся в иерархической структуре и явным образом импортируются при определении новых классов.

В предыдущих главах было описано применение методов для определения интерфейса с данными в классе. С помощью ключевого слова `interface` в Java можно полностью абстрагировать интерфейс от его реализации. Ключевое слово `interface` позволяет указать ряд методов, которые могут быть реализованы в одном или нескольких классах. В своей традиционной форме сам интерфейс не определяет никакой реализации. Несмотря на то что интерфейсы подобны абстрактным классам, они предоставляют дополнительную возможность: в одном классе можно реализовать несколько интерфейсов. Напротив, класс может наследоваться только от одного суперкласса (абстрактного или не абстрактного).

Пакеты

Ранее во всех примерах классов использовались имена из одного и того же пространства имен. Это означает, что во избежание конфликта имен для каждого класса нужно было указывать однозначное имя. Но со временем в отсутствие какого-нибудь способа управления пространством имен может возникнуть ситуация, когда выбор удобных описательных имен отдельных классов станет затруднительным. Кроме того, требуется каким-то образом обеспечить, чтобы выбранное имя класса было обоснованно однозначным и не конфликтовало с именами классов, выбранными другими программистами. (Представьте себе небольшую группу программистов, спорящих о том, кто имеет право использовать `Foobar` в качестве имени класса. Или вообразите себе все сообщество Интернета, спорящее о том, кто первым назвал класс `Espresso`.) К счастью, в Java предоставляется механизм разделения пространства имен на более удобные для управления фрагменты. Этим механизмом является пакет, который одновременно используется и как механизм присвоения имен, и как механизм управления доступностью объектов.

В пакете можно определить классы, недоступные для кода за пределами этого пакета. В нем можно также определить члены класса, доступные только другим чле-

нам этого же пакета. Благодаря такому механизму классы могут располагать полными сведениями друг о друге, но не предоставлять эти сведения остальному миру.

Определение пакета

Создать пакет совсем не трудно — достаточно включить оператор `package` в первую строку кода исходного файла программы на Java. Любые классы, объявленные в этом файле, будут принадлежать указанному пакету. Оператор `package` определяет пространство имен, в котором хранятся классы. Если же оператор `package` отсутствует, то имена классов размещаются в пакете, используемом по умолчанию и не имеющем имени. (Именно поэтому в приведенных до сих пор примерах не нужно было беспокоиться об определении пакетов.) Если пакет по умолчанию вполне подходит для коротких примеров программ, то он не годится для реальных приложений. Зачастую для прикладного кода придется определять отдельный пакет. Оператор `package` имеет следующую общую форму:

```
package пакет;
```

где *пакет* обозначает имя конкретного пакета. Например, в приведенной ниже строке кода создается пакет `MyPackage`.

```
package MyPackage;
```

Для хранения пакетов в Java используются каталоги файловой системы. Например, файлы с расширением `.class` для любых классов, объявленных в качестве составной части пакета `MyPackage`, должны храниться в каталоге `MyPackage`. Напомним, что в именах файлов и каталогов учитывается регистр символов, а кроме того, они должны точно соответствовать имени пакета.

Один и тот же оператор `package` может присутствовать в нескольких исходных файлах. Этот оператор просто обозначает пакет, которому принадлежат классы, определенные в данном файле. Это никак не мешает классам из других файлов входить в тот же самый пакет. Большинство пакетов, применяемых в реальных программах, распределено по многим файлам.

В Java допускается создавать иерархию пакетов. Для этой цели служит операция-точка. Объявление многоуровневого пакета имеет следующую общую форму:

```
package пакет1[.пакет2[.пакет3]];
```

Иерархия пакетов должна быть отражена в файловой системе той среды, где разрабатываются программы на Java. Например, в среде Windows пакет, объявленный как `package java.awt.image;`, должен храниться в каталоге `java\awt\image`. Имена пакетов следует выбирать очень аккуратно и внимательно. Ведь имя пакета нельзя изменить, не переименовав каталог, в котором хранятся классы.

Поиск пакетов и переменная окружения `CLASSPATH`

Как пояснялось в предыдущем разделе, пакеты соответствуют каталогам. В связи с этим возникает следующий важный вопрос: откуда исполняющей системе Java известно, где следует искать создаваемые пакеты? Ответ на него можно разделить

на три части. Во-первых, в качестве отправной точки исполняющая система Java использует по умолчанию текущий рабочий каталог. Так, если пакет находится в подкаталоге текущего каталога, он будет найден. Во-вторых, один или несколько путей к каталогу можно указать, установив соответствующее значение в переменной окружения `CLASSPATH`. И в-третьих, команды `java` и `javac` можно указывать в командной строке с параметром `-classpath`, обозначающим путь к классам.

Рассмотрим в качестве примера следующую спецификацию пакета:

```
package MyPack;
```

Чтобы программа могла найти пакет `MyPack`, должно быть выполнено одно из следующих условий. Программа должна быть запущена на выполнение из каталога, находящегося непосредственно над каталогом `MyPack`; переменная окружения `CLASSPATH` должна содержать путь к каталогу `MyPack`, или же параметр `-classpath` должен обозначать путь к каталогу `MyPack` в команде `java`, запускаемой из командной строки.

При использовании двух последних способов путь к классу *не должен* включать сам пакет `MyPack`, а просто указывать *путь* к этому пакету. Так, если в среде Windows путь к каталогу `MyPack` выглядит следующим образом:

```
C:\MyPrograms\Java\MyPack
```

то путь к классу из пакета `MyPack` будет таким:

```
C:\MyPrograms\Java
```

Самый простой способ опробовать примеры программ, приведенные в этой книге, состоит в том, чтобы создать каталоги пакетов в текущем каталоге разработки программ, разместить файлы с расширением `.class` в соответствующих каталогах и запускать программы из каталога разработки. В примере, рассматриваемом в следующем разделе, применяется именно такой подход.

Краткий пример пакета

Принимая во внимание все сказанное выше, рассмотрим следующий пример простого пакета:

```
// Простой пакет
package MyPack;

class Balance {
    String name;
    double bal;

    Balance(String n, double b) {
        name = n;
        bal = b;
    }

    void show() {
        if(bal<0)
            System.out.print("--> ");
        System.out.println(name + ": $" + bal);
    }
}
```

```

class AccountBalance {
    public static void main(String args[]) {
        Balance current[] = new Balance[3];

        current[0] = new Balance("K. J. Fielding", 123.23);
        current[1] = new Balance("Will Tell", 157.02);
        current[2] = new Balance("Tom Jackson", -12.33);

        for(int i=0; i<3; i++) current[i].show();
    }
}

```

Назовите этот исходный файл `AccountBalance.java` и разместите его в каталоге `MyPack`, а затем скомпилируйте его. Непременно расположите получаемый в итоге файл с расширением `.class` в том же каталоге `MyPack`. После этого попробуйте выполнить класс `AccountBalance`, введя в командной строке следующую команду:

```
java MyPack.AccountBalance
```

Напомним, что при выполнении этой команды текущим должен быть каталог, расположенный над каталогом `MyPack`, или же в переменной окружения `CLASSPATH` должен быть указан соответствующий путь. Как пояснялось ранее, теперь класс `AccountBalance` входит в пакет `MyPack`. Это означает, что его нельзя выполнять самостоятельно. Следовательно, в командной строке нельзя выполнить приведенную ниже команду, поскольку для уточнения имени класса `AccountBalance` требуется указать имя его пакета.

```
java AccountBalance
```

Защита доступа

В предыдущих главах были рассмотрены различные особенности механизма управления доступом в Java и его модификаторы. В частности, доступ к закрытому члену класса предоставляется только другим членам этого класса. Пакеты расширяют возможности управления доступом. Как станет ясно в дальнейшем, в Java предоставляются многие уровни защиты, обеспечивающие очень точное управление доступностью переменных и методов в классах, подклассах и пакетах.

Классы и пакеты одновременно служат средствами инкапсуляции и хранилищем пространства имен и области видимости переменных и методов. Пакеты играют роль контейнеров классов и других подчиненных пакетов. Классы служат контейнерами данных и кода. Класс — наименьшая единица абстракции в Java. Вследствие взаимодействия между классами и пакетами в Java определяются четыре категории доступности членов класса.

- Подклассы в одном пакете.
- Классы в одном пакете, не являющиеся подклассами.
- Подклассы в различных пакетах.
- Классы, которые не находятся в одном пакете и не являются подклассами.

Три модификатора доступа (`private`, `public` и `protected`) предоставляют различные способы создания многих уровней доступа, необходимых для этих категорий. Взаимосвязь между ними описана в табл. 9.1.

Таблица 9.1. Доступ к членам класса

	Private	Модификатор отсутствует	Protected	Public
Один и тот же класс	Да	Да	Да	Да
Подкласс, производный от класса из того же самого пакета	Нет	Да	Да	Да
Класс из того же самого пакета, не являющийся подклассом	Нет	Да	Да	Да
Подкласс, производный от класса другого пакета	Нет	Нет	Да	Да
Класс из другого пакета, не являющийся подклассом, производный от класса из данного пакета	Нет	Нет	Нет	Да

На первый взгляд, механизм управления доступом в Java может показаться сложным, поэтому следующие разъяснения помогут легче понять этот механизм. Любой компонент, объявленный как `public`, доступен из любого кода. А любой компонент, объявленный как `private`, недоступен для компонентов, находящихся за пределами его класса. Если в объявлении члена класса отсутствует явно указанный модификатор доступа, он доступен для подклассов и других классов из данного пакета. Этот уровень доступа используется по умолчанию. Если же требуется, чтобы элемент был доступен за пределами его текущего пакета, но только классам, непосредственно производным от данного класса, такой элемент должен быть объявлен как `protected`.

Правила доступа, приведенные в табл. 9.1, применимы только к членам класса. Для класса, не являющегося вложенным, может быть указан только один из двух возможных уровней доступа: по умолчанию и открытый (`public`). Если класс объявлен как `public`, он доступен из любого другого кода. Если у класса имеется уровень доступа по умолчанию, такой класс оказывается доступным только для кода из данного пакета. Если же класс оказывается открытым, он должен быть единственным открытым классом, объявленным в файле, а имя этого файла должно совпадать с именем класса.

Пример защиты доступа

В приведенном ниже примере демонстрируется применение модификаторов управления доступом во всех возможных сочетаниях. В этом примере употребляются два пакета и пять классов. Не следует, однако, забывать, что классы из двух разных пакетов должны храниться в каталогах, имена которых совпадают с именами соответствующих пакетов (в данном случае — `p1` и `p2`).

В исходном файле первого пакета определяются три класса: `Protection`, `Derived` и `SamePackage`. В первом классе определяются четыре переменные типа `int`: по одной в каждом из допустимых режимов защиты доступа. Переменная `n` объявлена с уровнем защиты по умолчанию, переменная `n_pri` — как `private`, переменная `n_pro` — как `protected`, переменная `n_pub` — как `public`.

В данном примере все другие классы будут предпринимать попытку обращения к переменным экземпляра первого класса. Строки кода, компиляция которых невозможна из-за нарушений правил доступа, закомментированы. Перед каждой из этих строк расположен комментарий с указанием мест программы, из которых был бы возможен доступ с заданным уровнем защиты.

Вторым в пакете `p1` является класс `Derived`, производный от класса `Protection` из этого же пакета. Это предоставляет классу `Derived` доступ ко всем переменным из класса `Protection`, кроме переменной `n_pri`, объявленной как `private`. Третий класс, `SamePackage`, не является производным от класса `Protection`, но находится в этом же пакете и обладает доступом ко всем переменным, кроме переменной `n_pri`.

Файл `Protection.java` содержит следующий код:

```
package p1;

public class Protection {
    int n = 1;
    private int n_pri = 2;
    protected int n_pro = 3;
    public int n_pub = 4;

    public Protection() {
        System.out.println("конструктор базового класса");
        System.out.println("n = " + n);
        System.out.println("n_pri = " + n_pri);
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}
```

Файл `Derived.java` содержит такой код:

```
package p1;

class Derived extends Protection {
    Derived() {
        System.out.println("конструктор подкласса");
        System.out.println("n = " + n);

        // доступно только для класса
        // System.out.println("n_pri = " + n_pri);

        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}
```

Файл `SamePackage.java` содержит следующий код:

```
package p1;

class SamePackage {
```

```
SamePackage() {  
  
    Protection p = new Protection();  
    System.out.println("конструктор этого же пакета");  
    System.out.println("n = " + p.n);  
  
    // доступно только для класса  
    // System.out.println("n_pri = " + p.n_pri);  
  
    System.out.println("n_pro = " + p.n_pro);  
    System.out.println("n_pub = " + p.n_pub);  
}  
}
```

Ниже приведен исходный код второго пакета p2. Два определенных в нем класса отражают два оставшихся режима управления доступом. Первый класс, Protection2, является производным от класса p1.Protection. Он имеет доступ ко всем переменным класса p1.Protection, кроме переменной n_pri, поскольку она объявлена как private, а также переменной n, объявленной с уровнем защиты по умолчанию. Напомним, что режим доступа по умолчанию разрешает доступ из данного класса или пакета, но не из подклассов другого пакета. И наконец, класс OtherPackage имеет доступ только к одной переменной n_pub, объявленной как public.

Файл Protection2.java содержит следующий код:

```
package p2;  
  
class Protection2 extends p1.Protection {  
    Protection2() {  
        System.out.println(  
            "конструктор, унаследованный из другого пакета");  
        // доступно только для данного класса или пакета  
        // System.out.println("n = " + n);  
  
        // доступно только для данного класса  
        // System.out.println("n_pri = " + n_pri);  
  
        System.out.println("n_pro = " + n_pro);  
        System.out.println("n_pub = " + n_pub);  
    }  
}
```

Файл OtherPackage.java содержит следующий код:

```
package p2;  
  
class OtherPackage {  
    OtherPackage() {  
        p1.Protection p = new p1.Protection();  
        System.out.println("конструктор из другого пакета");  
  
        // доступно только для данного класса или пакета  
        // System.out.println("n = " + p.n);  
  
        // доступно только для данного класса  
        // System.out.println("n_pri = " + p.n_pri);  
  
        // доступно только для данного класса, подкласса или пакета  
        // System.out.println("n_pro = " + p.n_pro);  
    }  
}
```



```

        System.out.println("n_pub = " + p.n_pub);
    }
}

```

Для того чтобы проверить работоспособность упомянутых выше двух пакетов, следует воспользоваться двумя тестовыми файлами. В частности, тестовый файл для пакета `p1` имеет следующее содержимое:

```

// Демонстрационный пакет p1
package p1;

// получить экземпляры различных классов из пакета p1
public class Demo {
    public static void main(String args[]) {
        Protection obl = new Protection();
        Derived ob2 = new Derived();
        SamePackage ob3 = new SamePackage();
    }
}

```

Еще один тестовый файл из пакета `p2` содержит следующее:

```

// Демонстрационный пакет p2
package p2;

// получить экземпляры различных классов из пакета p2
public class Demo {
    public static void main(String args[]) {
        Protection2 obl = new Protection2();
        OtherPackage ob2 = new OtherPackage();
    }
}

```

Импорт пакетов

Если вспомнить, что пакеты предлагают эффективный механизм изоляции различных классов друг от друга, то становится понятно, почему все встроенные классы Java хранятся в пакетах. Ни один из основных классов Java не хранится в неименованном пакете, используемом по умолчанию. Все стандартные классы непременно хранятся в каком-нибудь именованном пакете. А поскольку в самих пакетах классы должны полностью определяться по именам их пакетов, то длинное, разделяемое точками имя пути к пакету каждого используемого класса может оказаться слишком громоздким. Следовательно, чтобы отдельные классы или весь пакет можно было сделать доступными, в Java внедрен оператор `import`. После того как класс импортирован, на него можно ссылаться непосредственно, используя только его имя. Оператор `import` служит только для удобства программирования и не является обязательным с формальной точки зрения для создания завершенной программы на Java. Но если в прикладном коде приходится ссылаться на несколько десятков классов, то оператор `import` значительно сокращает объем вводимого исходного кода.

В исходном файле программы на Java операторы `import` должны следовать непосредственно за оператором `package` (если таковой имеется) и перед любыми определениями классов. Оператор `import` имеет следующую общую форму:

```
import пакет1 [.пакет2].(имя_класса | *);
```

где *пакет1* обозначает имя пакета верхнего уровня, а *пакет2* — имя подчиненного пакета из внешнего пакета, отделяемое знаком точки (.). Глубина вложенности пакетов практически не ограничивается ничем, кроме файловой системы. И наконец, *имя_класса* может быть задано явно или с помощью знака “звездочка” (*), который указывает компилятору Java на необходимость импорта всего пакета. В следующем фрагменте кода демонстрируется применение обоих вариантов общей формы оператора `import`:

```
import java.util.Date;
import java.io.*;
```

Все классы из стандартной библиотеки Java хранятся в пакете `java`. Основные языковые средства хранятся в пакете `java.lang`, входящем в пакет `java`. Обычно каждый пакет или класс, который требуется использовать, приходится импортировать. Но поскольку программировать на Java бесполезно без многих средств, определенных в пакете `java.lang`, компилятор неявно импортирует его для всех программ. Это равнозначно наличию следующей строки кода в каждой из программ на Java:

```
import java.lang.*;
```

Компилятор никак не отреагирует на наличие классов с одинаковыми именами в двух разных пакетах, импортируемых в форме со звездочкой, если только не будет предпринята попытка воспользоваться одним из этих классов. В таком случае возникнет ошибка во время компиляции, и тогда имя класса придется указать явно вместе с его пакетом.

Следует особо подчеркнуть, что указывать оператор `import` совсем не обязательно. *Полностью уточненное имя* класса с указанием всей иерархии пакетов можно использовать везде, где допускается имя класса. Например, в приведенном ниже фрагменте кода применяется оператор `import`.

```
import java.util.*;
class MyDate extends Date {
}
```

Этот же фрагмент кода, но без оператора `import`, показан ниже. В этой версии класс `Date` полностью определен.

```
class MyDate extends java.util.Date {
}
```

Как следует из табл. 9.1, при импорте пакета классам, не производным от классов из данного пакета в импортирующем коде, будут доступны только те элементы пакета, которые объявлены как `public`. Так, если требуется, чтобы упоминавшийся ранее класс `Balance` из пакета `MyPack` был доступен в качестве самостоятельного класса за пределами пакета `MyPack`, его следует объявить как `public` и разместить в отдельном файле, как показано в приведенном ниже примере кода.

```
package MyPack;
```

```
/* Теперь класс Balance, его конструктор и метод show()
   являются открытыми. Это означает, что за пределами своего пакета
   они доступны из кода классов, не производных от классов их пакета.
*/
```

```

public class Balance {
    String name;
    double bal;

    public Balance(String n, double b) {
        name = n;
        bal = b;
    }

    public void show() {
        if(bal<0)
            System.out.print("--> ");
        System.out.println(name + ": $" + bal);
    }
}

```

Как видите, класс `Balance` теперь объявлен как `public`. Его конструктор и метод `show()` также объявлены как `public`. Это означает, что они доступны для любого кода за пределами пакета `MyPack`. Например, класс `TestBalance` импортирует пакет `MyPack`, и поэтому в нем может быть использован класс `Balance`.

```

import MyPack.*;

class TestBalance {
    public static void main(String args[]) {
        /* Класс Balance объявлен как public, поэтому им можно
           воспользоваться и вызвать его конструктор. */
        Balance test = new Balance("J. J. Jaspers", 99.88);

        test.show(); // можно также вызвать метод show()
    }
}

```

В качестве эксперимента удалите модификатор `public` из объявления класса `Balance`, а затем попытайтесь скомпилировать класс `TestBalance`. Как упоминалось выше, это приведет к появлению ошибок.

Интерфейсы

С помощью ключевого слова `interface` можно полностью абстрагировать интерфейс класса от его реализации. Это означает, что с помощью ключевого слова `interface` можно указать, *что* именно должен выполнять класс, но не *как* это делать. Синтаксически интерфейсы аналогичны классам, но не содержат переменные экземпляра, а объявления их методов, как правило, не содержат тело метода. На практике это означает, что можно объявлять интерфейсы, которые не делают никаких допущений относительно их реализации. Как только интерфейс определен, его может реализовать любое количество классов. Кроме того, один класс может реализовать любое количество интерфейсов.

Чтобы реализовать интерфейс, в классе должен быть создан полный набор методов, определенных в этом интерфейсе. Но в каждом классе могут быть определены особенности собственной реализации данного интерфейса. Ключевое слово `interface` позволяет в полной мере использовать принцип полиморфизма “один интерфейс, несколько методов”.

Интерфейсы предназначены для поддержки динамического разрешения вызовов методов во время выполнения. Как правило, для нормального выполнения вызова метода из одного класса в другом оба класса должны присутствовать во время компиляции, чтобы компилятор Java мог проверить совместимость сигнатур методов. Само по себе это требование создает статическую и нерасширяемую среду распределения классов. В такой системе функциональные возможности неизбежно передаются вверх по иерархии классов, в результате чего механизмы будут становиться доступными все большему количеству подклассов. Интерфейсы предназначены для предотвращения этой проблемы. Они изолируют определение метода или набора методов от иерархии наследования. А поскольку иерархия интерфейсов не совпадает с иерархией классов, то классы, никак не связанные между собой иерархически, могут реализовать один и тот же интерфейс. Именно в этом возможности интерфейсов проявляются наиболее полно.

Объявление интерфейса

Во многом определение интерфейса подобно определению класса. Упрощенная общая форма интерфейса имеет следующий вид:

```
доступ interface имя {  
    возвращаемый_тип имя_метода1(список_параметров);  
    возвращаемый_тип имя_метода2(список_параметров);  
    тип имя_завершенной_переменной1 = значение;  
    тип имя_завершенной_переменной2 = значение;  
    // ...  
    возвращаемый_тип имя_методаN(список_параметров);  
    тип имя_завершенной_переменнойN = значение;  
}
```

Если определение не содержит никакого модификатора доступа, используется доступ по умолчанию, а интерфейс доступен только другим членам того пакета, в котором он объявлен. Если интерфейс объявлен как `public`, он может быть использован в любом другом коде. В этом случае интерфейс должен быть единственным открытым интерфейсом, объявленным в файле, а имя этого файла должно совпадать с именем интерфейса. В приведенной выше общей форме указанное *имя* обозначает конкретное имя интерфейса, которым может быть любой допустимый идентификатор. Обратите внимание на то, что объявляемые методы не содержат тел. Их объявления завершаются списком параметров, за которым следует точка с запятой. По существу, они являются абстрактными методами. Каждый класс, который включает в себя интерфейс, должен реализовать все его методы.

Прежде чем продолжить дальше, важно отметить, что в версии JDK 8 ключевое слово `interface` дополнено средством, значительно изменяющим его возможности. До версии JDK 8 в интерфейсе вообще нельзя было ничего реализовать. В интерфейсе, упрощенная форма объявления которого приведена выше, наличие тела в объявляемых методах не предполагалось. Следовательно, до версии JDK 8 в интерфейсе можно было определить только то, *что* именно следует сделать, но не *как* это сделать. Это положение изменилось в версии JDK 8. Теперь метод можно объявлять в интерфейсе с *реализацией по умолчанию*, т.е. указать его поведение. Но методы по умолчанию, по существу, служат специальной цели, сохраняя в то же

время исходное назначение интерфейса. Следовательно, интерфейсы можно по-прежнему создавать и использовать и без методов по умолчанию. Именно по этой причине обсуждение интерфейсов будет начато с их традиционной формы. А методы по умолчанию описываются в конце этой главы.

Как следует из приведенной выше общей формы, в объявлениях интерфейсов могут быть объявлены переменные. Они неявно объявляются как `final` и `static`, т.е. их нельзя изменить в классе, реализующем интерфейс. Кроме того, они должны быть инициализированы. Все методы и переменные неявно объявляются в интерфейсе как `public`.

Ниже приведен пример объявления интерфейса. В нем объявляется простой интерфейс, который содержит один метод `callback()`, принимающий единственный целочисленный параметр.

```
interface Callback {
    void callback(int param);
}
```

Реализация интерфейсов

Как только интерфейс определен, он может быть реализован в одном или нескольких классах. Чтобы реализовать интерфейс, в определение класса требуется включить выражение `implements`, а затем создать методы, определенные в интерфейсе. Общая форма класса, который содержит выражение `implements`, имеет следующий вид:

```
доступ class имя_класса [extends суперкласс]
    [implements интерфейс [, интерфейс...]] {
    // тело класса
}
```

Если в классе реализуется больше одного интерфейса, имена интерфейсов разделяются запятыми. Так, если в классе реализуются два интерфейса, в которых объявляется один и тот же метод, то этот же метод будет использоваться клиентами любого из двух интерфейсов. Методы, реализующие элементы интерфейса, должны быть объявлены как `public`. Кроме того, сигнатура типа реализующего метода должна в точности совпадать с сигнатурой типа, указанной в определении `interface`.

Рассмотрим небольшой пример класса, где реализуется приведенный ранее интерфейс `Callback`. Обратите внимание на то, что метод `callback()` объявлен с модификатором доступа `public`.

```
class Client implements Callback {
    // реализовать интерфейс Callback
    public void callback(int p) {

        System.out.println(
            "Метод callback(), вызываемый со значением " + p);
    }
}
```

Помните! Когда реализуется метод из интерфейса, он должен быть объявлен как `public`.

Вполне допустима и достаточно распространена ситуация, когда в классах, реализующих интерфейсы, определяются также собственные члены. Например, в следующей версии класса `Client` реализуется метод `callback()` и добавляется метод `nonIfaceMeth()`:

```
class Client implements Callback {
    // реализовать интерфейс Callback
    public void callback(int p) {
        System.out.println(
            "Метод callback(), вызываемый со значением " + p);
    }

    void nonIfaceMeth() {
        System.out.println("В классах, реализующих интерфейсы," +
            "могут определяться и другие члены.");
    }
}
```

Доступ к реализациям через ссылки на интерфейсы

Переменные можно объявлять как ссылки на объекты, в которых используется тип интерфейса, а не тип класса. С помощью такой переменной можно ссылаться на любой экземпляр какого угодно класса, реализующего объявленный интерфейс. При вызове метода по одной из таких ссылок нужный вариант будет выбираться в зависимости от конкретного экземпляра интерфейса, на который делается ссылка. Это одна из главных особенностей интерфейсов. Поиск исполняемого метода осуществляется динамически во время выполнения, что позволяет создавать классы позднее, чем код, из которого вызываются методы этих классов. Вызывающий код может выполнять диспетчеризацию методов с помощью интерфейса, даже не имея никаких сведений о вызываемом коде. Этот процесс аналогичен использованию ссылки на суперкласс для доступа к объекту подкласса (см. главу 8).

Внимание! Поскольку в Java динамический поиск методов во время выполнения сопряжен со значительными издержками по сравнению с обычным вызовом методов, в прикладном коде, критичном к производительности, интерфейсы следует использовать только тогда, когда это действительно необходимо.

В следующем примере программы метод `callback()` вызывается через переменную ссылки на интерфейс:

```
class TestIface {
    public static void main(String args[]) {
        Callback c = new Client();
        c.callback(42);
    }
}
```

Эта программа выводит следующий результат:

Метод `callback()`, вызываемый со значением 42

Обратите внимание на то, переменной `c` присвоен экземпляр класса `Client`, несмотря на то, что она объявлена с типом интерфейса `Callback`. Переменную `c` можно использовать для доступа к методу `callback()`, она не предоставляет до-

ступа к каким-нибудь другим членам класса `Client`. Переменная ссылки на интерфейс располагает только сведениями о методах, объявленных в том интерфейсе, на который она ссылается. Таким образом, переменной `c` нельзя пользоваться для доступа к методу `nonIfaceMeth()`, поскольку этот метод объявлен в классе `Client`, а не в интерфейсе `Callback`.

Приведенный выше пример формально показывает, каким образом переменная ссылки на интерфейс может получать доступ к объекту его реализации, тем не менее, он не демонстрирует полиморфные возможности такой ссылки. Чтобы продемонстрировать такие возможности, создадим сначала вторую реализацию интерфейса `Callback`, как показано ниже.

```
// Еще одна реализация интерфейса Callback
class AnotherClient implements Callback {
    // реализовать интерфейс Callback
    public void callback(int p) {
        System.out.println("Еще один вариант метода callback()");
        System.out.println("p в квадрате равно " + (p*p));
    }
}
```

Теперь попробуем создать и опробовать следующий класс:

```
class TestIface2 {
    public static void main(String args[]) {
        Callback c = new Client();
        AnotherClient ob = new AnotherClient();

        c.callback(42);

        c = ob; // теперь переменная c ссылается на
                // объект типа AnotherClient
        c.callback(42);
    }
}
```

Эта программа выводит следующий результат:

```
Метод callback(), вызываемый со значением 42
Еще один вариант метода callback()
p в квадрате равно 1764
```

Как видите, вызываемый вариант метода `callback()` выбирается в зависимости от типа объекта, на который переменная `c` ссылается во время выполнения. Приведенный выше пример довольно прост, поэтому далее будет рассмотрен еще один, более практический пример.

Частичные реализации

Если класс включает в себя интерфейс, но не полностью реализует определенные в нем методы, он должен быть объявлен как `abstract`:

```
abstract class Incomplete implements Callback {
    int a, b;

    void show() {
        System.out.println(a + " " + b);
    }
}
```

```
// ...  
}
```

В данном примере кода класс `Incomplete` не реализует метод `callback()`, поэтому он должен быть объявлен как абстрактный. Любой класс, наследующий от класса `Incomplete`, должен реализовать метод `callback()` или быть также объявленным как `abstract`.

Вложенные интерфейсы

Интерфейс может быть объявлен членом класса или другого интерфейса. Такой интерфейс называется *интерфейсом-членом* или *вложенным интерфейсом*. Вложенный интерфейс может быть объявлен как `public`, `private` или `protected`. Этим он отличается от интерфейса верхнего уровня, который должен быть объявлен как `public` или использовать уровень доступа по умолчанию, как отмечалось ранее. Когда вложенный интерфейс используется за пределами объемлющей его области действия, его имя должно быть дополнительно уточнено именем класса или интерфейса, членом которого он является. Это означает, что за пределами класса или интерфейса, в котором объявлен вложенный интерфейс, его имя должно быть уточнено полностью.

В следующем примере демонстрируется применение вложенного интерфейса:

```
// Пример вложенного интерфейса  
  
// Этот класс содержит интерфейс как свой член  
class A {  
    // это вложенный интерфейс  
    public interface NestedIF {  
        boolean isNotNegative(int x);  
    }  
}  
  
// Класс B реализует вложенный интерфейс  
class B implements A.NestedIF {  
    public boolean isNotNegative(int x) {  
        return x < 0 ? false : true;  
    }  
}  
  
class NestedIFDemo {  
    public static void main(String args[]) {  
  
        // использовать ссылку на вложенный интерфейс  
        A.NestedIF nif = new B();  
  
        if(nif.isNotNegative(10))  
            System.out.println("Число 10 неотрицательное");  
        if(nif.isNotNegative(-12))  
            System.out.println("Это не будет выведено");  
    }  
}
```

Обратите внимание на то, что в классе `A` определяется вложенный интерфейс `NestedIF`, объявленный как `public`. Затем вложенный интерфейс реализуется в классе `B` следующим образом:

```
implements A.NestedIF
```


Обратите также внимание на то, что имя интерфейса полностью уточнено и содержит имя класса. В теле метода `main()` создается переменная `nif` ссылки на интерфейс `A.NestedIF`, которой присваивается ссылка на объект класса `B`. И это вполне допустимо, поскольку класс `B` реализует интерфейс `A.NestedIF`.

Применение интерфейсов

Чтобы стали понятнее возможности интерфейсов, рассмотрим более практический пример. В предыдущих главах был разработан класс `Stack`, реализующий простой стек фиксированного размера. Но стек можно реализовать самыми разными способами. Например, стек может иметь фиксированный или “наращиваемый” размер. Стек может также храниться в массиве, связанном списке, двоичном дереве и т.п. Независимо от реализации стека, его интерфейс остается неизменным. Это означает, что методы `push()` и `pop()` определяют интерфейс стека независимо от особенностей его реализации. А поскольку интерфейс стека отделен от его реализации, то такой интерфейс можно определить без особого труда, оставив уточнение конкретных деталей в его реализации. Рассмотрим два примера применения интерфейса стека.

Создадим сначала интерфейс, определяющий целочисленный стек, разместив его в файле `IntStack.java`. Этот интерфейс будет использоваться в обеих реализациях стека.

```
// Определить интерфейс для целочисленного стека
interface IntStack {
    void push(int item); // сохранить элемент в стеке
    int pop();           // извлечь элемент из стека
}
```

В приведенной ниже программе создается класс `FixedStack`, реализующий версию целочисленного стека фиксированной длины.

```
// Реализация интерфейса IntStack для стека фиксированного размера
class FixedStack implements IntStack {
    private int stck[];
    private int tos;
    // выделить память и инициализировать стек
    FixedStack(int size) {
        stck = new int[size];
        tos = -1;
    }

    // разместить элемент в стеке
    public void push(int item) {
        if(tos==stck.length-1) // использовать поле длины стека
            System.out.println("Стек заполнен.");
        else
            stck[++tos] = item;
    }

    // извлечь элемент из стека
    public int pop() {
        if(tos < 0) {
            System.out.println("Стек не загружен.");
            return 0;
        }
    }
}
```

```
    }
    else
        return stck[tos--];
}
}

class IFTest {
public static void main(String args[]) {
    FixedStack mystack1 = new FixedStack(5);
    FixedStack mystack2 = new FixedStack(8);

    // разместить числа в стеке
    for(int i=0; i<5; i++) mystack1.push(i);
    for(int i=0; i<8; i++) mystack2.push(i);

    // извлечь эти числа из стека
    System.out.println("Стек в mystack1:");
    for(int i=0; i<5; i++)
        System.out.println(mystack1.pop());

    System.out.println("Стек в mystack2:");
    for(int i=0; i<8; i++)
        System.out.println(mystack2.pop());
}
}
```

Ниже приведена еще одна реализация интерфейса `IntStack`, в которой с помощью того же самого определения `interface` создается динамический стек. В этой реализации каждый стек создается с первоначальной длиной. При превышении этой начальной длины размер стека увеличивается. Каждый раз, когда возникает потребность в дополнительном свободном месте, размер стека удваивается.

```
// Реализация "наращиваемого" стека
class DynStack implements IntStack {
    private int stck[];
    private int tos;

    // выделить память и инициализировать стек
    DynStack(int size) {
        stck = new int[size];
        tos = -1;
    }

    // разместить элемент в стеке
    public void push(int item) {
        // если стек заполнен, выделить память под стек большего размера
        if(tos==stck.length-1) {
            int temp[] = new int[stck.length * 2]; // удвоить размер стека
            for(int i=0; i<stck.length; i++) temp[i] = stck[i];
            stck = temp;
            stck[++tos] = item;
        }
        else
            stck[++tos] = item;
    }

    // извлечь элемент из стека
    public int pop() {
        if(tos < 0) {
            System.out.println("Стек не загружен.");
        }
    }
}
```

```

        return 0;
    }
    else
        return stck[tos--];
    }
}

class IFTest2 {
    public static void main(String args[]) {
        DynStack mystack1 = new DynStack(5);
        DynStack mystack2 = new DynStack(8);

        // В этих циклах увеличиваются размеры каждого стека
        for(int i=0; i<12; i++) mystack1.push(i);
        for(int i=0; i<20; i++) mystack2.push(i);

        System.out.println("Стек в mystack1:");
        for(int i=0; i<12; i++)
            System.out.println(mystack1.pop());

        System.out.println("Стек в mystack2:");
        for(int i=0; i<20; i++)
            System.out.println(mystack2.pop());
    }
}

```

В приведенном ниже примере программы создается класс, в котором используются обе реализации данного интерфейса в классах `FixedStack` и `DynStack`. Для этого применяется ссылка на интерфейс. Это означает, что поиск вариантов при вызове методов `push()` и `pop()` осуществляется во время выполнения, а не во время компиляции.

```

/* Создать переменную интерфейса и
   обратиться к стекам через нее.
*/
class IFTest3 {
    public static void main(String args[]) {
        IntStack mystack; // создать переменную ссылки на интерфейс
        DynStack ds = new DynStack(5);
        FixedStack fs = new FixedStack(8);

        mystack = ds;      // загрузить динамический стек
        // разместить числа в стеке
        for(int i=0; i<12; i++) mystack.push(i);

        mystack = fs;      // загрузить фиксированный стек
        for(int i=0; i<8; i++) mystack.push(i);

        mystack = ds;
        System.out.println("Значения в динамическом стеке:");
        for(int i=0; i<12; i++)
            System.out.println(mystack.pop());

        mystack = fs;
        System.out.println("Значения в фиксированном стеке:");
        for(int i=0; i<8; i++)
            System.out.println(mystack.pop());
    }
}

```

В этой программе переменная `mystack` содержит ссылку на интерфейс `IntStack`. Следовательно, когда она ссылается на переменную `ds`, выбираются варианты методов `push()` и `pop()`, определенные при реализации данного интерфейса в классе `DynStack`. Когда же она ссылается на переменную `fs`, выбираются варианты методов `push()` и `pop()`, определенные при реализации данного интерфейса в классе `FixedStack`. Как отмечалось ранее, все эти решения принимаются во время выполнения. Обращение к нескольким реализациям интерфейса через ссылочную переменную интерфейса является наиболее эффективным средством в Java для поддержки полиморфизма во время выполнения.

Переменные в интерфейсах

Интерфейсы можно применять для импорта совместно используемых констант в несколько классов путем простого объявления интерфейса, который содержит переменные, инициализированные нужными значениями. Когда интерфейс включается в класс (т.е. реализуется в нем), имена всех этих переменных оказываются в области действия констант. (Это аналогично использованию в программе на C/C++ заголовочного файла для создания большого количества констант с помощью директив `#define` или объявлений `const`.) Если интерфейс не содержит никаких методов, любой класс, включающий в себя такой интерфейс, на самом деле ничего не реализует. Это все равно, как если бы класс импортировал постоянные поля в пространство имен класса в качестве завершенных переменных. В следующем примере программы эта методика применяется для реализации автоматизированной системы “принятия решений”.

```
import java.util.Random;

interface SharedConstants {
    int NO = 0;
    int YES = 1;
    int MAYBE = 2;
    int LATER = 3;
    int SOON = 4;
    int NEVER = 5;
}

class Question implements SharedConstants {
    Random rand = new Random();
    int ask() {
        int prob = (int) (100 * rand.nextDouble());
        if (prob < 30)
            return NO;           // 30%
        else if (prob < 60)
            return YES;          // 30%
        else if (prob < 75)
            return LATER;        // 15%
        else if (prob < 98)
            return SOON;         // 13%
        else
            return NEVER;        // 2%
    }
}
```

```

class AskMe implements SharedConstants {
    static void answer(int result) {
        switch(result) {
            case NO:
                System.out.println("Нет");
                break;
            case YES:
                System.out.println("Да");
                break;
            case MAYBE:
                System.out.println("Возможно");
                break;
            case LATER:
                System.out.println("Позднее");
                break;
            case SOON:
                System.out.println("Вскоре");
                break;
            case NEVER:
                System.out.println("Никогда");
                break;
        }
    }

    public static void main(String args[]) {
        Question q = new Question();

        answer(q.ask());
        answer(q.ask());
        answer(q.ask());
        answer(q.ask());
    }
}

```

Обратите внимание на то, что в этой программе используется класс `Random` из стандартной библиотеки `Java`. В этом классе создаются псевдослучайные числа. Он содержит несколько методов, которые позволяют получать случайные числа в требуемой для программы форме. В данном примере применяется метод `nextDouble()`, возвращающий случайные числа в пределах от 0,0 до 1,0.

В рассматриваемом здесь примере программы два класса, `Question` и `AskMe`, реализуют интерфейс `SharedConstants`, в котором определены константы `NO` (Нет), `YES` (Да), `MAYBE` (Возможно), `SOON` (Вскоре), `LATER` (Позднее) и `NEVER` (Никогда). Код из каждого класса ссылается на эти константы так, как если бы они определялись и наследовались непосредственно в каждом классе. Ниже приведен результат, выводимый при выполнении данной программы. Обратите внимание на то, что при каждом запуске программы результаты ее выполнения оказываются разными.

```

Позднее
Вскоре
Нет
Да

```

На заметку! Упомянутая выше методика применения интерфейса для определения общих констант весьма противоречива и представлена ради полноты изложения материала.

Расширение интерфейсов

Ключевое слово `extends` позволяет одному интерфейсу наследовать другой. Синтаксис определения такого наследования аналогичен синтаксису наследования классов. Когда класс реализует интерфейс, наследующий другой интерфейс, он должен предоставлять реализации всех методов, определенных по цепочке наследования интерфейсов. Ниже приведен характерный пример расширения интерфейсов.

```
// Один интерфейс может расширять другой
interface A {
    void meth1();
    void meth2();
}

// Теперь интерфейс B включает в себя методы meth1() и meth2()
// и добавляет метод meth3()
interface B extends A {
    void meth3();
}

// Этот класс должен реализовать все методы из интерфейсов A и B
class MyClass implements B {
    public void meth1() {
        System.out.println("Реализация метода meth1().");
    }

    public void meth2() {
        System.out.println("Реализация метода meth2().");
    }

    public void meth3() {
        System.out.println("Реализация метода meth3().");
    }
}

class IFExtend {
    public static void main(String arg[]) {
        MyClass ob = new MyClass();

        ob.meth1();
        ob.meth2();
        ob.meth3();
    }
}
```

В порядке эксперимента можете попытаться удалить реализацию метода `meth1()` из класса `MyClass`. Это приведет к ошибке во время компиляции. Как отмечалось ранее, любой класс, реализующий интерфейс, должен реализовать и все определенные в этом интерфейсе методы, в том числе и любые методы, унаследованные от других интерфейсов.

Методы по умолчанию

Как пояснялось ранее, до версии JDK 8 в интерфейсе нельзя было вообще реализовывать методы. Это означало, что во всех предыдущих версиях Java ме-

тоды, указанные в интерфейсе, были абстрактными и не имели своего тела. Это традиционная форма интерфейса, обсуждавшаяся в предыдущих разделах главы. Но с выпуском версии JDK 8 положение изменилось, поскольку появилась новая возможность вводить в интерфейс так называемый *метод по умолчанию*. Иными словами, метод по умолчанию позволяет теперь объявлять в интерфейсе метод не абстрактным, а с конкретным телом. На стадии разработки версии JDK 8 метод по умолчанию назывался *методом расширения*, поэтому в литературе можно встретить оба обозначения этого нового языкового средства Java.

Главной побудительной причиной для внедрения методов по умолчанию было стремление предоставить средства, позволявшие расширять интерфейсы, не нарушая уже существующий код. Напомним, что если ввести новый метод в широко применяемый интерфейс, такое дополнение нарушит уже существующий код из-за того, что не удастся обнаружить реализацию нового метода. Данное затруднение позволяет устранить метод по умолчанию, поскольку он предоставляет реализацию, которая будет использоваться в том случае, если не будет явно предоставлена другая реализация. Таким образом, ввод метода по умолчанию в интерфейс не нарушит уже существующий код.

Еще одной побудительной причиной для внедрения метода по умолчанию было стремление указывать в интерфейсе, по существу, необязательные методы в зависимости от того, каким образом используется интерфейс. Например, в интерфейсе можно определить группу методов, воздействующих на последовательность элементов. Один из этих методов можно назвать `remove()` и предназначить его для удаления элемента из последовательности. Но если интерфейс предназначен для поддержки как изменяемых, так и неизменяемых последовательностей, то метод `remove()` оказывается, по существу, необязательным, поскольку его нельзя применять к неизменяемым последовательностям. В прошлом в классе, предназначенном для обработки неизменяемых последовательностей, приходилось реализовывать фактически пустой метод `remove()`, даже если он и не был нужен. А теперь реализацию по умолчанию метода `remove()`, не выполняющего никаких действий или генерирующего исключение, можно указать в интерфейсе. Благодаря этому исключается потребность реализовывать свой замещающий вариант этого метода в классе, предназначенном для обработки неизменяемых последовательностей. Следовательно, если предоставить в интерфейсе метод `remove()` по умолчанию, его реализация в классе окажется необязательной.

Важно отметить, что внедрение методов по умолчанию не изменяет главную особенность интерфейсов: неспособность сохранять данные состояния. В частности, в интерфейсе по-прежнему недопустимы переменные экземпляра. Следовательно, интерфейс отличается от класса тем, что он не допускает сохранения состояния. Более того, создавать экземпляр самого интерфейса нельзя. Поэтому интерфейс должен быть по-прежнему реализован в классе, если требуется получить его экземпляр, несмотря на возможность определять в интерфейсе методы по умолчанию, начиная с версии JDK 8.

И последнее замечание: как правило, методы по умолчанию служат специальным целям. А создаваемые интерфейсы по-прежнему определяют, главным образом, что именно следует сделать, но не *как* это сделать. Тем не менее внедрение методов по умолчанию доставляет дополнительные удобства для программирования на Java.

Основы применения методов по умолчанию

Метод по умолчанию определяется в интерфейсе таким же образом, как и метод в классе. Главное отличие состоит в том, что в начале объявления метода по умолчанию указывается ключевое слово `default`. Рассмотрим в качестве примера следующий простой интерфейс:

```
public interface MyIF {  
    // Это объявление обычного метода в интерфейсе.  
    // Он НЕ предоставляет реализацию по умолчанию  
    int getNumber();  
  
    // А это объявление метода по умолчанию. Обратите  
    // внимание на его реализацию по умолчанию  
    default String getString() {  
        return "Объект типа String по умолчанию";  
    }  
}
```

В интерфейсе `MyIF` объявляются два метода. Первый из них, `getNumber()`, является стандартно объявляемым в интерфейсе и вообще не предоставляет никакой реализации. А второй метод, `getString()`, включает в себя реализацию по умолчанию. В данном случае он просто возвращает символьную строку "Объект типа `String` по умолчанию". Обратите особое внимание на порядок объявления метода `getString()`, где в самом начале указан модификатор доступа `default`. Этот синтаксис можно обобщить. В частности, для того чтобы определить метод по умолчанию, его объявление достаточно предварить ключевым словом `default`.

В связи с тем что в объявление метода `getString()` включена его реализация по умолчанию, его совсем не обязательно переопределять в классе, реализующем интерфейс `MyIF`. Например, объявление приведенного ниже класса `MyIFImp` вполне допустимо.

```
// Реализовать интерфейс MyIF  
class MyIFImp implements MyIF {  
    // В этом классе должен быть реализован только метод getNumber(),  
    // определенный в интерфейсе MyIF.  
    // А вызов метода getString() разрешается по умолчанию  
    public int getNumber() {  
        return 100;  
    }  
}
```

В следующем примере кода получается экземпляр класса `MyIFImp`, который используется для вызова обоих методов, `getNumber()` и `getString()`:

```
// Использовать метод по умолчанию  
class DefaultMethodDemo {  
    public static void main(String args[]) {  
  
        MyIFImp obj = new MyIFImp();  
  
        // Метод getNumber() можно вызвать, т.к. он явно реализован  
        // в классе MyIFImp:  
        System.out.println(obj.getNumber());  
  
        // Метод getString() также можно вызвать, т.к. в интерфейсе  
        // имеется его реализация по умолчанию:
```



```

        System.out.println(obj.getString());
    }
}

```

Ниже приведен результат, выводимый при выполнении данного кода.

```

100
Объект типа String по умолчанию

```

Как видите, в данном примере кода автоматически используется реализация метода `getString()` по умолчанию. Определять и тем более реализовывать этот метод в классе `MyIFImp` совсем не обязательно. (Разумеется, реализация метода `getString()` в классе *потребуется* в том случае, если его предполагается использовать в этом классе для каких-нибудь других целей, а не только по умолчанию.)

В классе, реализующем интерфейс, вполне возможно определять собственную реализацию метода по умолчанию. Например, метод `getString()` переопределяется в классе `MyIFImp2`, как показано ниже. Теперь в результате вызова метода `getString()` возвращается другая символьная строка.

```

class MyIFImp2 implements MyIF {
    // В этом классе предоставляются реализации обоих методов
    // getNumber() и getString()
    public int getNumber() {
        return 100;
    }

    public String getString() {
        return "Это другая символьная строка.";
    }
}

```

Более практический пример

Несмотря на то что в приведенном выше примере демонстрируется механизм применения методов по умолчанию, в нем все же не показана практическая польза от такого нововведения. С этой целью вернемся к интерфейсу `IntStack`, рассмотренному ранее в этой главе. Ради удобства обсуждения допустим, что интерфейс `IntStack` широко применяется во многих программах и что в нем требуется ввести метод, очищающий стек, чтобы подготовить его к повторному использованию. Следовательно, интерфейс `IntStack` требуется дополнить новыми функциональными возможностями, но не нарушая уже существующий код. Прежде это было просто невозможно, но благодаря внедрению методов по умолчанию это совсем не трудно сделать теперь. Например, интерфейс `IntStack` можно усовершенствовать следующим образом:

```

interface IntStack {
    void push(int item); // сохранить элемент в стеке
    int pop(); // извлечь элемент из стека

    // У метода clear() теперь имеется вариант по умолчанию, поэтому
    // его придется реализовывать в том существующем классе, где уже
    // применяется интерфейс IntStack
    default void clear() {
        System.out.println("Метод clear() не реализован.");
    }
}

```

В данном примере метод `clear()` по умолчанию выводит сообщение о том, что он не реализован. И это вполне допустимо, поскольку метод `clear()` нельзя вызывать из любого уже существующего класса, реализующего интерфейс `IntStack`, если он не был определен в предыдущей версии интерфейса `IntStack`. Но метод `clear()` может быть реализован в новом классе вместе с интерфейсом `IntStack`. Более того, новую реализацию метода `clear()` потребуется определить лишь в том случае, если он используется. Следовательно, метод по умолчанию предоставляет возможность сделать следующее:

- изящно расширить интерфейс со временем;
- предоставить дополнительные функциональные возможности, исключая замещающую реализацию в классе, если эти функциональные возможности не требуются.

Следует также иметь в виду, что в реальном коде метод `clear()` должен генерировать исключение, а не выводить сообщение. Об исключениях речь пойдет в следующей главе. Проработав материал этой главы, попробуйте видоизменить реализацию метода `clear()` по умолчанию таким образом, чтобы он генерировал исключение типа `UnsupportedOperationException`.

Вопросы множественного наследования

Как пояснялось ранее в этой книге, множественное наследование классов в Java не поддерживается. Но теперь, когда в интерфейсы внедрены методы по умолчанию, может возникнуть вопрос: позволяет ли интерфейс обойти это ограничение? По существу, позволяет. Напомним о следующем главном отличии класса от интерфейса: в классе могут сохраняться данные состояния, особенно с помощью переменных экземпляра, тогда как в интерфейсе этого сделать нельзя.

Несмотря на все сказанное выше, методы по умолчанию предоставляют отчасти возможности, которые обычно связываются с понятием множественного наследования. Например, в одном классе можно реализовать два интерфейса. Если в каждом из этих интерфейсов предоставляются методы по умолчанию, то некоторое поведение наследуется от обоих интерфейсов. Поэтому в какой-то, хотя и ограниченной, степени эти методы все же поддерживают множественное наследование. Нетрудно догадаться, что в подобных случаях может возникнуть конфликт имен.

Допустим, два интерфейса, `Alpha` и `Beta`, реализуются в классе `MyClass`. Что, если в обоих этих интерфейсах предоставляется метод `reset()`, объявляемый с реализацией по умолчанию? Какой из вариантов этого метода будет выбран в классе `MyClass`: из интерфейса `Alpha` или `Beta`? С другой стороны, рассмотрим ситуацию, когда интерфейс `Beta` расширяет интерфейс `Alpha`. Какой вариант метода по умолчанию используется в этом случае? А что, если в классе `MyClass` предоставляется собственная реализация этого метода? Для ответа на эти и другие аналогичные вопросы в Java определен ряд правил разрешения подобных конфликтов.

Во-первых, во всех подобных случаях приоритет отдается реализации метода в классе над его реализацией в интерфейсе. Так, если в классе `MyClass` переопределяется метод по умолчанию `reset()`, то выбирается его вариант, реализуемый

в классе `MyClass`. Это происходит даже в том случае, если в классе `MyClass` реализуются оба интерфейса, `Alpha` и `Beta`. И это означает, что методы по умолчанию переопределяются их реализацией в классе `MyClass`.

Во-вторых, если в классе реализуются два интерфейса с одинаковым методом по умолчанию, но этот метод не переопределяется в данном классе, то возникает ошибка. Если же в классе `MyClass` реализуются оба интерфейса, `Alpha` и `Beta`, но метод `reset()` в нем не переопределяется, то и в этом случае возникает ошибка.

В тех случаях, когда один интерфейс наследует другой и в обоих интерфейсах определяется общий метод по умолчанию, предпочтение отдается варианту метода из наследующего интерфейса. Так, если интерфейс `Beta` расширяет интерфейс `Alpha`, то используется вариант метода `reset()` из интерфейса `Beta`. Впрочем, используя новую форму ключевого слова `super`, вполне возможно сослаться на реализацию по умолчанию в наследуемом интерфейсе. Эта общая форма ключевого слова `super` выглядит следующим образом:

имя_интерфейса.super.имя_метода()

Так, если из интерфейса `Beta` требуется обратиться по ссылке к методу по умолчанию `reset()` в интерфейсе `Alpha`, то для этого достаточно воспользоваться следующим оператором:

```
Alpha.super.reset();
```

Применение статических методов в интерфейсе

Начиная с версии JDK 8, у интерфейсов появилась еще одна возможность: определять в нем один или несколько статических методов. Аналогично статическим методам в классе, метод, объявляемый как `static` в интерфейсе, можно вызывать независимо от любого объекта. И для этого не требуется ни реализация такого метода в интерфейсе, ни экземпляр самого интерфейса. Напротив, для вызова статического метода достаточно указать имя интерфейса и через точку имя самого метода. Ниже приведена общая форма вызова статического метода из интерфейса. Обратите внимание на ее сходство с формой вызова статического метода из класса.

имя_интерфейса.имя_статического_метода

В приведенном ниже примере кода демонстрируется ввод статического метода `getDefaultNumber()` в упоминавшийся ранее интерфейс `MyIF`. Этот метод возвращает нулевое значение.

```
public interface MyIF {
    // Это объявление обычного метода в интерфейсе.
    // Он НЕ предоставляет реализацию по умолчанию
    int getNumber();

    // А это объявление метода по умолчанию. Обратите
    // внимание на его реализацию по умолчанию
    default String getString() {
        return "Объект типа String по умолчанию";
    }
}
```

```
// Это объявление статического метода в интерфейсе
static int getDefaultNumber() {
    return 0;
}
}
```

Метод `getDefaultNumber()` может быть вызван следующим образом:

```
int defNum = MyIF.getDefaultNumber();
```

Как упоминалось выше, для вызова метода `getDefaultNumber()` реализация или экземпляр интерфейса `MyIF` не требуется, поскольку это статический метод. И последнее замечание: статические методы из интерфейсов не наследуются ни реализующими их классами, ни подчиненными интерфейсами.

Заключительные соображения по поводу пакетов и интерфейсов

В примерах, представленных в этой книге, пакеты и или интерфейсы используются нечасто. Тем не менее оба эти языковые средства являются очень важной составляющей среды программирования на Java. Буквально все реальные программы, которые придется писать на Java, будут входить в состав пакетов и скорее всего будут реализовывать интерфейсы. Поэтому очень важно освоить пакет и интерфейсы настолько, чтобы свободно пользоваться этими языковыми средствами, программируя на Java.

Обработка исключений

В этой главе рассматривается механизм обработки исключений в Java. *Исключение* — это ненормальная ситуация, возникающая во время выполнения последовательности кода. Иными словами, исключение — это ошибка, возникающая во время выполнения. В тех языках программирования, где не поддерживается обработка исключений, ошибки должны проверяться и обрабатываться вручную и, как правило, благодаря использованию кодов ошибок и т.п. Такой подход временительный и хлопотный. Обработка исключений в Java позволяет избежать подобных трудностей, а кроме того, переносит управление ошибками, возникающими во время выполнения, в область ООП.

Основы обработки исключений

Исключение в Java представляет собой объект, описывающий исключительную (т.е. ошибочную) ситуацию, возникающую в определенной части программного кода. Когда возникает такая ситуация, в вызвавшем ошибку методе *генерируется* объект, который представляет исключение. Этот метод может обработать исключение самостоятельно или же пропустить его. Так или иначе, в определенный момент исключение *перехватывается* и *обрабатывается*. Исключения могут генерироваться автоматически исполняющей системой Java или вручную в прикладном коде. Исключения, генерируемые исполняющей системой Java, имеют отношение к фундаментальным ошибкам, нарушающим правила языка Java или ограничения, накладываемые исполняющей системой Java. А исключения, генерируемые вручную, обычно служат для уведомления вызывающего кода о некоторых ошибках в вызываемом методе.

Управление обработкой исключений в Java осуществляется с помощью пяти ключевых слов: `try`, `catch`, `throw`, `throws` и `finally`. Если говорить вкратце, то эти ключевые слова действуют следующим образом. Операторы программы, которые требуется отслеживать на предмет исключений, размещаются в блоке `try`. Если исключение возникает в блоке `try`, оно генерируется. Прикладной код может перехватить исключение, используя блок `catch`, а затем обработать его некоторым рациональным способом. Системные исключения автоматически генерируются исполняющей системой Java. Для генерирования исключения вручную служит ключевое слово `throw`. Любое исключение, генерируемое в теле метода,

должно быть обозначено в его объявлении ключевым словом `throws`. А любой код, который должен быть непременно выполнен по завершении блока `try`, размещается в блоке `finally`. Ниже приведена общая форма блока обработки исключений.

```
try {
    // блок кода, в котором отслеживаются ошибки
}
catch (тип_исключения_1 exOb) {
    // обработчик исключений тип_исключения_1
}
catch (тип_исключения_2 exOb) {
    // обработчик исключений тип_исключения_2
}
// ...
finally {
    // блок кода, который должен быть непременно
    // выполнен по завершении блока try
}
```

Здесь `тип_исключения` обозначает тип происходящего исключения. В остальной части этой главы описывается, каким образом приведенная выше языковая структура применяется для обработки исключений.

На заметку! В версии JDK 7 внедрена новая форма оператора `try`, обеспечивающая автоматическое управление ресурсами. Эта форма называется оператором `try с ресурсами` и подробнее рассматривается в главе 13 в контексте управления файлами, поскольку файлы относятся к наиболее часто используемым ресурсам.

Типы исключений

Все типы исключений являются подклассами, производными от встроенного класса `Throwable`. Это означает, что класс `Throwable` находится на вершине иерархии классов исключений. Сразу же за классом `Throwable` ниже по иерархии следуют два подкласса, разделяющие все исключения на две отдельные ветви. Одну ветвь возглавляет класс `Exception`. Он служит для исключительных условий, которые должна перехватывать прикладная программа. Именно от этого класса вам и предстоит наследовать свои подклассы при создании собственных типов исключений. У класса `Exception` имеется важный подкласс — `RuntimeException`. Исключения этого типа автоматически определяются для создаваемых вами прикладных программ и охватывают такие ошибки, как деление на ноль и ошибочная индексация массивов.

Другая ветвь возглавляется классом `Error`, определяющим исключения, появление которых не предполагается при нормальном выполнении программы. Исключения типа `Error` используются в исполняющей системе Java для обозначения ошибок, происходящих в самой исполняющей среде. Примером такой ошибки может служить переполнение стека. В этой главе не рассматриваются исключения типа `Error`, поскольку они, как правило, возникают в связи с аварийными сбоями, которые не могут быть обработаны в прикладной программе. Верхний уровень иерархии исключений представлен на рис. 10.1.

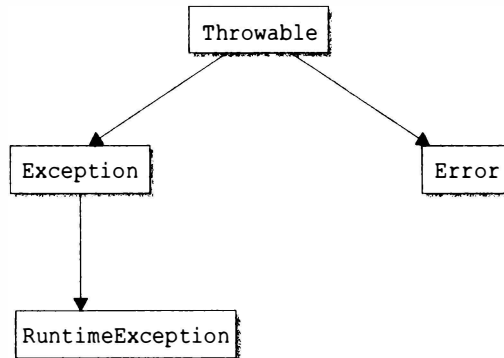


Рис. 10.1. Верхний уровень иерархии исключений

Необработываемые исключения

Прежде чем перейти непосредственно к обработке исключений, имеет смысл продемонстрировать, что происходит, когда исключения не обрабатываются. В приведенный ниже пример небольшой программы намеренно введен оператор, вызывающий ошибку деления на ноль.

```
class Exc0 {  
    public static void main(String args[]) {  
        int d = 0;  
        int a = 42 / d;  
    }  
}
```

Когда исполняющая система Java обнаруживает попытку деления на ноль, она создает новый объект исключения, а затем *генерирует* исключение. Это прерывает выполнение класса `Exc0`, ведь как только исключение сгенерировано, оно должно быть *перехвачено* обработчиком исключений и немедленно обработано. В данном примере обработчик исключений отсутствует, и поэтому исключение перехватывается стандартным обработчиком, предоставляемым исполняющей системой Java. Любое исключение, не перехваченное прикладной программой, в конечном итоге будет перехвачено и обработано этим стандартным обработчиком. Стандартный обработчик выводит символьную строку с описанием исключения и результат трассировки стека, начиная с момента возникновения исключения, а затем прерывает выполнение программы. Ниже приведен пример исключения, сгенерированного при выполнении приведенного выше кода.

```
java.lang.ArithmeticException: / by zero  
    at Exc0.main(Exc0.java:4)
```

Обратите внимание на то, что в результат трассировки стека включены имена класса `Exc0`, метода `main()`, файла `Exc0.java` и номер 4 строки кода. Следует также иметь в виду, что сгенерированное исключение относится к подклассу `ArithmeticException`, производному от класса `Exception` и точнее описывающему тип возникшей ошибки. Как будет показано далее в этой главе, в Java предо-

ставляется несколько встроенных типов исключений, соответствующих разным типам ошибок, которые могут возникнуть во время выполнения.

Трассировка стека позволяет проследить последовательность вызовов методов, которые привели к ошибке. В качестве примера ниже приведена другая версия предыдущей программы, где вносится та же самая ошибка, но уже не в методе `main()`, а в другом методе.

```
class Excl {
    static void subroutine() {
        int d = 0;
        int a = 10 / d;
    }
    public static void main(String args[]) {
        Excl.subroutine();
    }
}
```

Результат трассировки стека стандартного обработчика исключений отображает весь стек вызовов следующим образом:

```
java.lang.ArithmeticException: / by zero
    at Excl.subroutine(Excl.java:4)
    at Excl.main(Excl.java:7)
```

Как видите, на дне стека находится седьмая строка кода из метода `main()`, в которой делается вызов метода `subroutine()`, вызвавший исключение при выполнении четвертой строки кода. Трассировкой стека удобно пользоваться для отладки, поскольку ее результат показывает всю последовательность вызовов, приведших к ошибке.

Использование блоков операторов `try` и `catch`

Стандартный обработчик исключений, предоставляемый исполняющей системой Java, безусловно, удобен для отладки, но, как правило, обрабатывать исключения приходится вручную. Это дает два существенных преимущества. Во-первых, появляется возможность исправить ошибку. И во-вторых, предотвращается автоматическое прерывание выполнения программы. Большинство пользователей будут, по меньшей мере, недовольны, если программа будет прерываться и выводить результат трассировки стека всякий раз, когда возникает ошибка. Правда, предотвратить это совсем нетрудно.

Чтобы застраховаться от подобных сбойных ситуаций и организовать обработку ошибок, возникающих во время выполнения, достаточно разместить контролируемый код в блоке оператора `try`. Сразу же за блоком оператора `try` должен следовать блок оператора `catch`, где указывается тип перехватываемого исключения. Для того чтобы показать, насколько просто это делается, в следующий пример программы включены блоки операторов `try` и `catch` для обработки исключения типа `ArithmeticException`, генерируемого при попытке деления на ноль:

```
class Exc2 {
    public static void main(String args[]) {
        int d, a;
```

```
try { // проконтролировать блок кода
    d = 0;
    a = 42 / d;
    System.out.println("Это не будет выведено.");
} catch (ArithmeticException e) { // перехватить ошибку
    // деления на ноль
    System.out.println("Деление на ноль.");
}
System.out.println("После оператора catch.");
}
```

Эта программа выводит следующий результат:

Деление на ноль.
После оператора catch.

Обратите внимание на то, что вызов метода `println()` в блоке оператора `try` вообще не будет выполняться. Как только возникнет исключение, управление сразу же передается из блока оператора `try` в блок оператора `catch`. Это означает, что символьная строка "Это не будет выведено" не выводится. По завершении блока оператора `catch` управление передается в строку кода, следующую после всего блока операторов `try/catch`.

Операторы `try` и `catch` составляют единое целое. Область действия блока оператора `catch` не распространяется на операторы, предшествующие блоку оператора `try`. Оператор `catch` не в состоянии перехватить исключение, переданное другим оператором `try`, кроме описываемых далее конструкций вложенных операторов `try`. Операторы, защищаемые блоком оператора `try`, должны быть заключены в фигурные скобки (т.е. должны находиться в самом блоке). Оператор `try` нельзя применять к отдельному оператору в исходном коде программы.

Целью большинства правильно построенных операторов `catch` является разрешение исключительных ситуаций и продолжение нормальной работы программы, как если бы ошибки вообще не было. В приведенном ниже примере программы на каждом шаге цикла `for` получаются два случайных числа. Эти два числа делятся одно на другое, а результат используется для деления числового значения **12345**. Окончательный результат размещается в переменной `a`. Если какая-нибудь из этих операций деления приводит к ошибке деления на ноль, эта ошибка перехватывается, в переменной `a` устанавливается нулевое значение и программа продолжает выполняться дальше.

```
// Обработать исключение и продолжить работу
import java.util.Random;

class HandleError {
    public static void main(String args[]) {
        int a=0, b=0, c=0;
        Random r = new Random();

        for(int i=0; i<32000; i++) {
            try {
                b = r.nextInt();
                c = r.nextInt();
                a = 12345 / (b/c);
            } catch (ArithmeticException e) {
```

```

        System.out.println("Деление на ноль.");
        a = 0; // присвоить ноль и продолжить работу
    }
    System.out.println("a: " + a);
}
}
}

```

Вывод описания исключения

В классе `Throwable` переопределяется метод `toString()`, определенный в классе `Object` таким образом, чтобы возвращать символьную строку, содержащую описание исключения. Это описание можно вывести с помощью метода `println()`, просто передав ему исключение в виде аргумента. Например, блок оператора `catch` из предыдущего примера может быть переписан следующим образом:

```

catch (ArithmeticException e) {
    System.out.println("Исключение: " + e);
    a = 0; // присвоить ноль и продолжить работу
}

```

Когда эта версия блока оператора `catch` подставляется в программу и программа запускается, то при любой попытке деления на ноль выводится следующее сообщение:

Исключение: `java.lang.ArithmeticException: / by zero`

И хотя в данном контексте это не имеет особого значения, тем не менее возможность вывести описание исключения в некоторых случаях оказывается весьма кстати, особенно на стадии экспериментирования с исключениями или отладки программы.

Применение нескольких операторов `catch`

Иногда в одном фрагменте кода может возникнуть не одно исключение. Чтобы справиться с такой ситуацией, можно указать два или больше оператора `catch`, каждый из которых предназначается для перехвата отдельного типа исключения. Когда генерируется исключение, каждый оператор `catch` проверяется по порядку и выполняется тот из них, который совпадает по типу с возникшим исключением. По завершении одного из операторов `catch` все остальные пропускаются и выполнение программы продолжается с оператора, следующего сразу после блока операторов `try/catch`. В следующем примере программы перехватываются два разных типа исключений:

```

// Продемонстрировать применение нескольких операторов catch
class MultipleCatches {
    public static void main(String args[]) {
        try {
            int a = args.length;
            System.out.println("a = " + a);
            int b = 42 / a;
            int c[] = { 1 };
            c[42] = 99;

```

```

    } catch(ArithmeticException e) {
        System.out.println("Деление на ноль: " + e);
    } catch(ArrayIndexOutOfBoundsException e) {
        System.out.println(
            "Ошибка индексации за пределами массива: " + e);
    }
    System.out.println("После блока операторов try/catch.");
}
}

```

В этой программе произойдет исключение в связи с делением на ноль, если она будет запущена без аргументов командной строки. Ведь в этом случае значение переменной `a` будет равно нулю. Деление будет выполнено нормально, если программе будет передан аргумент командной строки, устанавливающий в переменной `a` значение больше нуля. Но в этом случае возникнет исключение типа `ArrayIndexOutOfBoundsException`, поскольку длина массива целых чисел `c` равна 1, тогда как программа пытается присвоить значение элементу массива `c[42]`.

Ниже приведены результаты выполнения данной программы обоими способами.

```

C:\>java MultipleCatches
a = 0
Деление на ноль: java.lang.ArithmeticException: / by zero
После блока операторов try/catch.

```

```

C:\>java MultipleCatches TestArg
a = 1
Ошибка индексации за пределами массива:
    java.lang.ArrayIndexOutOfBoundsException:42
После блока операторов try/catch.

```

Применяя несколько операторов `catch`, важно помнить, что перехват исключений из подклассов должен следовать до перехвата исключений из суперклассов. Дело в том, что оператор `catch`, в котором перехватывается исключение из суперкласса, будет перехватывать все исключения из этого суперкласса, а также все исключения из его подклассов. Это означает, что исключения из подкласса вообще не будут обработаны, если попытаться перехватить их после исключений из его суперкласса. Кроме того, недостижимый код считается в Java ошибкой. Рассмотрим в качестве примера следующую программу:

/* Эта программа содержит ошибку.

В последовательности операторов **catch** подкласс исключений должен быть указан перед его суперклассом, иначе это приведет к недостижимому коду и ошибке во время компиляции.

```

*/
class SuperSubCatch {
    public static void main(String args[]) {
        try {
            int a = 0;
            int b = 42 / a;
        } catch(Exception e) {
            System.out.println(
                "Перехват исключений общего класса Exception.");
        }
        /* Этот оператор catch вообще не будет достигнут, т.к.
           подкласс ArithmeticException является производным
           от класса Exception. */
    }
}

```

```

        catch(ArithmeticException e) { // ОШИБКА: недостижимый код!
            System.out.println("Этот код вообще недостижим.");
        }
    }
}

```

Если попытаться скомпилировать эту программу, то появится сообщение об ошибке, уведомляющее, что второй оператор `catch` недостижим, потому что исключение уже перехвачено. Класс исключения типа `ArithmeticException` является производным от класса `Exception`, и поэтому первый оператор `catch` обработает все ошибки, относящиеся к классу `Exception`, включая и класс `ArithmeticException`. Это означает, что второй оператор `catch` так и не будет выполнен. Чтобы исправить это положение, придется изменить порядок следования операторов `catch`.

Вложенные операторы `try`

Операторы `try` могут быть вложенными. Это означает, что один оператор `try` может находиться в блоке другого оператора `try`. Всякий раз, когда управление передается блоку оператора `try`, контекст соответствующего исключения размещается в стеке. Если во вложенном операторе `try` отсутствует оператор `catch` для перехвата и обработки конкретного исключения, стек разворачивается и проверяется на соответствие оператор `catch` из внешнего блока оператора `try`, и так до тех пор, пока не будет найден подходящий оператор `catch` или не будут исчерпаны все уровни вложенности операторов `try`. Если подходящий оператор `catch` не будет найден, то возникшее исключение обработает исполняющая система Java. Ниже приведен пример программы, демонстрирующий применение вложенных операторов `try`.

```

// Пример применения вложенных операторов try
class NestTry {
    public static void main(String args[]) {
        try {
            int a = args.length;

            /* Если не указаны аргументы командной строки,
             * в следующем операторе будет сгенерировано
             * исключение в связи с делением на ноль. */
            int b = 42 / a;

            System.out.println("a = " + a);

            try { // вложенный блок try
                /* Если указан один аргумент командной строки,
                 * то исключение в связи с делением на ноль
                 * будет сгенерировано в следующем коде. */
                if(a==1) a = a/(a-a); // деление на ноль
                /* Если указаны два аргумента командной строки,
                 * то генерируется исключение в связи с выходом
                 * за пределы массива.
                 */
                if(a==2) {
                    int c[] = { 1 };
                    c[42] = 99; // здесь генерируется исключение в связи

```

```

        // с выходом за пределы массива
    }
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Индекс за пределами массива: " + e);
    }
    } catch (ArithmeticException e) {
        System.out.println("Деление на ноль: " + e);
    }
}
}

```

Как видите, в этой программе один блок `try` вложен в другой. Программа работает следующим образом. Когда она запускается на выполнение без аргументов командной строки, во внешнем блоке оператора `try` генерируется исключение в связи с делением на ноль. А если программа запускается на выполнение с одним аргументом, то исключение в связи с делением на ноль генерируется во вложенном блоке оператора `try`. Но поскольку это исключение не обрабатывается во вложенном блоке, то оно передается внешнему блоку оператора `try`, где и обрабатывается. Если же программе передаются два аргумента командной строки, то во внутреннем блоке оператора `try` генерируется исключение в связи с выходом индекса за пределы массива. Ниже приведены результаты выполнения этой программы в каждом из трех случаев ее запуска на выполнение.

```

C:\>java NestTry
Деление на ноль: java.lang.ArithmeticException: / by zero

```

```

C:\>java NestTry One
a = 1
Деление на ноль: java.lang.ArithmeticException: / by zero

```

```

C:\>java NestTry One Two
a = 2
Индекс за пределами массива: java.lang.ArrayIndexOutOfBoundsException:42

```

Вложение операторов `try` может быть не столь очевидным при вызовах методов. Например, вызов метода можно заключить в блок оператора `try`, а в теле этого метода организовать еще один блок оператора `try`. В этом случае блок оператора `try` в теле метода оказывается вложенным во внешний блок оператора `try`, откуда вызывается этот метод. Ниже приведена версия предыдущей программы, где блок вложенного оператора `try` перемещен в тело метода `nesttry()`.

```

/* Операторы try могут быть неявно вложены в вызовы методов. */
class MethNestTry {
    static void nesttry(int a) {
        try { // вложенный блок оператора try
            /* Если указан один аргумент командной строки,
               то исключение в связи с делением на ноль
               будет сгенерировано в следующем коде. */
            if(a==1) a = a/(a-a); // деление на ноль

            /* Если указаны два аргумента командной строки,
               то генерируется исключение в связи с выходом
               за пределы массива.
            */
            if(a==2) {
                int c[] = { 1 };
            }
        }
    }
}

```

```

        c[42] = 99; // здесь генерируется исключение в связи
                   // с выходом за пределы массива
    }
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Индекс за пределами массива: " + e);
}

public static void main(String args[]) {
    try {
        int a = args.length;

        /* Если не указаны аргументы командной строки,
           в следующем операторе будет сгенерировано
           исключение в связи с делением на нуль. */
        int b = 42 / a;
        System.out.println("a = " + a);
        nesttry(a);
    } catch (ArithmeticException e) {
        System.out.println("Деление на нуль: " + e);
    }
}
}

```

Эта версия программы выводит такой же результат, как и предыдущая.

Оператор throw

В приведенных до сих пор примерах перехватывались только те исключения, которые генерировала исполняющая система Java. Но исключения можно генерировать и непосредственно в прикладной программе, используя оператор `throw`. Его общая форма выглядит следующим образом:

throw генерируемый_экземпляр;

где *генерируемый_экземпляр* должен быть объектом класса `Throwable` или производного от него подкласса. Примитивные типы вроде `int` или `char`, а также классы, кроме `Throwable`, например `String` или `Object`, нельзя использовать для генерирования исключений. Получить объект класса `Throwable` можно двумя способами, указав соответствующий параметр в операторе `catch` или создав этот объект с помощью оператора `new`.

Поток исполнения программы останавливается сразу же после оператора `throw`, и все последующие операторы не выполняются. В этом случае ближайший объемлющий блок оператора `try` проверяется на наличие оператора `catch` с совпадающим типом исключения. Если совпадение обнаружено, управление передается этому оператору. В противном случае проверяется следующий внешний блок оператора `try` и т.д. Если же не удастся найти оператор `catch`, совпадающий с типом исключения, то стандартный обработчик исключений прерывает выполнение программы и выводит результат трассировки стека.

Ниже приведен пример программы, в которой генерируется исключение. Обработчик, перехватывающий это исключение, повторно генерирует его для внешнего обработчика.

```
// Продемонстрировать применение оператора throw
class ThrowDemo {
    static void demoproc() {
        try {
            throw new NullPointerException("демонстрация");
        } catch (NullPointerException e) {
            System.out.println(
                "Исключение перехвачено в теле метода demoproc().");
            throw e; // повторно сгенерировать исключение
        }
    }

    public static void main(String args[]) {
        try {
            demoproc();
        } catch (NullPointerException e) {
            System.out.println("Повторный перехват: " + e);
        }
    }
}
```

Эта программа получает две возможности для обработки одной и той же ошибки. Сначала в методе `main()` устанавливается контекст исключения, затем вызывается метод `demoproc()`, где задается другой контекст обработки исключения и сразу же генерируется новый экземпляр исключения типа `NullPointerException`, который перехватывается в следующей строке кода. Затем исключение генерируется повторно. Ниже приведен результат, выводимый этой программой.

Исключение перехвачено в теле метода `demoproc()`.
Повторный перехват: `java.lang.NullPointerException: демонстрация`

В этом примере программы демонстрируется также, каким образом создаются объекты стандартных исключений в Java. Обратите внимание на следующую строку кода:

```
throw new NullPointerException("демонстрация");
```

Здесь оператор `new` служит для создания экземпляра исключения типа `NullPointerException`. Многие классы встроенных в Java исключений, возникающих во время выполнения, имеют по меньшей мере две формы конструктора: без параметров и со строковым параметром. Если применяется вторая форма конструктора, его аргумент обозначает символьную строку, описывающую исключение. Эта символьная строка выводится, когда объект исключения передается в качестве аргумента методу `print()` или `println()`. Она может быть также получена в результате вызова метода `getMessage()`, определенного в классе `Throwable`.

Оператор `throws`

Если метод способен вызвать исключение, которое он сам не обрабатывает, то он должен задать свое поведение таким образом, чтобы вызывающий его код мог обезопасить себя от такого исключения. С этой целью в объявление метода вводится оператор `throws`, где перечисляются типы исключений, которые метод может генерировать. Это обязательно для всех исключений, кроме тех, кото-

рые относятся к классам `Error` и `RuntimeException` или любым их подклассам. Все остальные исключения, которые может сгенерировать метод, должны быть объявлены в операторе `throws`. Если этого не сделать, то во время компиляции возникнет ошибка.

Ниже приведена общая форма объявления метода, которая включает оператор `throws`.

```
тип имя_метода(список_параметров) throws список_исключений
{
    // тело метода
}
```

Здесь *список_исключений* обозначает разделяемый запятыми список исключений, которые метод может сгенерировать.

Ниже приведен пример неверно написанной программы, пытающейся сгенерировать исключение, которое в самой программе не перехватывается. Эта программа не подлежит компиляции, поскольку в ней отсутствует оператор `throws`, в котором объявляется перехватываемое исключение.

```
// Эта программа содержит ошибку, и поэтому она не подлежит компиляции
class ThrowsDemo {
    static void throwOne() {
        System.out.println("В теле метода throwOne().");
        throw new IllegalAccessException("демонстрация");
    }
    public static void main(String args[]) {
        throwOne();
    }
}
```

Чтобы эту программу можно было скомпилировать, в ее исходный следует внести два изменения. Во-первых, объявить в методе `throwOne()` генерирование исключения типа `IllegalAccessException`. И во-вторых, определить в методе `main()` блок оператора `try/catch` для перехвата этого исключения. Ниже приведен исправленный код данной программы.

```
// Эта программа написана верно
class ThrowsDemo {
    static void throwOne() throws IllegalAccessException {
        System.out.println("В теле метода throwOne().");
        throw new IllegalAccessException("демонстрация");
    }
    public static void main(String args[]) {
        try {
            throwOne();
        } catch (IllegalAccessException e) {
            System.out.println("Перехвачено исключение: " + e);
        }
    }
}
```

Результат, выводимый этой программой, выглядит следующим образом:

```
В теле метода throwOne().
Перехвачено исключение: java.lang.IllegalAccessException: демонстрация
```

Оператор `finally`

Когда генерируется исключение, выполнение метода направляется по нелинейному пути, резко изменяющему нормальную последовательность выполнения операторов в теле метода. В зависимости от того, как написан метод, исключение может даже стать причиной преждевременного возврата из метода. В некоторых методах это может вызвать серьезные осложнения. Так, если файл открывается в начале метода и закрывается в конце, то вряд ли кого-нибудь устроит, что код, закрывающий файл, будет обойден механизмом обработки исключений. Для таких непредвиденных обстоятельств и служит оператор `finally`.

Оператор `finally` образует блок кода, который будет выполнен по завершении блока операторов `try/catch`, но перед следующим за ним кодом. Блок оператора `finally` выполняется независимо от того, сгенерировано ли исключение или нет. Если исключение сгенерировано, блок оператора `finally` выполняется, даже при условии, что ни один из операторов `catch` не совпадает с этим исключением. В любой момент, когда метод собирается возвратить управление вызывающему коду из блока оператора `try/catch` (через необработанное исключение или явным образом через оператор `return`), блок оператора `finally` выполняется перед возвратом управления из метода. Это может быть удобно для закрытия файловых дескрипторов либо освобождения других ресурсов, которые были выделены в начале метода и должны быть освобождены перед возвратом из него. Указывать оператор `finally` необязательно, но каждому оператору `try` требуется хотя бы один оператор `catch` или `finally`.

Ниже приведен пример программы, в котором демонстрируются три метода, возвращающих управление разными способами. Но ни в одном из них не пропускается выполнения блока оператора `finally`.

```
// Продемонстрировать применение оператора finally
class FinallyDemo {
    // сгенерировать исключение в методе
    static void procA() {
        try {
            System.out.println("В теле метода procA()");
            throw new RuntimeException("демонстрация");
        } finally {
            System.out.println(
                "Блок оператора finally в методе procA()");
        }
    }

    // вернуть управление из блока оператора try
    static void procB() {
        try {
            System.out.println("В теле метода procB()");
            return;
        } finally {
            System.out.println(
                "Блок оператора finally в методе procB()");
        }
    }
}

// выполнить блок try, как обычно
```

```

static void procC() {
    try {
        System.out.println("В теле метода procC()");
    } finally {
        System.out.println(
            "Блок оператора finally в методе procC()");
    }
}

public static void main(String args[]) {
    try {
        procA();
    } catch (Exception e) {
        System.out.println("Исключение перехвачено");
    }

    procB();
    procC();
}
}

```

В данном примере выполнение метода `procA()` преждевременно прерывается в блоке оператора `try`, где генерируется исключение, но блок оператора `finally` все равно выполняется. Выход из блока оператора `try` в методе `procB()` происходит через оператор `return`, а блок оператора `finally` выполняется перед возвратом из метода `procB()`. В методе `procC()` блок оператора `try` выполняется обычным образом, когда ошибки отсутствуют, но блок оператора `finally` выполняется все равно.

Помните! Если блок оператора `finally` связан с блоком оператора `try`, то блок оператора `finally` будет выполнен по завершении блока оператора `try`.

Ниже приведен результат, выводимый данной программой.

```

В теле метода procA()
Блок оператора finally в методе procA()
Исключение перехвачено
В теле метода procB()
Блок оператора finally в методе procB()
В теле метода procC()
Блок оператора finally в методе procC()

```

Встроенные в Java исключения

В стандартном пакете `java.lang` определен ряд классов исключений. Некоторые из них использовались в предыдущих примерах. Большинство из этих исключений относятся к подклассам стандартного типа `RuntimeException`. Как пояснялось ранее, эти исключения не обязательно включать в список оператора `throws` в объявлении метода. В языке Java такие исключения называются *непроверяемыми исключениями*, поскольку компилятор не проверяет, обрабатываются или генерируются они в каком-нибудь методе. Непроверяемые исключения, определенные в пакете `java.lang`, приведены в табл. 10.1, тогда как в табл. 10.2 — те

исключения из пакета `java.lang`, которые должны быть включены в список оператора `throws` в объявлении методов, способных генерировать их, но не обрабатывать самостоятельно. Такие исключения называются *проверяемыми*. Помимо исключений из пакета `java.lang`, в Java определен ряд дополнительных исключений, относящихся к другим стандартным пакетам.

Таблица 10.1. Подклассы непроверяемых исключений, производные от класса `RuntimeException` и определенные в пакете `java.lang`

Исключение	Описание
<code>ArithmeticException</code>	Арифметическая ошибка (например, деление на ноль)
<code>ArrayIndexOutOfBoundsException</code>	Выход индекса за пределы массива
<code>ArrayStoreException</code>	Присваивание элементу массива объекта несовместимого типа
<code>ClassCastException</code>	Неверное приведение типов
<code>EnumConstantNotPresentException</code>	Попытка воспользоваться неопределенным значением перечисления
<code>IllegalArgumentException</code>	Употребление недопустимого аргумента при вызове метода
<code>IllegalMonitorStateException</code>	Недопустимая контрольная операция (например, ожидание незаблокированного потока исполнения)
<code>IllegalStateException</code>	Неверное состояние среды или приложения
<code>IllegalThreadStateException</code>	Несовместимость запрашиваемой операции с текущим состоянием потока исполнения
<code>IndexOutOfBoundsException</code>	Выход индекса некоторого типа за допустимые пределы
<code>NegativeArraySizeException</code>	Создание массива отрицательного размера
<code>NullPointerException</code>	Неверное использование пустой ссылки
<code>NumberFormatException</code>	Неверное преобразование символьной строки в числовой формат
<code>SecurityException</code>	Попытка нарушения безопасности
<code>StringIndexOutOfBoundsException</code>	Попытка индексации за пределами символьной строки
<code>TypeNotPresentException</code>	Тип не найден
<code>UnsupportedOperationException</code>	Обнаружена неподдерживаемая операция

Таблица 10.2. Проверяемые исключения, определенные в пакете `java.lang`

Исключение	Описание
<code>ClassNotFoundException</code>	Класс не найден
<code>CloneNotSupportedException</code>	Попытка клонировать объект из класса, не реализующего интерфейс <code>Cloneable</code>
<code>IllegalAccessException</code>	Доступ к классу не разрешен
<code>InstantiationException</code>	Попытка создать объект абстрактного класса или интерфейса
<code>InterruptedException</code>	Один поток исполнения прерван другим потоком
<code>NoSuchFieldException</code>	Запрашиваемое поле не существует
<code>NoSuchMethodException</code>	Запрашиваемый метод не существует
<code>ReflectiveOperationException</code>	Суперкласс исключений, связанных с рефлексией

Создание собственных подклассов исключений

Встроенные в Java исключения позволяют обрабатывать большинство распространенных ошибок. Тем не менее в прикладных программах возможны особые ситуации, требующие наличия и обработки соответствующих исключений. Для того чтобы создать класс собственного исключения, достаточно определить его как производный от класса `Exception`, который, в свою очередь, является производным от класса `Throwable`. В подклассах собственных исключений совсем не обязательно реализовать что-нибудь. Их присутствия в системе типов уже достаточно, чтобы пользоваться ими как исключениями.

В самом классе `Exception` не определено никаких методов. Он, безусловно, наследует методы из класса `Throwable`. Таким образом, всем классам исключений, в том числе и создаваемым самостоятельно, доступны методы, определенные в классе `Throwable`. Все они перечислены в табл. 10.3. Один или несколько этих методов можно также переопределить в своих классах исключений.

В классе `Exception` определяются четыре открытых конструктора. Первые два конструктора поддерживают цепочки исключений, описываемые в следующем разделе, а два других показаны в общей форме ниже. В первой форме конструктора создается исключение без описания, а во второй форме допускается указывать описание исключения.

```
Exception()
Exception(String сообщение)
```

Зачастую указывать описание исключения непосредственно при его создании очень удобно, но иногда для этого лучше переопределить метод `toString()`. Дело в том, что в варианте метода `toString()`, определенном в классе `Throwable` и наследуемом в классе `Exception`, сначала выводится имя исключения, затем двоеточие и, наконец, описание исключения. Переопределив метод `toString()`, можно вывести только описание исключения, отбросив его имя и двоеточие. В итоге выводимый результат получается более ясным, что иногда весьма желательно.

Таблица 10.3. Методы, определенные в классе **Throwable**

Метод	Описание
final void addSuppressed (Throwable <i>исключение</i>)	Добавляет заданное <i>исключение</i> в список подавляемых исключений, связанный с вызывающим исключением. Предназначен главным образом для применения в операторе try с ресурсами
Throwable fillInStackTrace ()	Возвращает объект класса Throwable , содержащий полную трассировку стека. Этот объект может быть сгенерирован повторно
Throwable getCause ()	Возвращает исключение, положенное в основу текущего исключения. Если такое исключение отсутствует, то возвращается пустое значение null
String getLocalizedMessage ()	Возвращает локализованное описание исключения
String getMessage ()	Возвращает описание исключения
StackTraceElement [] getStackTrace ()	Возвращает массив, содержащий поэлементную трассировку стека в виде объектов класса StackTraceElement . На вершине стека находится метод, который был вызван непосредственно перед генерированием исключения. Этот метод содержится в первом элементе массива. Класс StackTraceElement предоставляет доступ к данным о каждом элементе трассировки, например, к имени вызванного метода
final Throwable [] getSuppressed ()	Получает подавленные исключения, связанные с вызывающим исключением, и возвращает массив, содержащий результат. Подавленные исключения генерируются главным образом в операторе try с ресурсами
Throwable initCause (Throwable <i>причина_исключения</i>)	Связывает <i>причину_исключения</i> с вызывающим исключением, указывая ее как причину этого вызывающего исключения. Возвращает ссылку на исключение ;
void printStackTrace ()	Выводит трассировку стека
void printStackTrace (PrintStream <i>поток_вывода</i>)	Направляет трассировку стека в заданный поток вывода
void printStackTrace (PrintWriter <i>поток_вывода</i>)	Направляет трассировку стека в заданный поток вывода
void setStackTrace (StackTraceElement <i>элементы</i> [])	Устанавливает трассировку стека для заданных <i>элементов</i> . Этот метод предназначен для специального, а не нормального применения
String toString ()	Возвращает объект типа String , содержащий описание исключения. Этот метод вызывается из метода println () при выводе объекта типа Throwable

В приведенном ниже примере программы сначала объявляется новый подкласс, производный от класса `Exception`, а затем он используется для вывода сообщения об ошибке в методе. В этом подклассе метод `toString()` переопределяется таким образом, чтобы вывести тщательно подготовленное описание исключения.

```
// В этой программе создается специальный тип исключения
class MyException extends Exception {
    private int detail;

    MyException(int a) {
        detail = a;
    }

    public String toString() {
        return "MyException[" + detail + "]";
    }
}

class ExceptionDemo {
    static void compute(int a) throws MyException {
        System.out.println("Вызван метод compute(" + a + ")");
        if(a > 10)
            throw new MyException(a);
        System.out.println("Нормальное завершение");
    }

    public static void main(String args[]) {
        try {
            compute(1);
            compute(20);
        } catch (MyException e) {
            System.out.println("Перехвачено исключение: " + e);
        }
    }
}
```

В данном примере определяется подкласс `MyException`, производный от класса `Exception`. Этот подкласс достаточно прост: он содержит только конструктор и переопределенный метод `toString()`, выводящий значение исключения. В классе `ExceptionDemo` определяется метод `compute()`, генерирующий объект исключения типа `MyException`. Это исключение генерируется, когда целочисленный параметр метода `compute()` принимает значение больше 10. В методе `main()` сначала устанавливается обработчик исключений типа `MyException`, затем вызывается метод `compute()` с правильным (меньше 10) и неправильным (больше 10) значением параметра, чтобы продемонстрировать оба пути выполнения кода. Ниже приведен результат, выводимый данной программой.

```
Вызван метод compute(1)
Нормальное завершение
Вызван метод compute(20)
Перехвачено исключение: MyException[20]
```

Цепочки исключений

В версию JDK 1.4 в подсистему исключений было внедрено средство, называемое *цепочками исключений*, которое позволяет связывать одно исключение с другим,

чтобы описывать в последнем причину появления первого. Допустим, в методе генерируется исключение типа `ArithmeticException` в связи с попыткой деления на нуль. Но истинная причина состоит в ошибке ввода-вывода, которая и приводит к появлению неверного делителя. И хотя этот метод должен сгенерировать исключение типа `ArithmeticException`, поскольку произошла именно эта ошибка, тем не менее вызывающему коду можно также сообщить, что основной причиной данного исключения служит ошибка ввода-вывода. Цепочки исключений позволяют справиться с этой и любой другой ситуацией, когда исключения возникают на разных уровнях.

Чтобы разрешить цепочки исключений, в класс `Throwable` были введены два конструктора и два метода. Ниже приведены общие формы конструкторов.

```
Throwable(Throwable причина_исключения)  
Throwable(String сообщение, Throwable причина_исключения)
```

В первой форме параметр *причина_исключения* обозначает исключение, вызвавшее текущее исключение. Таким образом, параметр *причина_исключения* обозначает основную причину, по которой возникло текущее исключение. Вторая форма позволяет указать описание вместе с причиной исключения. Оба эти конструктора были введены также в классы `Error`, `Exception` и `RuntimeException`.

Для организации цепочки исключений в класс `Throwable` были также введены методы `getCause()` и `initCause()`. Они перечислены в табл. 10.3 среди прочих методов, но повторяются здесь ради полноты изложения.

```
Throwable getCause()  
Throwable initCause(Throwable причина_исключения)
```

Метод `getCause()` возвращает исключение, вызывающее текущее исключение. Если же такое исключение отсутствует, то возвращается пустое значение `null`. Метод `initCause()` связывает *причину_исключения* с вызывающим исключением и возвращает ссылку на исключение. Таким образом, причину можно связать с исключением после его создания. Но причина исключения может быть установлена только один раз. Следовательно, метод `initCause()` можно вызвать для каждого объекта исключения только один раз. Даже если причина исключения была установлена в конструкторе, ее все равно нельзя установить снова, используя метод `initCause()`. Вообще говоря, метод `initCause()` служит для установки причины исключений из устаревших классов, где оба упомянутых выше дополнительных конструктора не поддерживаются.

В приведенном ниже примере программы демонстрируется применение механизма обработки цепочки исключений.

```
// Продемонстрировать цепочки исключений  
class ChainExcDemo {  
    static void demoproc() {  
  
        // создать исключение  
        NullPointerException e =  
            new NullPointerException("верхний уровень");  
  
        // добавить причину исключения  
        e.initCause(new ArithmeticException("причина"));  
    }  
}
```



```
        throw e;
    }

    public static void main(String args[]) {
        try {
            demoproc();
        } catch (NullPointerException e) {
            // вывести исключение верхнего уровня
            System.out.println("Перехвачено исключение: " + e);

            // вывести исключение, послужившее причиной
            // для исключения верхнего уровня
            System.out.println("Первопричина: " + e.getCause());
        }
    }
}
```

Эта программа выводит следующий результат:

Перехвачено исключение: java.lang.NullPointerException: верхний уровень
Первопричина: java.lang.ArithmeticException: причина

В данном примере исключение верхнего уровня относится к типу `NullPointerException`. Это исключение дополнено исключением типа `ArithmeticException`, по причине которого оно возникает. Когда исключение верхнего уровня генерируется в методе `demoproc()`, оно перехватывается в методе `main()`, где оно выводится, а вслед за ним — исключение, которое послужило причиной для его появления и которое получается в результате вызова метода `getCause()`.

Цепочки исключений могут быть составлены на любую глубину. Это означает, что причина исключения может иметь собственную причину. Но имейте в виду, что слишком длинные цепочки исключений, скорее всего, свидетельствуют о неудачном проектном решении. Цепочки исключений требуются далеко не в каждой программе. Но в тех случаях, когда сведения о причине исключения считаются полезными, цепочки исключений предлагают изящное решение.

Недавно внедренные средства для обработки исключений

В версии JDK 7 подсистема исключений в Java была дополнена тремя интересными и полезными средствами. Первое из них автоматизирует процесс освобождения такого ресурса, как файл, когда он больше не нужен. Оно опирается на расширенную форму оператора `try`, называемую оператором *try с ресурсами* и описываемую в главе 13 при обсуждении вопросов обращения с файлами. Второе средство называется *многократным перехватом*, а третье иногда называют как *окончательное повторное генерирование исключений* или *более точное повторное генерирование исключений*. Два последних средства описаны ниже.

Многократный перехват позволяет перехватывать несколько исключений в одном и том же операторе `catch`. Не так уж и редко в обработчиках нескольких исключений используется одинаковый код, хотя они реагируют на разные исключения. Вместо обработки исключений каждого типа в отдельности многократный

перехват позволяет организовать в одном блоке оператора `catch` обработку всех исключений, не дублируя код.

Чтобы организовать многократный перехват, достаточно объединить отдельные типы исключений в операторе `catch` с помощью логической операции ИЛИ. Каждый параметр многократного перехвата неявно считается завершенным. (При желании можно, конечно, явно указать ключевое слово `final`, но это совсем не обязательно.) А поскольку каждый параметр многократного перехвата неявно считается завершенным, то ему не может быть присвоено новое значение.

Ниже приведен пример оператора `catch` с многократным перехватом. В нем перехватываются оба типа исключений — `ArithmeticException` и `ArrayIndexOutOfBoundsException`.

```
catch(ArithmeticException | ArrayIndexOutOfBoundsException e) {
```

В следующем примере программы демонстрируется, каким образом многократный перехват действует на практике:

```
// Продемонстрировать многократный перехват
class MultiCatch {
    public static void main(String args[]) {
        int a=10, b=0;
        int vals[] = { 1, 2, 3 };

        try {
            int result = a / b; // сгенерировать исключение
                               // типа ArithmeticException
            // vals[10] = 19;    // сгенерировать исключение
                               // типа ArrayIndexOutOfBoundsException
            // В этом операторе catch перехватываются оба исключения
        } catch(ArithmeticException | ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Исключение перехвачено: " + e);
        }
        System.out.println("После многократного перехвата.");
    }
}
```

В данном примере программы исключение типа `ArithmeticException` генерируется при попытке деления на нуль. Если закомментировать операцию деления и удалить комментарий из следующей строки кода, будет сгенерировано исключение типа `ArrayIndexOutOfBoundsException`. Оба исключения перехватываются одним из тем же оператором `catch`.

Средство более точного повторного генерирования исключений ограничивает их тип только теми проверяемыми исключениями, которые связаны с блоком оператора `try`, где они генерируются, но не обрабатываются предыдущим оператором `catch` и являются подтипом или супертипом его параметра. Потребность обращаться к этому средству может возникать нечасто, хотя теперь оно доступно для использования. Для того чтобы средство более точного повторного генерирования исключений возымело действие, параметр оператора `catch` должен быть фактически завершенным. Это означает, что ему нельзя присваивать новое значение в блоке оператора `catch` или же он должен быть явно объявлен как `final`.

Применение исключений

Обработка исключений является эффективным механизмом управления сложными программами, обладающими многими динамическими характеристиками. Операторы `try`, `throws` и `catch` относятся к простым средствам обработки ошибок и необычных граничных условий в логике выполнения программы. В отличие от ряда других языков программирования, где в качестве признаков сбоев служат коды ошибок, в Java применяются исключения. Иными словами, когда метод может завершиться сбоем, в нем должно быть сгенерировано исключение. Это более простой способ обработки сбойных ситуаций.

И последнее замечание: операторы обработки исключений в Java не следует рассматривать как общий механизм нелокального ветвления. Употребляя их именно таким образом, можно лишь усложнить прикладной код и затруднить его сопровождение.

Многопоточное программирование

В отличие от некоторых языков программирования, в Java предоставляется встроенная поддержка *многопоточного программирования*. Многопоточная программа содержит две или несколько частей, которые могут выполняться одновременно. Каждая часть такой программы называется *поток* *исполнения*, причем каждый поток задает отдельный путь исполнения кода. Следовательно, многопоточность — это особая форма многозадачности.

Вам, вероятно всего, приходилось сталкиваться с многозадачностью на практике, поскольку она поддерживается практически во всех современных операционных системах. Тем не менее существуют два отдельных вида многозадачности: многозадачность на основе процессов и многозадачность на основе потоков. Важно понимать, в чем состоит отличие этих видов многозадачности. Большинству читателей лучше известна многозадачность на основе процессов. *Процесс*, по существу, является выполняющейся программой. Следовательно, *многозадачность на основе процессов* — это средство, которое позволяет одновременно выполнять две или несколько программ на компьютере. Так, многозадачность на основе процессов позволяет запускать компилятор Java, работая одновременно в текстовом редакторе или посещая веб-сайт. В среде многозадачности на основе процессов программа оказывается наименьшей единицей кода, которую может диспетчеризировать планировщик операционной системы.

В среде *многозадачности на основе потоков* наименьшей единицей диспетчеризируемого кода является поток исполнения. Это означает, что одна программа может выполнять две или несколько задач одновременно. Например, текстовый редактор может форматировать текст в то время, как выполняется его вывод на печать, при условии, что оба эти действия выполняются в двух отдельных потоках исполнения. Таким образом, многозадачность на основе процессов имеет дело с «общей картиной», тогда как многозадачность на основе потоков — с отдельными деталями.

Многозадачные потоки исполнения требуют меньших издержек, чем многозадачные процессы. Процессы являются крупными задачами, каждой из которых требуется свое адресное пространство. Связь между процессами ограничена и обходится дорого. Переключение контекста с одного процесса на другой также обходится дорого. С другой стороны, потоки исполнения более просты. Они совместно используют одно и то же адресное пространство и один и тот же крупный процесс. Связь между потоками исполнения обходится недорого, как, впрочем, и переключение контекста с одного потока исполнения на другой. Несмотря на то

что программы на Java пользуются многозадачными средами на основе процессов, такая многозадачность в Java не контролируется, а вот многопоточная многозадачность контролируется.

Многопоточность позволяет писать эффективные программы, максимально использующие доступные вычислительные мощности в системе. Еще одним преимуществом многопоточности является сведение к минимуму времени ожидания. Это особенно важно для интерактивных сетевых сред, где работают программы на Java, поскольку простои в таких средах — обычное явление. Например, скорость передачи данных по сети много ниже, чем скорость их обработки на компьютере. Даже чтение и запись ресурсов в локальной файловой системе выполняется много медленнее, чем их обработка на центральном процессоре (ЦП). И, конечно, пользователь много медленнее вводит данные с клавиатуры, чем их может обработать компьютер. В однопоточных средах прикладной программе приходится ожидать завершения таких задач, прежде чем переходить к следующей задаче, даже если большую часть времени программа простаивает, ожидая ввода. Многопоточность помогает сократить простои, поскольку в то время, как один поток исполнения ожидает, другой может выполняться.

Если у вас имеется опыт программирования для таких операционных систем, как Windows, значит, вы уже владеете основами многопоточного программирования. Но то обстоятельство, что в Java можно управлять потоками исполнения, делает многопоточность особенно удобной, поскольку многие мелкие вопросы ее организации решаются автоматически.

Модель потоков исполнения в Java

Исполняющая система Java во многом зависит от потоков исполнения, и все библиотеки классов разработаны с учетом многопоточности. По существу, потоки исполнения используются в Java для того, чтобы обеспечить асинхронность работы всей исполняющей среды. Благодаря предотвращению бесполезной траты циклов ЦП удастся повысить эффективность выполнения кода в целом.

Ценность многопоточной среды лучше понять в сравнении. В однопоточных системах применяется подход, называемый *циклом ожидания событий с опросом*. В этой модели единственный поток управления выполняется в бесконечном цикле, опрашивая единственную очередь событий, чтобы принять решение, что делать дальше. Как только этот механизм опроса возвращает, скажем, сигнал, что сетевой файл готов к чтению, цикл ожидания событий передает управление соответствующему обработчику событий. И до тех пор, пока тот не возвратит управление, в программе ничего не может произойти. На это расходует драгоценное время ЦП. Это может также привести к тому, что одна часть программы будет господствовать над другими, не позволяя обрабатывать любые другие события. Вообще говоря, когда в однопоточной среде поток исполнения *блокируется* (т.е. приостанавливается) по причине ожидания некоторого ресурса, то приостанавливается выполнение и всей программы.

Выгода от многопоточности состоит в том, что основной механизм циклического опроса исключается. Один поток может быть приостановлен без остано-

ки других частей программы. Например, время ожидания при чтении данных из сети или вводе пользователем данных может быть выгодно использовано в любом другом месте программы. Многопоточность позволяет переводить циклы анимации в состояние ожидания на секунду в промежутках между соседними кадрами, не приостанавливая работу всей системы. Когда поток исполнения блокируется в программе на Java, приостанавливается только один заблокированный поток, а все остальные потоки продолжают выполняться.

За последние годы многоядерные системы стали вполне обычным явлением. Безусловно, одноядерные системы все еще широко распространены и применяются. Следует, однако, иметь в виду, что многопоточные средства Java вполне работоспособны в обоих типах систем. В одноядерной системе одновременно выполняющиеся потоки совместно используют ЦП, получая каждый в отдельности некоторый квант времени ЦП. Поэтому в одноядерной системе два или более потока фактически не выполняются одновременно, но ожидают своей очереди на использование времени ЦП. А в многоядерных системах два или более потока фактически могут выполняться одновременно. Как правило, это позволяет увеличить эффективность программы и повысить скорость выполнения некоторых операций.

На заметку! В версии JDK 7 внедрен каркас Fork/Join Framework, предоставляющий мощные средства для создания многопоточных приложений с автоматическим масштабированием, позволяющим лучше использовать многоядерные системы. Каркас Fork/Join Framework является частью общей поддержки в Java *параллельного программирования*, под которым понимаются методики оптимизации некоторых видов алгоритмов параллельного выполнения в системах с несколькими процессорами. Подробнее о каркасе Fork/Join Framework и других утилитах параллелизма речь пойдет в главе 28, а здесь рассматриваются традиционные многопоточные возможности Java.

Потоки исполнения находятся в нескольких состояниях. Рассмотрим их вкратце. Поток может *выполняться*. Он может быть *готовым к выполнению*, как только получит время ЦП. Работающий поток может быть *приостановлен*, что приводит к временному прекращению его активности. Выполнение приостановленного потока может быть *возобновлено*, что позволяет продолжить его выполнение с того места, где он был приостановлен. Поток может быть *заблокирован* на время ожидания какого-нибудь ресурса. В любой момент поток может быть *прерван*, что приводит к немедленной остановке его исполнения. Однажды прерванный поток исполнения уже не может быть возобновлен.

Приоритеты потоков

Каждому потоку исполнения в Java присваивается свой приоритет, который определяет поведение данного потока по отношению к другим потокам. Приоритеты потоков исполнения задаются целыми числами, определяющими относительный приоритет одного потока над другими. Абсолютное значение приоритета еще ни о чем не говорит, поскольку высокоприоритетный поток не выполняется быстрее, чем низкоприоритетный, когда он является единственным исполняемым потоком в данный момент. Вместо этого приоритет потока исполне-

ния используется для принятия решения при переходе от одного потока исполнения к другому. Это так называемое *переключение контекста*. Правила, которые определяют, когда должно происходить переключение контекста, достаточно просты и приведены ниже.

- **Поток может добровольно уступить управление.** Для этого достаточно явно уступить очередь на исполнение, приостановить или блокировать поток на время ожидания ввода-вывода. В этом случае все прочие потоки исполнения проверяются, а ресурсы ЦП передаются потоку исполнения, имеющему наибольший приоритет и готовому к выполнению.
- **Один поток исполнения может быть вытеснен другим, более приоритетным потоком.** В этом случае низкоприоритетный поток исполнения, который не уступает ЦП, просто вытесняется высокоприоритетным потоком, независимо от того, что он делает. По существу, высокоприоритетный поток выполняется, как только это ему потребуется. Это так называемая *вытесняющая многозадачность* (или *многозадачность с приоритетами*).

Дело усложняется, если два потока исполнения имеют одинаковый приоритет и претендуют на цикл ЦП. В таких операционных системах, как Windows, потоки исполнения с одинаковым приоритетом разделяют время по кругу. А в операционных системах других типов потоки исполнения с одинаковым приоритетом должны принудительно передавать управление равноправным с ними потокам. Если они этого не делают, другие потоки исполнения не запускаются.

Внимание! В силу отличий в способах переключения потоковых контекстов в операционных системах могут возникать трудности переносимости.

Синхронизация

Многопоточность дает возможность для асинхронного поведения прикладных программ, поэтому требуется каким-то образом обеспечить синхронизацию, когда в этом возникает потребность. Так, если требуется, чтобы два потока исполнения взаимодействовали и совместно использовали сложную структуру данных вроде связанного списка, нужно найти способ предотвратить возможный конфликт между этими потоками. Это означает предотвратить запись данных в одном потоке исполнения, когда в другом потоке исполнения выполняется их чтение. Для этой цели в Java реализован изящный прием из старой модели межпроцессной синхронизации, называемый *монитором*. Монитор — это механизм управления, впервые определенный Чарльзом Энтони Ричардом Хоаром. Монитор можно рассматривать как маленький ящик, одновременно хранящий только один поток исполнения. Как только поток исполнения войдет в монитор, все другие потоки исполнения должны ожидать до тех пор, пока тот не покинет монитор. Таким образом, монитор может служить для защиты общих ресурсов от одновременного использования более чем одним потоком исполнения.

Для монитора в Java отсутствует отдельный класс вроде `Monitor`. Вместо этого у каждого объекта имеется свой неявный монитор, вход в который осуществляется

автоматически, когда для этого объекта вызывается синхронизированный метод. Когда поток исполнения находится в теле синхронизированного метода, ни один другой поток исполнения не может вызвать какой-нибудь другой синхронизированный метод для того же самого объекта. Это позволяет писать очень ясный и краткий многопоточный код, поскольку поддержка синхронизации встроена в язык.

Обмен сообщениями

Разделив программу на отдельные потоки исполнения, нужно организовать их взаимное общение. Для организации взаимодействия потоков исполнения в других языках программирования приходится опираться на средства операционной системы, а это, конечно, влечет за собой накладные расходы. Вместо этого в Java предоставляется ясный и экономичный способ организации взаимодействия двух или нескольких потоков исполнения через вызовы предопределенных методов, которые имеются у всех объектов. Система обмена сообщениями в Java позволяет потоку исполнения войти в синхронизированный метод объекта и ожидать до тех пор, пока какой-нибудь другой поток явно не уведомит его об освобождении нужных ресурсов.

Класс Thread и интерфейс Runnable

Многопоточная система в Java построена на основе класса Thread, его методах и дополняющем его интерфейсе Runnable. Класс Thread инкапсулирует поток исполнения. Обратиться напрямую к нематериальному состоянию работающего потока исполнения нельзя, поэтому приходится иметь дело с его “заместителем” — экземпляром класса Thread, который и породил его. Чтобы создать новый поток исполнения, следует расширить класс Thread или же реализовать интерфейс Runnable.

В классе Thread определяется ряд методов, помогающих управлять потоками исполнения. Некоторые из тех методов, которые упоминаются в этой главе, перечислены в табл. 11.1.

Таблица 11.1. Методы управления потоками исполнения из класса Thread

Метод	Назначение
getName	Получает имя потока исполнения
getPriority	Получает приоритет потока исполнения
isAlive	Определяет, выполняется ли поток
join	Ожидает завершения потока исполнения
run	Задаёт точку входа в поток исполнения
sleep	Приостанавливает выполнение потока на заданное время
start	Запускает поток исполнения, вызывая его метода run()

Во всех упоминавшихся до сих пор примерах программ использовался единственный поток исполнения. А далее поясняется, как пользоваться классом Thread и интерфейсом Runnable для создания потоков исполнения и управления ими, начиная с главного потока, присутствующего в каждой программе на Java.

Главный поток исполнения

Когда программа на Java запускается на выполнение, сразу же начинает выполняться один поток. Он обычно называется *главным потоком* программы, потому что он запускается вместе с ней. Главный поток исполнения важен по двум причинам.

- От этого потока исполнения порождаются все дочерние потоки.
- Зачастую он должен быть последним потоком, завершающим выполнение программы, поскольку в нем производятся различные завершающие действия.

Несмотря на то что главный поток исполнения создается автоматически при запуске программы, им можно управлять через объект класса `Thread`. Для этого достаточно получить ссылку на него, вызвав метод `currentThread()`, который объявляется как открытый и статический (`public static`) в классе `Thread`. Его общая форма выглядит следующим образом:

```
static Thread currentThread()
```

Этот метод возвращает ссылку на тот поток исполнения, из которого он был вызван. Получив ссылку на главный поток, можно управлять им таким же образом, как и любым другим потоком исполнения. Рассмотрим следующий пример программы:

```
// Управление главным потоком исполнения
class CurrentThreadDemo {
    public static void main(String args[]) {
        Thread t = Thread.currentThread();

        System.out.println("Текущий поток исполнения: " + t);

        // изменить имя потока исполнения
        t.setName("My Thread");
        System.out.println("После изменения имени потока: " + t);

        try {
            for(int n = 5; n > 0; n--) {
                System.out.println(n);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Главный поток исполнения прерван");
        }
    }
}
```

В этом примере программы ссылка на текущий поток исполнения (в данном случае — главный поток) получается в результате вызова метода `currentThread()` и сохраняется в локальной переменной `t`. Затем выводятся сведения о потоке исполнения. Далее вызывается метод `setName()` для изменения внутреннего имени потока исполнения. После этого сведения о потоке исполнения выводятся заново. А в следующем далее цикле выводятся цифры в обратном порядке с задержкой на 1 секунду после каждой строки. Пауза организуется с помощью метода `sleep()`. Аргумент метода `sleep()` задает время задержки в миллисекундах. Обратите внимание на блок операторов `try/catch`, в котором находится цикл. Метод `sleep()` из класса `Thread` может сгенерировать исключение типа `InterruptedException`, если в каком-нибудь другом потоке исполнения потребуется прервать ожидающий

поток. В данном примере просто выводится сообщение, если поток исполнения прерывается, а в реальных программах подобную ситуацию придется обрабатывать иначе. Ниже приведен результат, выводимый данной программой.

```
Текущий поток исполнения: Thread[main,5,main]
После изменения имени потока: Thread[My Thread,5,main]
5
4
3
2
1
```

Обратите внимание на то, что вывод производится тогда, когда переменная `t` служит в качестве аргумента метода `println()`. Этот метод выводит по порядку имя потока исполнения, его приоритет и имя его группы. По умолчанию главный поток исполнения имеет имя `main` и приоритет, равный 5. Именем `main` обозначается также группа потоков исполнения, к которой относится данный поток. *Группа потоков исполнения* — это структура данных, которая управляет состоянием всего ряда потоков исполнения в целом. После изменения имени потока исполнения содержимое переменной `t` выводится снова — на этот раз новое имя потока исполнения.

Рассмотрим подробнее методы из класса `Thread`, используемые в приведенном выше примере программы. Метод `sleep()` вынуждает тот поток, из которого он вызывается, приостановить свое выполнение на указанное количество миллисекунд. Общая форма этого метода выглядит следующим образом:

```
static void sleep(long миллисекунд) throws InterruptedException
```

Количество миллисекунд, на которое нужно приостановить выполнение, задает аргумент *миллисекунд*. Метод `sleep()` может сгенерировать исключение типа `InterruptedException`. У него имеется и вторая, приведенная ниже форма, которая позволяет точнее задать время ожидания в милли- и наносекундах.

```
static void sleep(long миллисекунд, long наносекунд) throws
InterruptedException
```

Вторая форма данного метода может применяться только в тех средах, где предусматривается задание промежутков времени в наносекундах.

Как показано в предыдущем примере программы, установить имя потока исполнения можно с помощью метода `setName()`. А для того чтобы получить имя потока исполнения, достаточно вызвать метод `getName()`, хотя это в данном примере программы не показано. Оба эти метода являются членами класса `Thread` и объявляются так, как показано ниже, где *имя_потока* обозначает имя конкретного потока исполнения.

```
final void setName(String имя_потока)
final String getName()
```

Создание потока исполнения

В наиболее общем смысле для создания потока исполнения следует получить экземпляр объекта типа `Thread`. В языке Java этой цели можно достичь следующими двумя способами:

- реализовав интерфейс `Runnable`;
- расширив класс `Thread`.

В последующих разделах эти способы рассматриваются по очереди.

Реализация интерфейса `Runnable`

Самый простой способ создать поток исполнения состоит в том, чтобы объявить класс, реализующий интерфейс `Runnable`. Этот интерфейс предоставляет абстракцию единицы исполняемого кода. Поток исполнения можно создать из объекта любого класса, реализующего интерфейс `Runnable`. Для реализации интерфейса `Runnable` в классе должен быть объявлен единственный метод `run()`:

```
public void run()
```

В теле метода `run()` определяется код, который, собственно, и составляет новый поток исполнения. Однако в методе `run()` можно вызывать другие методы, использовать другие классы, объявлять переменные таким же образом, как и в главном потоке исполнения. Единственное отличие заключается в том, что в методе `run()` устанавливается точка входа в другой, параллельный поток исполнения в программе. Этот поток исполнения завершится, когда метод `run()` возвратит управление.

После создания класса, реализующего интерфейс `Runnable`, в этом классе следует получить экземпляр объекта типа `Thread`. Для этой цели в классе `Thread` определен ряд конструкторов. Тот конструктор, который должен использоваться в данном случае, выглядит в общей форме следующим образом:

```
Thread(Runnable объект_потока, String имя_потока)
```

В этом конструкторе параметр *объект_потока* обозначает экземпляр класса, реализующего интерфейс `Runnable`. Этим определяется место, где начинается выполнение потока. Имя нового потока исполнения передается данному конструктору в качестве параметра *имя_потока*.

После того как новый поток исполнения будет создан, он не запускается до тех пор, пока не будет вызван метод `start()`, объявленный в классе `Thread`. По существу, в методе `start()` вызывается метод `run()`. Ниже показано, каким образом объявляется метод `start()`.

```
void start()
```

Рассмотрим следующий пример программы, демонстрирующий создание и запуск нового потока на выполнение:

```
// Создать второй поток исполнения
class NewThread implements Runnable {
    Thread t;

    NewThread() {
        // создать новый, второй поток исполнения
        t = new Thread(this, "Демонстрационный поток");
        System.out.println("Дочерний поток создан: " + t);
        t.start(); // запустить поток исполнения
    }
}
```

```
// Точка входа во второй поток исполнения
public void run() {
    try {
        for(int i = 5; i > 0; i--) {
            System.out.println("Дочерний поток: " + i);
            Thread.sleep(500);
        }
    } catch (InterruptedException e) {
        System.out.println("Дочерний поток прерван.");
    }
    System.out.println("Дочерний поток завершен.");
}

class ThreadDemo {
    public static void main(String args[]) {
        new NewThread(); // создать новый поток

        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Главный поток:" + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Главный поток прерван.");
        }

        System.out.println("Главный поток завершен.");
    }
}
```

Новый объект класса Thread создается в следующем операторе из конструктора `NewThread()`:

```
t = new Thread(this, "Демонстрационный поток");
```

Передача ссылки `this` на текущий объект в первом аргументе данного конструктора означает следующее: в новом потоке исполнения для текущего объекта по ссылке `this` следует вызвать метод `run()`. Далее в приведенном выше примере программы вызывается метод `start()`, в результате чего поток исполнения запускается, начиная с метода `run()`. Это, в свою очередь, приводит к началу цикла `for` в дочернем потоке исполнения. После вызова метода `start()` конструктор `NewThread()` возвращает управление методу `main()`. Возобновляя свое исполнение, главный поток входит в свой цикл `for`. Далее потоки выполняются параллельно, совместно используя ресурсы процессора в одноядерной системе, вплоть до завершения своих циклов. Ниже приведен результат, выводимый данной программой (у вас он может оказаться иным в зависимости от конкретной исполняющей среды).

```
Дочерний поток: Thread[Демонстрационный поток,5,main]
Главный поток: 5
Дочерний поток: 5
Дочерний поток: 4
Главный поток: 4
Дочерний поток: 3
Дочерний поток: 2
Главный поток: 3
Дочерний поток: 1
Дочерний поток завершен.
```

Главный поток: 2
 Главный поток: 1
 Главный поток завершен.

Как упоминалось ранее, в многопоточной программе главный поток исполнения зачастую должен завершаться последним. На самом же деле, если главный поток исполнения завершается раньше дочерних потоков, то исполняющая система Java может “зависнуть”, что характерно для некоторых старых виртуальных машин JVM. В приведенном выше примере программы гарантируется, что главный поток исполнения завершится последним, поскольку главный поток исполнения находится в состоянии ожидания в течение 1000 миллисекунд в промежутках между последовательными шагами цикла, а дочерний поток исполнения — только 500 миллисекунд. Это заставляет дочерний поток исполнения завершиться раньше главного потока. Впрочем, далее будет показано, как лучше организовать ожидание завершения потоков исполнения.

Расширение класса Thread

Еще один способ создать поток исполнения состоит в том, чтобы сначала объявить класс, расширяющий класс Thread, а затем получить экземпляр этого класса. В расширяющем классе должен быть непременно переопределен метод run(), который является точкой входа в новый поток исполнения. Кроме того, в этом классе должен быть вызван метод start() для запуска нового потока на исполнение. Ниже приведена версия программы из предыдущего примера, переделанная с учетом расширения класса Thread.

```
// Создать второй поток исполнения, расширив класс Thread
class NewThread extends Thread {

    NewThread() {
        // создать новый поток исполнения
        super("Демонстрационный поток");
        System.out.println("Дочерний поток: " + this);
        start(); // запустить поток на исполнение
    }

    // Точка входа во второй поток исполнения
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Дочерний поток: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Дочерний поток прерван.");
        }
        System.out.println("Дочерний поток завершен.");
    }
}

class ExtendThread {
    public static void main(String args[]) {
        new NewThread(); // создать новый поток исполнения

        try {
```

```

        for(int i = 5; i > 0; i--) {
            System.out.println("Главный поток: " + i);
            Thread.sleep(1000);
        }
    } catch (InterruptedException e) {
        System.out.println("Главный поток прерван.");
    }

    System.out.println("Главный поток завершен.");
}
}

```

Эта версия программы выводит такой же результат, как и предыдущая ее версия. Как видите, дочерний поток исполнения создается при конструировании объекта класса `NewThread`, наследующего от класса `Thread`. Обратите внимание на метод `super()` в классе `NewThread`. Он вызывает конструктор `Thread()`, общая форма которого приведена ниже, где параметр *имя_потока* обозначает имя порождаемого потока исполнения.

```
public Thread(String имя_потока)
```

Выбор способа создания потоков исполнения

В связи с изложенным выше могут возникнуть следующие вопросы: почему в Java предоставляются два способа для создания порождаемых потоков исполнения и какой из них лучше? Ответы на эти вопросы взаимосвязаны. В классе `Thread` определяется ряд методов, которые могут быть переопределены в производных классах. И только один из них *должен* быть *непременно* переопределен: метод `run()`. Безусловно, этот метод требуется и в том случае, когда реализуется интерфейс `Runnable`. Многие программирующие на Java считают, что классы следует расширять только в том случае, если они должны быть усовершенствованы или каким-то образом видоизменены. Следовательно, если ни один из других методов не переопределяется в классе `Thread`, то лучше и проще реализовать интерфейс `Runnable`. Кроме того, при реализации интерфейса `Runnable` класс порождаемого потока исполнения не должен наследовать класс `Thread`, что освобождает его от наследования другого класса. В конечном счете выбор конкретного способа для создания потоков исполнения остается за вами. Тем не менее в примерах, приведенных далее в этой главе, потоки будут создаваться с помощью классов, реализующих интерфейс `Runnable`.

Создание многих потоков исполнения

В приведенных до сих пор примерах использовались только два потока исполнения: главный и дочерний. Но в прикладной программе можно порождать сколько угодно потоков исполнения. Например, в следующей программе создаются три дочерних потока исполнения:

```

// Создать несколько потоков исполнения
class NewThread implements Runnable {
    String name; // имя потока исполнения

```

```

Thread t;

NewThread(String threadname) {
    name = threadname;
    t = new Thread(this, name);
    System.out.println("Новый поток: " + t);
    t.start(); // запустить поток на исполнение
}

// Точка входа в поток исполнения
public void run() {
    try {
        for(int i = 5; i > 0; i--) {
            System.out.println(name + ": " + i);
            Thread.sleep(1000);
        }
    } catch (InterruptedException e) {
        System.out.println(name + " прерван");
    }
    System.out.println(name + " завершен.");
}

}

class MultiThreadDemo {
    public static void main(String args[]) {
        new NewThread("Один"); // запустить потоки на исполнение
        new NewThread("Два");
        new NewThread("Три");

        try {
            // ожидать завершения других потоков исполнения
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            System.out.println("Главный поток прерван");
        }

        System.out.println("Главный поток завершен.");
    }
}

```

Ниже приведен результат, выводимый данной программой (у вас он может оказаться иным в зависимости от конкретной исполняющей среды).

```

Новый поток: Thread[Один,5,main]
Новый поток: Thread[Два,5,main]
Новый поток: Thread[Три,5,main]
Один: 5
Два: 5
Три: 5
Один: 4
Два: 4
Три: 4
Один: 3
Три: 3
Два: 3
Один: 2
Три: 2
Два: 2
Один: 1
Три: 1
Два: 1

```

Один завершен.
Два завершен.
Три завершен.
Главный поток завершен.

Как видите, после запуска на исполнение все три дочерних потока совместно используют общие ресурсы ЦП. Обратите внимание на вызов метода `sleep(10000)` в методе `main()`. Это вынуждает главный поток перейти в состояние ожидания на 10 секунд и гарантирует его завершение последним.

Применение методов `isAlive()` и `join()`

Как упоминалось ранее, нередко требуется, чтобы главный поток исполнения завершался последним. С этой целью метод `sleep()` вызывался в предыдущих примерах из метода `main()` с достаточной задержкой, чтобы все дочерние потоки исполнения завершились раньше главного. Но это неудовлетворительное решение, вызывающее следующий серьезный вопрос: откуда одному потоку исполнения известно, что другой поток завершился? Правда, в классе `Thread` предоставляется средство, позволяющее разрешить этот вопрос.

Определить, был ли поток исполнения завершен, можно двумя способами. Во-первых, для этого потока можно вызвать метод `isAlive()`, определенный в классе `Thread`. Ниже приведена общая форма этого метода.

```
final Boolean isAlive()
```

Метод `isAlive()` возвращает логическое значение `true`, если поток, для которого он вызван, еще исполняется. В противном случае он возвращает логическое значение `false`.

И во-вторых, в классе `Thread` имеется метод `join()`, который применяется чаще, чем метод `isAlive()`, чтобы дождаться завершения потока исполнения. Ниже приведена общая форма этого метода.

```
final void join() throws InterruptedException
```

Этот метод ожидает завершения того потока исполнения, для которого он вызван. Его имя отражает следующий принцип: вызывающий поток ожидает, когда указанный поток *присоединится* к нему. Дополнительные формы метода `join()` позволяют указывать максимальный промежуток времени, в течение которого требуется ожидать завершения указанного потока исполнения.

Ниже приведена усовершенствованная версия программы из предыдущего примера, где с помощью метода `join()` гарантируется, что главный поток завершится последним. В данном примере демонстрируется также применение метода `isAlive()`.

```
// Применить метод join(), чтобы ожидать завершения потоков исполнения
class NewThread implements Runnable {
    String name; // имя потока исполнения
    Thread t;

    NewThread(String threadname) {
        name = threadname;
```



```

        t = new Thread(this, name);
        System.out.println("Новый поток: " + t);
        t.start(); // запустить поток исполнения
    }

    // Точка входа в поток исполнения
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println(name + " прерван.");
        }
        System.out.println(name + " завершен.");
    }
}

class DemoJoin {
    public static void main(String args[]) {
        NewThread obl = new NewThread("Один");
        NewThread ob2 = new NewThread("Два");
        NewThread ob3 = new NewThread("Три");

        System.out.println("Поток Один запущен: "
            + obl.t.isAlive());
        System.out.println("Поток Два запущен: "
            + ob2.t.isAlive());
        System.out.println("Поток Три запущен: "
            + ob3.t.isAlive());
        // ожидать завершения потоков исполнения
        try {
            System.out.println("Ожидание завершения потоков.");
            obl.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch (InterruptedException e) {
            System.out.println("Главный поток прерван");
        }

        System.out.println("Поток Один запущен: "
            + obl.t.isAlive());
        System.out.println("Поток Два запущен: "
            + ob2.t.isAlive());
        System.out.println("Поток Три запущен: "
            + ob3.t.isAlive());

        System.out.println("Главный поток завершен.");
    }
}

```

Ниже приведен результат, выводимый данной программой (у вас он может оказаться иным в зависимости от конкретной исполняющей среды).

```

Новый поток: Thread[Один,5,main]
Новый поток: Thread[Два,5,main]
Новый поток: Thread[Три,5,main]
Поток Один запущен: true
Поток Два запущен: true
Поток Три запущен: true

```

Ожидание завершения потоков.

Один: 5

Два: 5

Три: 5

Один: 4

Два: 4

Три: 4

Один: 3

Два: 3

Три: 3

Один: 2

Два: 2

Три: 2

Один: 1

Два: 1

Три: 1

Два завершен.

Три завершен.

Один завершен.

Поток Один запущен: false

Поток Два запущен: false

Поток Три запущен: false

Главный поток завершен.

Как видите, потоки прекращают исполнение после того, как вызовы метода `join()` возвращают управление.

Приоритеты потоков исполнения

Планировщик потоков использует приоритеты потоков исполнения, чтобы принять решение, когда разрешить исполнение каждому потоку. Теоретически высокоприоритетные потоки исполнения получают больше времени ЦП, чем низкоприоритетные. А на практике количество времени ЦП, которое получает поток исполнения, нередко зависит не только от его приоритета, но и от ряда других факторов. (Например, особенности реализации многозадачности в операционной системе могут оказывать влияние на относительную доступность времени ЦП.) Высокоприоритетный поток исполнения может также вытеснять низкоприоритетный. Например, когда низкоприоритетный поток исполняется, а высокоприоритетный собирается возобновить свое исполнение, прерванное в связи с приостановкой или ожиданием завершения операции ввода-вывода, то он вытесняет низкоприоритетный поток.

Теоретически потоки исполнения с одинаковым приоритетом должны получать равный доступ к ЦП. Но не следует забывать, что язык Java предназначен для применения в обширном ряде сред. В одних из этих сред многозадачность реализуется совершенно иначе, чем в других. В целях безопасности потоки исполнения с одинаковым приоритетом должны получать управление лишь время от времени. Этим гарантируется, что все потоки получают возможность выполняться в среде операционной системы с невытесняющей многозадачностью. Но на практике даже в средах с невытесняющей многозадачностью большинство потоков все-таки имеют шанс для исполнения, поскольку во всех потоках неизбежно возникают ситуации блокировки, например, в связи с ожиданием ввода-вывода. Когда случается нечто подоб-

ное, исполнение заблокированного потока приостанавливается, а остальные потоки могут исполняться. Но если требуется добиться плавной работы многопоточной программы, то полагаться на случай лучше не стоит. К тому же в некоторых видах задач весьма интенсивно используется ЦП. Потоки, исполняющие такие задачи, стремятся захватить ЦП, поэтому передавать им управление следует изредка, чтобы дать возможность выполняться другим потокам.

Чтобы установить приоритет потока исполнения, следует вызвать метод `setPriority()` из класса `Thread`. Его общая форма выглядит следующим образом:

```
final void setPriority(int уровень)
```

где аргумент *уровень* обозначает новый уровень приоритета для вызывающего потока исполнения. Значение аргумента *уровень* должно быть в пределах от `MIN_PRIORITY` до `MAX_PRIORITY`. В настоящее время эти значения равны соответственно 1 и 10. Чтобы вернуть потоку исполнения приоритет по умолчанию, следует указать значение `NORM_PRIORITY`, которое в настоящее время равно 5. Эти приоритеты определены в классе `Thread` как статические завершенные (`static final`) переменные. Для того чтобы получить текущее значение приоритета потока исполнения, достаточно вызвать метод `getPriority()` из класса `Thread`, как показано ниже.

```
final int getPriority()
```

Разные реализации Java могут вести себя совершенно иначе в отношении планирования потоков исполнения. Большинство несоответствий возникает при наличии потоков исполнения, опирающихся на вытесняющую многозадачность вместо совместного использования времени ЦП. Наиболее безопасный способ получить предсказуемое межплатформенное поведение многопоточных программ на Java состоит в том, чтобы использовать потоки исполнения, которые добровольно уступают управление ЦП.

Синхронизация

Когда два или более потока исполнения имеют доступ к одному совместно используемому ресурсу, они нуждаются в гарантии, что ресурс будет одновременно использован только одним потоком. Процесс, обеспечивающий такое поведение потоков исполнения, называется *синхронизацией*. Как будет показано далее, в Java предоставляется особая поддержка синхронизации на уровне языка.

Ключом к синхронизации является понятие монитора. *Монитор* — это объект, используемый в качестве *взаимоисключающей блокировки*. Только один поток исполнения может в одно и то же время *владеть* монитором. Когда поток исполнения запрашивает блокировку, то говорят, что он *входит* в монитор. Все другие потоки исполнения, пытающиеся войти в заблокированный монитор, будут приостановлены до тех пор, пока первый поток не *выйдет* из монитора. Обо всех прочих потоках говорят, что они *ожидают* монитор. Поток, владеющий монитором, может, если пожелает, повторно войти в него. Синхронизировать прикладной код можно двумя способами, предусматривающими использование ключевого слова `synchronized`. Оба эти способа будут рассмотрены далее по очереди.

Применение синхронизированных методов

Синхронизация достигается в Java просто, поскольку у объектов имеются свои, неявно связанные с ними мониторы. Чтобы войти в монитор объекта, достаточно вызвать метод, объявленный с модификатором доступа `synchronized`. Когда поток исполнения оказывается в теле синхронизированного метода, все другие потоки исполнения или любые другие синхронизированные методы, пытающиеся вызвать его для того же самого экземпляра, вынуждены ожидать. Чтобы выйти из монитора и передать управление объектом другому ожидающему потоку исполнения, владелец монитора просто возвращает управление из синхронизированного метода.

Чтобы стала понятнее потребность в синхронизации, рассмотрим сначала простой пример, в котором синхронизация отсутствует, хотя и должна быть осуществлена. Приведенная ниже программа состоит из трех простых классов. Первый из них, `Callme`, содержит единственный метод `call()`. Этот метод принимает параметр `msg` типа `String` и пытается вывести символьную строку `msg` в квадратных скобках. Любопытно отметить, что после того, как метод `call()` выводит открывающую квадратную скобку и символьную строку `msg`, он вызывает метод `Thread.sleep(1000)`, который приостанавливает текущий поток исполнения на одну секунду.

Конструктор следующего класса, `Caller`, принимает ссылку на экземпляры классов `Callme` и `String`, которые сохраняются в переменных `target` и `msg` соответственно. В этом конструкторе создается также новый поток исполнения, в котором вызывается метод `run()` для данного объекта. Этот поток запускается немедленно. В методе `run()` из класса `Caller` вызывается метод `call()` для экземпляра `target` класса `Callme`, передавая ему символьную строку `msg`. И наконец, класс `Synch` начинается с создания единственного экземпляра класса `Callme` и трех экземпляров класса `Caller` с отдельными строками сообщения. Один и тот же экземпляр класса `Callme` передается каждому конструктору `Caller()`.

```
// Эта программа не синхронизирована
class Callme {
    void call(String msg) {
        System.out.print "[" + msg);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("Прервано");
        }
        System.out.println("]");
    }
}

class Caller implements Runnable {
    String msg;
    Callme target;
    Thread t;

    public Caller(Callme targ, String s) {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
    }

    public void run() {
```

```

        target.call(msg);
    }
}

class Synch {
    public static void main(String args[]) {
        Callme target = new Callme();
        Caller ob1 = new Caller(target, "Добро пожаловать");
        Caller ob2 = new Caller(target, "в синхронизированный");
        Caller ob3 = new Caller(target, "мир!");

        // ожидать завершения потока исполнения
        try {
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch (InterruptedException e) {
            System.out.println("Прервано");
        }
    }
}

```

Ниже приведен результат, выводимый данной программой.

```

Добро пожаловать[в синхронизированный[мир!]]
]
]

```

Как видите, благодаря вызову метода `sleep()` из метода `call()` удается переключиться на исполнение другого потока. Это приводит к смешанному выводу трех строк сообщений. В данной программе отсутствует механизм, предотвращающий одновременный вызов в потоках исполнения одного и того же метода для того же самого объекта, или так называемое *состояние гонок*, поскольку три потока соперничают за окончание метода. В данном примере применяется метод `sleep()`, чтобы добиться повторяемости и наглядности получаемого эффекта. Но, как правило, состояние гонок менее заметно и предсказуемо, поскольку трудно предугадать, когда именно произойдет переключение контекста. В итоге программа может быть выполнена один раз правильно, а другой раз — неправильно.

Чтобы исправить главный недостаток данной программы, следует *упорядочить* доступ к методу `call()`. Это означает, что доступ к этому методу из потоков исполнения следует разрешить только по очереди. Для этого достаточно предварить объявление метода `call()` ключевым словом `synchronized`, как показано ниже.

```

class Callme {
    synchronized void call(String msg) {
        ...
    }
}

```

Этим предотвращается доступ к методу `call()` из других потоков исполнения, когда он уже используется в одном потоке. После ввода модификатора доступа `synchronized` в объявление метода `call()` результат выполнения данной программы будет выглядеть следующим образом:

```

[Добро пожаловать]
[в синхронизированный]
[мир!]

```

Всякий раз, когда имеется метод или группа методов, манипулирующих внутренним состоянием объекта в многопоточной среде, следует употребить ключевое слово `synchronized`, чтобы исключить состояние гонок. Напомним, что как только поток исполнения входит в любой синхронизированный метод экземпляра, ни один другой поток исполнения не сможет войти в какой-нибудь другой синхронизированный метод того же экземпляра. Тем не менее несинхронизированные методы экземпляра по-прежнему остаются доступными для вызова.

Оператор `synchronized`

Несмотря на всю простоту и эффективность синхронизации, которую обеспечивает создание синхронизированных методов, такой способ оказывается пригодным далеко не всегда. Чтобы стало понятнее, почему так происходит, рассмотрим следующую ситуацию. Допустим, требуется синхронизировать доступ к объектам класса, не предназначенного для многопоточного доступа. Это означает, что в данном классе не используются синхронизированные методы. Более того, класс написан сторонним разработчиком, и его исходный код недоступен, а следовательно, в объявление соответствующих методов данного класса нельзя ввести модификатор доступа `synchronized`. Как же синхронизировать доступ к объектам такого класса? К счастью, существует довольно простое решение этого вопроса: заключить вызовы методов такого класса в блок оператора `synchronized`. Ниже приведена общая форма оператора `synchronized`.

```
synchronized(ссылка_на_объект) {
    // синхронизируемые операторы
}
```

Здесь `ссылка_на_объект` обозначает ссылку на синхронизируемый объект. Блок оператора `synchronized` гарантирует, что вызов метода, являющегося членом того же класса, что и синхронизируемый объект, на который делается `ссылка_на_объект`, произойдет только тогда, когда текущий поток исполнения успешно войдет в монитор данного объекта.

Ниже приведена альтернативная версия программы из предыдущего примера, где в теле метода `run()` используется синхронизированный блок.

```
// В этой программе используется синхронизированный блок
class Callme {
    void call(String msg) {
        System.out.print "[" + msg);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("Прервано");
        }
        System.out.println("]");
    }
}

class Caller implements Runnable {
    String msg;
    Callme target;
    Thread t;
```

```

public Caller(Callme targ, String s) {
    target = targ;
    msg = s;
    t = new Thread(this);
    t.start();
}

// синхронизированные вызовы метода call()
public void run() {
    synchronized(target) { // синхронизированный блок
        target.call(msg);
    }
}
}

class Synchl {
    public static void main(String args[]) {
        Callme target = new Callme();
        Caller ob1 = new Caller(target, "Добро пожаловать");
        Caller ob2 = new Caller(target, "в синхронизированный");
        Caller ob3 = new Caller(target, "мир!");

        // ожидать завершения потока исполнения
        try {
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch (InterruptedException e) {
            System.out.println("Прервано");
        }
    }
}

```

В данном примере метод `call()` объявлен без модификатора доступа `synchronized`. Вместо этого используется оператор `synchronized` в теле метода `run()` из класса `Caller`. Благодаря этому получается тот же правильный результат, что и в предыдущем примере, поскольку каждый поток исполнения ожидает завершения предыдущего потока.

Взаимодействие потоков исполнения

В предыдущих примерах другие потоки исполнения, безусловно, блокировались от асинхронного доступа к некоторым методам. Такое применение неявных мониторов объектов в Java оказывается довольно эффективным, но более точного управления можно добиться, организовав взаимодействие потоков исполнения. Как будет показано ниже, добиться такого взаимодействия особенно просто в Java.

Как обсуждалось ранее, многопоточность заменяет программирование циклов ожидания событий благодаря разделению задач на дискретные, логически обособленные единицы. Еще одно преимущество предоставляют потоки исполнения, исключая опрос. Как правило, опрос реализуется в виде цикла, организуемого для периодической проверки некоторого условия. Как только условие оказывается истинным, выполняется определенное действие. Но на это расходуется время ЦП. Рассмотрим в качестве примера классическую задачу организации очереди, когда некоторые дан-

ные поставляются в одном потоке исполнения, а в другом потоке они потребляются. Чтобы сделать эту задачу более интересной, допустим, что поставщик данных должен ожидать завершения работы потребителя, прежде чем сформировать новые данные. В системах с опросом потребитель данных тратит немало циклов ЦП на ожидание данных от поставщика. Как только поставщик завершит работу, он должен начать опрос, напрасно расходуя лишние циклы ЦП в ожидании завершения работы потребителя данных, и т.д. Ясно, что такая ситуация нежелательна.

Чтобы избежать опроса, в Java внедрен изящный механизм взаимодействия потоков исполнения с помощью методов `wait()`, `notify()` и `notifyAll()`. Эти методы реализованы как завершенные в классе `Object`, поэтому они доступны всем классам. Все три метода могут быть вызваны только из синхронизированного контекста. Правила применения этих методов достаточно просты, хотя с точки зрения вычислительной техники они принципиально прогрессивны. Эти правила состоят в следующем.

- Метод `wait()` вынуждает вызывающий поток исполнения уступить монитор и перейти в состояние ожидания до тех пор, пока какой-нибудь другой поток исполнения не войдет в тот же монитор и не вызовет метод `notify()`.
- Метод `notify()` возобновляет исполнение потока, из которого был вызван метод `wait()` для того же самого объекта.
- Метод `notifyAll()` возобновляет исполнение всех потоков, из которых был вызван метод `wait()` для того же самого объекта. Одному из этих потоков предоставляется доступ.

Все эти методы объявлены в классе `Object`, как показано ниже. Существуют дополнительные формы метода `wait()`, позволяющие указать время ожидания.

```
final void wait() throws InterruptedException
final void notify()
final void notifyAll()
```

Прежде чем рассматривать пример, демонстрирующий взаимодействие потоков исполнения, необходимо сделать одно важное замечание. Метод `wait()` обычно ожидает до тех пор, пока не будет вызван метод `notify()` или `notifyAll()`. Но вполне вероятно, хотя и в очень редких случаях, что ожидающий поток исполнения может быть возобновлен вследствие *ложной активизации*. При этом исполнение ожидающего потока возобновляется без вызова метода `notify()` или `notifyAll()`. (По существу, исполнение потока возобновляется без очевидных причин.) Из-за этой маловероятной возможности в компании Oracle рекомендуют вызывать метод `wait()` в цикле, проверяющем условие, по которому поток ожидает возобновления. И такой подход демонстрируется в представленном далее примере программы.

А до тех пор рассмотрим несложный пример программы, неправильно реализующей простую форму поставщика и потребителя данных. Эта программа состоит из четырех классов: `Q` — синхронизируемой очереди; `Producer` — поточного объекта, создающего элементы очереди; `Consumer` — поточного объекта, принимающего элементы очереди; а также `PC` — мелкого класса, в котором создаются объекты классов `Q`, `Producer` и `Consumer`. Ниже приведен исходный код этой программы.


```
// Неправильная реализация поставщика и потребителя
class Q {
    int n;

    synchronized int get() {
        System.out.println("Получено: " + n);
        return n;
    }

    synchronized void put(int n) {
        this.n = n;
        System.out.println("Отправлено: " + n);
    }
}

class Producer implements Runnable {
    Q q;

    Producer(Q q) {
        this.q = q;
        new Thread(this, "Поставщик").start();
    }

    public void run() {
        int i = 0;

        while(true) {
            q.put(i++);
        }
    }
}

class Consumer implements Runnable {
    Q q;

    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Потребитель").start();
    }

    public void run() {
        while(true) {
            q.get();
        }
    }
}

class PC {
    public static void main(String args[]) {
        Q q = new Q();
        new Producer(q);
        new Consumer(q);

        System.out.println("Для остановки нажмите Ctrl-C.");
    }
}
```

Несмотря на то что методы `put()` и `get()` синхронизированы в классе `Q`, ничто не остановит переполнение потребителя данными от поставщика, как и ничто не мешает потребителю дважды извлечь один и тот же элемент из очереди. В итоге будет выведен неверный результат, как показано ниже (конкретный результат может быть иным в зависимости от быстродействия и загрузки ЦП).

```
Отправлено: 1
Получено: 1
Получено: 1
Получено: 1
Получено: 1
Получено: 1
Отправлено: 2
Отправлено: 3
Отправлено: 4
Отправлено: 5
Отправлено: 6
Отправлено: 7
Получено: 7
```

Как видите, после того, как поставщик отправит значение 1, запускается потребитель, который получает это значение пять раз подряд. Затем поставщик продолжает свою работу, поставляя значения от 2 до 7, не давая возможности потребителю получить их. Для того чтобы правильно реализовать взаимодействие поставщика и потребителя в рассматриваемом здесь примере программы на Java, следует применить методы `wait()` и `notify()` для передачи уведомлений в обоих направлениях:

```
// Правильная реализация поставщика и потребителя
class Q {
    int n;
    boolean valueSet = false;

    synchronized int get() {
        while(!valueSet)
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println(
                    "Исключение типа InterruptedException перехвачено");
            }

        System.out.println("Получено: " + n);
        valueSet = false;
        notify();
        return n;
    }

    synchronized void put(int n) {
        while(valueSet)
            try {
                wait();
            } catch (InterruptedException e) {
                System.out.println(
                    "Исключение типа InterruptedException перехвачено");
            }

        this.n = n;
        valueSet = true;
        System.out.println("Отправлено: " + n);
        notify();
    }
}

class Producer implements Runnable {
    Q q;

    Producer(Q q) {
        this.q = q;
    }
}
```

```

        new Thread(this, "Поставщик").start();
    }

    public void run() {
        int i = 0;

        while(true) {
            q.put(i++);
        }
    }
}

class Consumer implements Runnable {
    Q q;

    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Потребитель").start();
    }

    public void run() {
        while(true) {
            q.get();
        }
    }
}

class PCFixed {
    public static void main(String args[]) {
        Q q = new Q();
        new Producer(q);
        new Consumer(q);

        System.out.println("Для остановки нажмите Ctrl-C.");
    }
}

```

В методе `get()` вызывается метод `wait()`. В итоге исполнение потока приостанавливается до тех пор, пока объект класса `Producer` не уведомит, что данные прочитаны. Когда это произойдет, исполнение потока в методе `get()` возобновится. Как только данные будут получены, в методе `get()` вызывается метод `notify()`. Этим объект класса `Producer` уведомляется о том, что все в порядке и в очереди можно разместить следующий элемент данных. Метод `wait()` приостанавливает исполнение потока в методе `put()` до тех пор, пока объект класса `Consumer` не извлечет элемент из очереди. Когда исполнение потока возобновится, следующий элемент данных размещается в очереди и вызывается метод `notify()`. Этим объект класса `Consumer` уведомляется, что он теперь может извлечь элемент из очереди.

Ниже приведен результат, выводимый данной программой. Он наглядно показывает, что теперь синхронизация потоков исполнения действует правильно.

```

Отправлено: 1
Получено: 1
Отправлено: 2
Получено: 2
Отправлено: 3
Получено: 3
Отправлено: 4
Получено: 4
Отправлено: 5
Получено: 5

```

Взаимная блокировка

Следует избегать особого типа ошибок, имеющего отношение к многозадачности и называемого *взаимной блокировкой*, которая происходит в том случае, когда потоки исполнения имеют циклическую зависимость от пары синхронизированных объектов. Допустим, один поток исполнения входит в монитор объекта X, а другой — в монитор объекта Y. Если поток исполнения в объекте X попытается вызвать любой синхронизированный метод для объекта Y, он будет заблокирован, как и предполагалось. Но если поток исполнения в объекте Y, в свою очередь, попытается вызвать любой синхронизированный метод для объекта X, то этот поток будет ожидать вечно, поскольку для получения доступа к объекту X он должен снять свою блокировку с объекта Y, чтобы первый поток исполнения мог завершиться. Взаимная блокировка является ошибкой, которую трудно отладить, по двум следующим причинам.

- В общем, взаимная блокировка возникает очень редко, когда исполнение двух потоков точно совпадает по времени.
- Взаимная блокировка может возникнуть, когда в ней участвует больше двух потоков исполнения и двух синхронизированных объектов. (Это означает, что взаимная блокировка может произойти в результате более сложной последовательности событий, чем в упомянутой выше ситуации.)

Чтобы полностью разобраться в этом явлении, его лучше рассмотреть в действии. В приведенном ниже примере программы создаются два класса, A и B, с методами `foo()` и `bar()` соответственно, которые приостанавливаются непосредственно перед попыткой вызова метода из другого класса. Сначала в главном классе `Deadlock` получают экземпляры классов A и B, а затем запускается второй поток исполнения, в котором устанавливается состояние взаимной блокировки. В методах `foo()` и `bar()` используется метод `sleep()`, чтобы стимулировать появление взаимной блокировки.

// Пример взаимной блокировки

```
class A {
    synchronized void foo(B b) {
        String name = Thread.currentThread().getName();

        System.out.println(name + " вошел в метод A.foo()");

        try {
            Thread.sleep(1000);
        } catch (Exception e) {
            System.out.println("Класс A прерван");
        }
        System.out.println(name + " пытается вызвать метод B.last()");
        b.last();
    }

    synchronized void last() {
        System.out.println("В методе A.last()");
    }
}

class B {
    synchronized void bar(A a) {
```

```

String name = Thread.currentThread().getName();
System.out.println(name + " вошел в метод B.bar()");

try {
    Thread.sleep(1000);
} catch (Exception e) {
    System.out.println("Класс B прерван");
}

System.out.println(name + " пытается вызвать метод A.last()");
a.last();
}

synchronized void last() {
    System.out.println("В методе A.last()");
}
}

class Deadlock implements Runnable {
    A a = new A();
    B b = new B();

    Deadlock() {
        Thread.currentThread().setName("Главный поток");
        Thread t = new Thread(this, "Соперничающий поток");
        t.start();

        a.foo(b); // получить блокировку для объекта a
                  // в этом потоке исполнения
        System.out.println("Назад в главный поток");
    }

    public void run() {
        b.bar(a); // получить блокировку для объекта b
                  // в другом потоке исполнения
        System.out.println("Назад в другой поток");
    }

    public static void main(String args[]) {
        new Deadlock();
    }
}

```

Запустив эту программу на выполнение, вы получите следующий результат:

```

Главный поток вошел в метод A.foo()
Соперничающий поток вошел в метод B.bar()
Главный поток пытается вызвать метод B.last()
Соперничающий поток пытается вызвать метод A.last()

```

В связи со взаимной блокировкой придется нажать комбинацию клавиш <Ctrl+C>, чтобы завершить данную программу. Нажав комбинацию клавиш <Ctrl+Pause> на ПК, можно увидеть весь дамп (вывод из оперативной памяти) потока и кеша монитора. В частности, Соперничающий поток владеет монитором объекта b, тогда как он ожидает монитор объекта a. В то же время Главный поток владеет объектом a и ожидает получить объект b. Следовательно, программа никогда не завершится. Как демонстрирует данный пример, если многопоточная программа неожиданно зависла, то прежде всего следует проверить возможность взаимной блокировки.

Приостановка, возобновление и остановка потоков исполнения

Иногда возникает потребность в приостановке исполнения потоков. Например, отдельный поток исполнения может служить для отображения времени дня. Если пользователю не требуется отображение текущего времени, этот поток исполнения можно приостановить. Но в любом случае приостановить исполнение потока совсем не трудно. Выполнение приостановленного потока может быть легко возобновлено.

Механизм временной или окончательной остановки потока исполнения, а также его возобновления отличался в ранних версиях Java, например, Java 1.0, от современных версий, начиная с Java 2. До версии Java 2 методы `suspend()` и `resume()`, определенные в классе `Thread`, использовались в программах для приостановки и возобновления потоков исполнения. На первый взгляд применение этих методов кажется вполне благоразумным и удобным подходом к управлению выполнением потоков. Тем не менее пользоваться ими в новых программах на Java не рекомендуется по следующей причине: метод `suspend()` из класса `Thread` несколько лет назад был объявлен не рекомендованным к употреблению, начиная с версии Java 2. Это было сделано потому, что иногда он способен порождать серьезные системные сбои. Допустим, что поток исполнения получил блокировки для очень важных структур данных. Если в этот момент приостановить исполнение данного потока, блокировки не будут сняты. Другие потоки исполнения, ожидающие эти ресурсы, могут оказаться взаимно заблокированными.

Метод `resume()` также не рекомендован к употреблению. И хотя его применение не вызовет особых осложнений, тем не менее им нельзя пользоваться без метода `suspend()`, который его дополняет.

Метод `stop()` из класса `Thread` также объявлен устаревшим с версии Java 2. Это было сделано потому, что он может иногда послужить причиной серьезных системных сбоев. Допустим, поток выполняет запись в критически важную структуру данных и успел произвести лишь частичное ее обновление. Если его остановить в этот момент, структура данных может оказаться в поврежденном состоянии. Дело в том, что метод `stop()` вызывает снятие любой блокировки, устанавливаемой вызывающим потоком исполнения. Следовательно, поврежденные данные могут быть использованы в другом потоке исполнения, ожидающем по той же самой блокировке.

Если методы `suspend()`, `resume()` или `stop()` нельзя использовать для управления потоками исполнения, то можно прийти к выводу, что теперь вообще нет никакого механизма для приостановки, возобновления или прерывания потока исполнения. К счастью, это не так. Вместо этого код управления выполнением потока должен быть составлен таким образом, чтобы метод `run()` периодически проверял, должно ли исполнение потока быть приостановлено, возобновлено или прервано. Обычно для этой цели служит флаговая переменная, обозначающая состояние потока исполнения. До тех пор, пока эта флаговая переменная содержит признак “выполняется”, метод `run()` должен продолжать выполнение. Если же эта переменная содержит признак “приостановить”, поток исполнения должен быть приостановлен. А если флаговая переменная получает признак “оста-

новить", то поток исполнения должен завершиться. Безусловно, имеются самые разные способы написать кода управления выполнением потока, но основной принцип остается неизменным для всех программ.

В приведенном ниже примере программы демонстрируется применение методов `wait()` и `notify()`, унаследованных из класса `Object`, для управления выполнением потока. Рассмотрим подробнее работу этой программы. Класс `NewThread` содержит переменную экземпляра `suspendFlag` типа `boolean`, используемую для управления выполнением потока. В конструкторе этого класса она инициализируется логическим значением `false`. Метод `run()` содержит блок оператора `synchronized`, где проверяется состояние переменной `suspendFlag`. Если она принимает логическое значение `true`, то вызывается метод `wait()` для приостановки выполнения потока. В методе `mysuspend()` устанавливается логическое значение `true` переменной `suspendFlag`, а в методе `myresume()` — логическое значение `false` этой переменной и вызывается метод `notify()`, чтобы активизировать поток исполнения. И наконец, в методе `main()` вызываются оба метода — `mysuspend()` и `myresume()`.

// Приостановка и возобновление исполнения потока современным способом

```
class NewThread implements Runnable {
    String name; // имя потока исполнения
    Thread t;
    boolean suspendFlag;

    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("Новый поток: " + t);
        suspendFlag = false;
        t.start(); // запустить поток исполнения
    }

    // Точка входа в поток исполнения
    public void run() {
        try {
            for(int i = 15; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(200);
                synchronized(this) {
                    while(suspendFlag) {
                        wait();
                    }
                }
            }
        } catch (InterruptedException e) {
            System.out.println(name + " прерван.");
        }

        System.out.println(name + " завершен.");
    }

    synchronized void mysuspend() {
        suspendFlag = true;
    }

    synchronized void myresume() {
        suspendFlag = false;
        notify();
    }
}
```

```
class SuspendResume {
    public static void main(String args[]) {
        NewThread obl = new NewThread("Один");
        NewThread ob2 = new NewThread("Два");

        try {
            Thread.sleep(1000);
            obl.mysuspend();
            System.out.println("Приостановка потока Один");
            Thread.sleep(1000);
            obl.myresume();
            System.out.println("Возобновление потока Один");
            ob2.mysuspend();
            System.out.println("Приостановка потока Два");
            Thread.sleep(1000);
            ob2.myresume();
            System.out.println("Возобновление потока Два");
        } catch (InterruptedException e) {
            System.out.println("Главный поток прерван");
        }

        // ожидать завершения потоков исполнения
        try {
            System.out.println("Ожидание завершения потоков.");
            obl.t.join();
            ob2.t.join();
        } catch (InterruptedException e) {
            System.out.println("Главный поток прерван");
        }

        System.out.println("Главный поток завершен");
    }
}
```

Если запустить эту программу на выполнение, то можно увидеть, как исполнение потоков приостанавливается и возобновляется. Далее в этой книге будут представлены другие примеры, в которых применяется современный механизм управления исполнением потоков. И хотя этот механизм не так прост, как прежний, его следует все же придерживаться, чтобы избежать ошибок во время выполнения. Именно такой механизм *должен* применяться во всяком новом коде.

Получение состояния потока исполнения

Как упоминалось ранее в этой главе, поток исполнения может находиться в нескольких состояниях. Для того чтобы получить текущее состояние потока исполнения, достаточно вызвать метод `getState()`, определенный в классе `Thread`, следующим образом:

```
Thread.State getState()
```

Этот метод возвращает значение типа `Thread.State`, обозначающее состояние потока исполнения на момент вызова. Перечисление `State` определено в классе `Thread`. (Перечисление представляет собой список именованных констант и подробно обсуждается в главе 12.) Значения, которые может вернуть метод `getState()`, перечислены в табл. 11.2.

Таблица 11.2. Значения, возвращаемые методом `getState()`

Значение	Состояние
BLOCKED	Поток приостановил выполнение, поскольку ожидает получения блокировки
NEW	Поток еще не начал выполнение
RUNNABLE	Поток в настоящее время выполняется или начнет выполняться, когда получит доступ к ЦП
TERMINATED	Поток завершил выполнение
TIMED_WAITING	Поток приостановил выполнение на определенный промежуток времени, например, после вызова метода <code>sleep()</code> . Поток переходит в это состояние и при вызове метода <code>wait()</code> или <code>join()</code>
WAITING	Поток приостановил выполнение, поскольку он ожидает некоторого действия, например, вызова версии метода <code>wait()</code> или <code>join()</code> без заданного времени ожидания

На рис. 11.1 схематически показана взаимосвязь различных состояний потока исполнения.

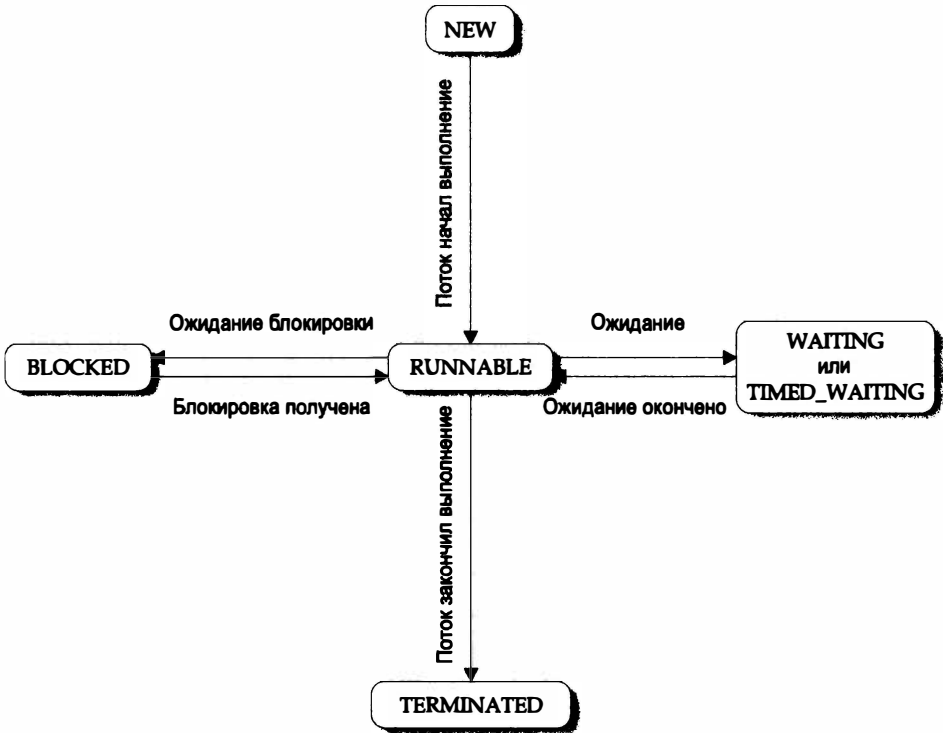


Рис. 11.1. Состояния потока исполнения

Имея в своем распоряжении экземпляр класса `Thread`, можно вызвать метод `getState()`, чтобы получить состояние потока исполнения. Например, в следующем фрагменте кода определяется, находится ли поток исполнения `thrd` в состоянии `RUNNABLE` во время вызова метода `getState()`:

```
Thread.State ts = thrd.getState();  
if(ts == Thread.State.RUNNABLE) // ...
```

Следует, однако, иметь в виду, что состояние потока исполнения может измениться после вызова метода `getState()`. Поэтому в зависимости от обстоятельств состояние, полученное при вызове метода `getState()`, мгновение спустя может уже не отражать фактическое состояние потока исполнения. По этой и другим причинам метод `getState()` не предназначен для синхронизации потоков исполнения. Он служит прежде всего для отладки или профилирования характеристик потока во время выполнения.

Применение многопоточности

Чтобы эффективно пользоваться многопоточными средствами в Java, нужно научиться мыслить категориями параллельного, а не последовательного выполнения операций. Так, если в программе имеются две подсистемы, которые могут выполняться одновременно, оформите их в виде отдельных потоков исполнения. Благоразумно применяя многопоточность, можно научиться писать довольно эффективные программы. Но не следует забывать, что, создав слишком много потоков исполнения, можно снизить производительность программы в целом, вместо того чтобы повысить ее. Следует также иметь в виду, что переключение контекста с одного потока на другой требует определенных издержек. Если создать очень много потоков исполнения, то на переключение контекста будет затрачено больше времени процессора, чем на выполнение самой программы! И последнее замечание: для создания прикладной программы, предназначенной для интенсивных вычислений и допускающей автоматическое масштабирование с целью задействовать имеющиеся процессоры в многоядерной системе, рекомендуется воспользоваться каркасом `Fork/Join Framework`, описанным в главе 28.

Перечисления, автоупаковка и аннотации (метаданные)

В этой главе рассматриваются три относительно новых дополнения языка Java: перечисления, автоупаковка и аннотации (называемые также метаданными). Каждое из них увеличивает эффективность этого языка, предлагая изящный подход к решению часто возникающих задач программирования. В этой главе обсуждаются также оболочки типов данных в Java и рефлексия.

Перечисления

До версии JDK 5 явно недоставало перечислений — одного из языковых средств, потребность в котором остро ощущали многие программирующие на Java. В простейшей форме *перечисление* представляет собой список именованных констант. Несмотря на то что в Java имеются другие языковые средства с похожими функциональными возможностями, например завершенные переменные, многим программирующим на Java все же не хватало принципиальной ясности перечислений, особенно потому, что они применяются во многих других языках программирования. В версии JDK 5 перечисления были внедрены в Java и наконец-то стали доступны для программирования на этом языке.

В простейшей форме перечисления в Java похожи на перечисления в других языках программирования, но это поверхностное сходство. В языках вроде C++ перечисления просто являются списками целочисленных констант. А в Java перечисления определяют тип класса. Благодаря тому что перечисления реализованы в виде классов, само понятие перечисления значительно расширяется. Например, перечисления в Java могут иметь конструкторы, методы и переменные экземпляра. И несмотря на то что внедрения перечислений пришлось ждать несколько лет, реализация их в Java того стоила.

Основные положения о перечислениях

Перечисления создаются с помощью ключевого слова `enum`. В качестве примера ниже показано простое перечисление сортов яблок.

```
// Перечисление сортов яблок
enum Apple {
    Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
```

Идентификаторы Jonathan, GoldenDel и так далее называются *константами перечислимого типа*. Каждая из них явно объявлена как открытый статический конечный член класса Apple. Более того, они относятся к типу того перечисления, в котором объявлены (в данном случае — Apple). В языке Java такие константы называются *самотипизированными*, причем префикс *само* относится к охватываемому их перечислению.

Объявив перечисление, можно создавать переменные данного типа. Но, несмотря на то, что перечисления определяют тип класса, получать экземпляры класса типа enum с помощью оператора new нельзя. Вместо этого переменная перечисления объявляется и применяется практически так же, как и переменные примитивных типов. В приведенном ниже примере объявляется переменная ar перечислимого типа Apple.

```
Apple ar;
```

Переменная ar относится к типу Apple, и поэтому ей можно присвоить только те значения, которые определены в перечислении. В следующем примере переменной ar присваивается значение RedDel:

```
ar = Apple.RedDel;
```

Обратите внимание на то, что значению RedDel предшествует тип Apple. Две константы перечислимого типа можно проверять на равенство с помощью операции отношения ==. Например, в следующем условном операторе переменная ar сравнивается с константой Apple.GoldenDel:

```
if (ar == Apple.GoldenDel) // ...
```

Значения перечислимого типа можно также использовать в управляющем операторе switch. Разумеется, для этого во всех выражениях ветвей case должны использоваться константы из того же самого перечисления, что и в самом операторе switch. Например, следующий оператор switch составлен совершенно правильно:

```
// Использовать перечисление для управления оператором switch
switch(ar) {
case Jonathan:
    // ...
case Winesap:
    // ...
```

Обратите внимание на то, что в выражениях ветвей case имена констант указываются без уточнения имени их перечислимого типа, например Winesap, а не Apple.Winesap. Дело в том, что тип перечисления в операторе switch уже неявно задает тип перечисления для выражений в ветвях case. Следовательно, уточнять имена констант в выражениях ветвей case не нужно. В действительности любая попытка сделать это приведет к ошибке во время компиляции.

Когда выводится константа перечислимого типа, например, методом println(), то отображается ее имя. Например, в следующей строке кода:

```
System.out.println(Apple.Winesapp)
```

выводится имя Winesapp.

Приведенный ниже пример программы подытоживает все сказанное выше, демонстрируя применение перечисления Apple.

```
// Перечисление сортов яблок
enum Apple {
    Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo {
    public static void main(String args[])
    {
        Apple ap;

        ap = Apple.RedDel;

        // вывести значение перечислимого типа
        System.out.println("Значение ap: " + ap);
        System.out.println();

        ap = Apple.GoldenDel;

        // сравнить два значения перечислимого типа
        if(ap == Apple.GoldenDel)
            System.out.println("Переменная ap содержит GoldenDel.\n");

        // применить перечисление для управления оператором switch
        switch(ap) {
            case Jonathan:
                System.out.println("Сорт Jonathan красный.");
                break;
            case GoldenDel:
                System.out.println("Сорт Golden Delicious желтый.");
                break;
            case RedDel:
                System.out.println("Сорт Red Delicious красный.");
                break;
            case Winesap:
                System.out.println("Сорт Winesap красный.");
                break;
            case Cortland:
                System.out.println("Сорт Cortland красный.");
                break;
        }
    }
}
```

Эта программа выводит следующий результат:

Значение ap: RedDel

Переменная ap содержит GoldenDel.

Сорт Golden Delicious желтый.

Методы values () и valueOf ()

Перечисления автоматически включают в себя два predefined метода: values () и valueOf (). Ниже приведена их общая форма.

```
public static тип_перечисления [] values()
public static тип_перечисления valueOf(String строка)
```

Метод `values()` возвращает массив, содержащий список констант перечислимого типа. А метод `valueOf()` возвращает константу перечислимого типа, значение которой соответствует символьной строке, переданной в качестве аргумента строка. В обоих случаях *тип_перечисления* обозначает тип конкретного перечисления. Так, если вызвать метод `Apple.valueOf("Winesap")` для упоминавшегося ранее перечисления `Apple`, то он возвратит значение константы перечислимого типа `Winesap`.

В следующей программе демонстрируется применение методов `values()` и `valueOf()`:

```
// Воспользоваться встроенными в перечисление методами

// Перечисление сортов яблок
enum Apple {
    Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo2 {
    public static void main(String args[])
    {
        Apple ap;

        System.out.println("Константы перечислимого типа Apple:");

        // применить метод values()
        Apple allapples[] = Apple.values();
        for(Apple a : allapples)
            System.out.println(a);

        System.out.println();

        // применить метод valueOf()
        ap = Apple.valueOf("Winesap");
        System.out.println("Переменная ap содержит " + ap);
    }
}
```

Эта программа выводит следующий результат:

```
Константы перечислимого типа Apple:
Jonathan
GoldenDel
RedDel
Winesap
Cortland
```

```
Переменная ap содержит Winesap
```

Обратите внимание на то, что для перебора массива констант, возвращаемых методом `values()`, в данной программе используется цикл `for` в стиле `for each`. В целях демонстрации создается переменная `allapples`, которой присваивается ссылка на массив значений перечислимого типа. Но это совсем не обязательно, поскольку цикл `for` можно написать и без переменной `allapples` следующим образом:

```
for(Apple a : Apple.values())  
    System.out.println(a);
```

Обратите также внимание на получение значения, соответствующего имени Winesap, в результате вызова метода `valueOf()`:

```
ap = Apple.valueOf("Winesap");
```

Как пояснялось ранее, метод `valueOf()` возвращает значение перечислимого типа, связанное с именем константы того же типа, передаваемым этому методу в виде символьной строки.

Перечисления в Java относятся к типам классов

Как пояснялось ранее, перечисление в Java относится к типу класса. Создать экземпляр перечисления с помощью оператора `new` нельзя, но в остальном перечисление обладает всеми возможностями, которые имеются у других классов. Тот факт, что перечисление определяет класс, наделяет перечисления в Java огромным потенциалом, которого лишены перечисления в других языках программирования. В частности, перечисления допускают предоставление конструкторов, добавление переменных экземпляров и методов и даже реализацию интерфейсов.

Важно понимать, что каждая константа перечислимого типа является объектом класса своего перечисления. Так, если для перечисления определяется конструктор, он вызывается всякий раз, когда создается константа перечислимого типа. Кроме того, у каждой константы перечислимого типа имеется своя копия любой из переменных экземпляра, объявленных в перечислении. Рассмотрим в качестве примера следующую версию перечисления `Apple`:

// Использовать конструктор, переменную экземпляра и метод в перечислении

```
enum Apple {  
    Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);  
  
    private int price; // цена яблока каждого сорта  
  
    // Конструктор  
    Apple(int p) { price = p; }  
  
    int getPrice() { return price; }  
}  
  
class EnumDemo3 {  
    public static void main(String args[])  
    {  
        Apple ap;  
  
        // вывести цену на яблоко сорта Winesap  
        System.out.println("Яблоко сорта Winesap стоит " +  
            Apple.Winesap.getPrice() +  
            " центов.\n");  
  
        // вывести цены на все сорта яблок  
        System.out.println("Цены на все сорта яблок:");  
        for(Apple a : Apple.values())  
            System.out.println(a + " стоит " + a.getPrice() +  
                " центов.");  
    }  
}
```


Ниже приведен результат, выводимый при выполнении данного кода.

```
Яблоко сорта Winesap стоит 15 центов.  
Цены на все сорта яблок:  
Jonathan стоит 10 центов.  
GoldenDel стоит 9 центов.  
RedDel стоит 12 центов.  
Winesap стоит 15 центов.  
Cortland стоит 8 центов.
```

В данной версии перечисления `Apple` добавлено следующее. Во-первых, переменная экземпляра `price`, которая служит для хранения цены яблока каждого сорта. Во-вторых, конструктор `Apple()`, которому передается цена на яблоко. И в-третьих, метод `getPrice()`, возвращающий значение цены.

Когда в методе `main()` объявляется переменная `ap`, конструктор `Apple()` вызывается один раз для каждой объявленной константы. Обратите внимание на то, что аргументы конструктору передаются в скобках после каждой перечисляемой константы, как показано ниже.

```
Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);
```

Эти значения передаются параметру `p` конструктора `Apple()`, который затем присваивает их переменной экземпляра `price`. Опять же конструктор вызывается один раз для каждой константы перечислимого типа.

У каждой константы перечислимого типа имеется своя копия переменной экземпляра `price`, поэтому для получения цены на определенный сорт яблок достаточно вызвать метод `getPrice()`. Например, цена на сорт яблок `Winesap` получается в результате следующего вызова в методе `main()`:

```
Apple.Winesap.getPrice()
```

Цены на все сорта яблок получаются при переборе перечисления в цикле `for`. Копия переменной экземпляра `price` существует для каждой константы перечислимого типа, поэтому значение, связанное с одной константой, отделено и отличается от значения, связанного с другой константой. Столь эффективный принцип оказывается возможным только благодаря реализации перечислений в виде классов, как это и сделано в языке Java.

В предыдущем примере перечисление содержит только один конструктор, но на самом деле в перечислении могут быть предоставлены две или более перегружаемых формы конструкторов, как и в любом другом классе. Например, в приведенной ниже версии перечисления `Apple` дополнительно предоставляется конструктор по умолчанию, инициализирующий цену значением `-1`, которое означает, что цена не указана.

```
// Использовать конструкторы в перечислении  
enum Apple {  
    Jonathan(10), GoldenDel(9), RedDel, Winesap(15), Cortland(8);  
  
    private int price; // цена яблока каждого сорта  
  
    // Конструктор  
    Apple(int p) { price = p; }  
  
    // Перегружаемый конструктор
```

```

Apple() { price = -1; }

int getPrice() { return price; }
}

```

Обратите внимание на то, что в этой версии перечисления `Apple` константе `RedDel` не передается аргумент. Это означает, что вызывается конструктор по умолчанию и в переменной `price` для цены на яблоко сорта `RedDel` устанавливается значение `-1`.

Однако на перечисления накладываются два ограничения. Во-первых, перечисление не может наследоваться от другого класса. И во-вторых, перечисление не может быть суперклассом. Это означает, что перечисление не может быть расширено. А в остальном перечисление ведет себя так же, как и любой другой тип класса. Самое главное — не забывать, что каждая константа перечислимого типа является объектом класса, в котором она определена.

Перечисления наследуются от класса `Enum`

Несмотря на то что при объявлении перечисления нельзя наследовать суперкласс, все перечисления автоматически наследуют от класса `java.lang.Enum`. В этом классе определяется ряд методов, доступных для использования во всех перечислениях. Класс `Enum` подробнее рассматривается в части II данной книги, но три его метода требуют хотя бы краткого описания уже теперь.

Вызвав метод `ordinal()`, можно получить значение, которое обозначает позицию константы в списке констант перечислимого типа. Это значение называется *порядковым* и извлекается из перечисления так, как показано ниже.

```
final int ordinal()
```

Этот метод возвращает порядковое значение вызывающей константы. Порядковые значения начинаются с нуля. Так, в перечислении `Apple` константа `Johnatan` имеет порядковое значение `0`, константа `GoldenDel` — `1`, константа `RedDel` — `2` и т.д.

С помощью метода `compareTo()` можно сравнить порядковые значения двух констант одного и того же перечислимого типа. Этот метод имеет следующую общую форму:

```
final int compareTo(тип_перечисления e)
```

где *тип_перечисления* обозначает тип конкретного перечисления, а *e* — константу, которую требуется сравнить с вызывающей константой. Напомним, что обе константы (вызывающая и *e*) должны относиться к одному и тому же перечислимому типу. Если порядковое значение вызывающей константы меньше, чем у константы *e*, то метод `compareTo()` возвращает отрицательное значение. Если же порядковые значения обеих констант одинаковы, возвращается ноль. А если порядковое значение вызывающей константы больше, чем у константы *e*, то возвращается положительное значение.

Вызвав метод `equals()`, переопределяющий аналогичный метод из класса `Object`, можно сравнить на равенство константу перечислимого типа с любым другим объектом. Несмотря на то что метод `equals()` позволяет сравнивать кон-

станту перечислимого типа с любым другим объектом, оба эти объекта будут равны только в том случае, если они ссылаются на одну и ту же константу из одного и того же перечисления. Простое совпадение порядковых значений не вынудит метод `equals()` вернуть логическое значение `true`, если две константы относятся к разным перечислениям.

Напомним, что две ссылки на перечисления можно сравнивать на равенство с помощью операции `==`. В следующем примере программы демонстрируется применение методов `ordinal()`, `compareTo()` и `equals()`:

```
// Продемонстрировать применение методов ordinal(), compareTo() и equals()

// Перечисление сортов яблок
enum Apple {
    Jonathan, GoldenDel, RedDel, Winesap, Cortland
}

class EnumDemo4 {
    public static void main(String args[])
    {
        Apple ap, ap2, ap3;

        // получить все порядковые значения с помощью метода ordinal()
        System.out.println("Все константы сортов яблок " +
            " и их порядковые значения: ");
        for(Apple a : Apple.values())
            System.out.println(a + " " + a.ordinal());

        ap = Apple.RedDel;
        ap2 = Apple.GoldenDel;
        ap3 = Apple.RedDel;

        System.out.println();

        // продемонстрировать применение методов compareTo() и equals()

        if(ap.compareTo(ap2) < 0)
            System.out.println(ap + " предшествует " + ap2);

        if(ap.compareTo(ap2) > 0)
            System.out.println(ap2 + " предшествует " + ap);

        if(ap.compareTo(ap3) == 0)
            System.out.println(ap + " равно " + ap3);

        System.out.println();

        if(ap.equals(ap2))
            System.out.println("Ошибка!");

        if(ap.equals(ap3))
            System.out.println(ap + " равно " + ap3);

        if(ap == ap3)
            System.out.println(ap + " == " + ap3);
    }
}
```

Ниже приведен результат, выводимый данной программой.

Все константы сортов яблок и их порядковые значения:
Jonathan 0

```
GoldenDel 1
RedDel 2
Winesap 3
Cortland 4
```

```
GoldenDel предшествует RedDel
RedDel равно RedDel
```

```
RedDel равно RedDel
RedDel == RedDel
```

Еще один пример перечисления

Перед тем как продолжить дальше, рассмотрим еще один пример, в котором применяется перечисление. В главе 9 рассматривался пример программы для автоматического принятия решений. В той ее версии переменные NO, YES, MAYBE, LATER, SOON и NEVER объявлялись в интерфейсе и использовались для представления возможных ответов. И хотя в этом нет ничего формально неверного, в данном случае лучше подходит перечисление. Ниже приведена усовершенствованная версия данной программы, в которой для представления ответов используется перечисление Answers. Можете сравнить эту версию с первоначальной из главы 9.

```
// Усовершенствованная версия программы принятия решений
// из главы 9. В этой версии для представления возможных ответов
// используется перечисление, а не переменные экземпляра
```

```
import java.util.Random;

// Перечисление возможных ответов
enum Answers {
    NO, YES, MAYBE, LATER, SOON, NEVER
}

class Question {
    Random rand = new Random();
    Answers ask() {
        int prob = (int) (100 * rand.nextDouble());

        if (prob < 15)
            return Answers.MAYBE; // 15%
        else if (prob < 30)
            return Answers.NO; // 15%
        else if (prob < 60)
            return Answers.YES; // 30%
        else if (prob < 75)
            return Answers.LATER; // 15%
        else if (prob < 98)
            return Answers.SOON; // 13%
        else
            return Answers.NEVER; // 2%
    }
}

class AskMe {
    static void answer(Answers result) {
        switch(result) {
            case NO:
                System.out.println("Нет");
                break;
        }
    }
}
```

```

        case YES:
            System.out.println("Да");
            break;
        case MAYBE:
            System.out.println("Возможно");
            break;
        case LATER:
            System.out.println("Позднее");
            break;
        case SOON:
            System.out.println("Вскоре");
            break;
        case NEVER:
            System.out.println("Никогда");
            break;
    }
}

public static void main(String args[]) {
    Question q = new Question();
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
}
}

```

Оболочки типов

Как вам должно быть уже известно, в Java для хранения основных типов данных, поддерживаемых в этом языке программирования, используются примитивные (или простые) типы вроде `int` или `double`. Примитивные типы данных, в отличие от объектов, используются для хранения простых значений из соображений производительности. Применение объектов для хранения таких значений приводит к нежелательным издержкам даже при простейших вычислениях. Поэтому примитивные типы данных не являются частью иерархии объектов и не наследуются от класса `Object`.

Несмотря на то что примитивные типы данных обеспечивают выигрыш в производительности, иногда может понадобиться объектное представление этих типов данных. Например, данные примитивного типа нельзя передать методу по ссылке. Кроме того, многие стандартные структуры данных, реализованные в Java, оперируют объектами, а это означает, что такие структуры данных нельзя применять для хранения примитивных типов. Для выхода из таких (и подобных им) ситуаций в Java предоставляются *оболочки типов*, которые представляют собой классы, заключающие примитивный тип данных в оболочку объекта. Классы-оболочки типов подробнее описываются в части II данной книги, но поскольку они имеют непосредственное отношение к автоупаковке в Java, то здесь они представлены вкратце.

К оболочкам типов относятся классы `Double`, `Float`, `Long`, `Integer`, `Short`, `Byte`, `Character` и `Boolean`, которые предоставляют обширный ряд методов, позволяющих полностью интегрировать примитивные типы в иерархию объектов в Java. В последующих разделах каждый из них рассматривается вкратце.

Класс Character

Служит оболочкой для типа `char`. Конструктор `Character()` имеет следующую общую форму:

```
Character(char символ)
```

где параметр *символ* обозначает тот символ, который будет заключен в оболочку при создании объекта типа `Character`. Чтобы получить значение типа `char`, содержащееся в объекте типа `Character`, достаточно вызвать метод `charValue()`, как показано ниже. Этот метод возвращает инкапсулированный символ.

```
char charValue()
```

Класс Boolean

Служит оболочкой для логических значений типа `boolean`. В нем определены следующие конструкторы:

```
Boolean(boolean логическое_значение)
```

```
Boolean(String логическая_строка)
```

В первом конструкторе *логическое_значение* должно быть равно `true` или `false`. А во втором конструкторе новый объект типа `Boolean` будет содержать логическое значение `true`, если *логическая_строка* содержит символьную строку `"true"` (в верхнем или нижнем регистре). В противном случае этот объект будет содержать логическое значение `false`.

Чтобы получить логическое значение типа `boolean` из объекта типа `Boolean`, достаточно вызвать метод `booleanValue()`, как показано ниже. Это метод возвращает значение типа `boolean`, эквивалентное вызывающему объекту.

```
boolean booleanValue()
```

Оболочки числовых типов

Безусловно, наиболее часто употребляемыми являются оболочки числовых типов. К ним относятся классы `Byte`, `Short`, `Integer`, `Long`, `Float` и `Double`. Все оболочки числовых типов наследуют абстрактный класс `Number`. В этом классе являются методы, возвращающие значение объекта в разных числовых форматах. Все эти методы перечислены ниже.

```
byte byteValue()  
double doubleValue()  
float floatValue()  
int intValue()  
long longValue()  
short shortValue()
```

Например, метод `doubleValue()` возвращает значение объекта в виде типа `double`, метод `floatValue()` — значение объекта в виде типа `float` и т.д. Все эти методы реализованы каждой из оболочек числовых типов.

В классах оболочек всех числовых типов определяются конструкторы, позволяющие создавать объекты из заданного значения или строкового представления

этого значения. В качестве примера ниже приведены конструкторы, определенные в классе `Integer`. Если строка не содержит числовое значение, то генерируется исключение типа `NumberFormatException`.

```
Integer(int число)
Integer(String строка)
```

В классах оболочек всех числовых типов переопределяется метод `toString()`. Он возвращает удобочитаемую форму значения, содержащегося в оболочке, что позволяет, например, выводить значение, передавая объект оболочки типа методу `println()` без дополнительного преобразования в примитивный тип. В следующем примере программы показано, как пользоваться оболочкой числового типа для инкапсуляции числового значения и последующего его извлечения:

```
// Продемонстрировать оболочку числового типа
class Wrap {
    public static void main(String args[]) {
        Integer iOb = new Integer(100);
        int i = iOb.intValue();
        System.out.println(i + " " + iOb); // выводит значения 100 и 100
    }
}
```

В этой программе целое значение **100** размещается в объекте `iOb` класса `Integer`. Затем это значение получается в результате вызова метода `intValue()` и размещается в переменной `i`.

Процесс инкапсуляции значения в объекте называется *упаковкой*. Так, в следующей строке кода из рассматриваемой здесь программы значение **100** упаковывается в объект типа `Integer`:

```
Integer iOb = new Integer(100);
```

Процесс извлечения значения из оболочки типа называется *распаковкой*. Например, в приведенной ниже строке кода из рассматриваемой здесь программы целочисленное значение распаковывается из объекта `iOb`.

```
int i = iOb.intValue();
```

Описываемая здесь общая процедура упаковки и распаковки значений стала применяться с самой первой версии Java. Но в версию Java J2SE 5 были внесены существенные усовершенствования благодаря внедрению автоупаковки, которая рассматривается ниже.

Автоупаковка

С версии JDK 5 в Java внедрены два важных средства: *автоупаковка* и *автораспаковка*. Автоупаковка — это процесс, в результате которого примитивный тип автоматически инкапсулируется (упаковывается) в эквивалентную ему оболочку типа всякий раз, когда требуется объект данного типа. Благодаря этому отпадает необходимость в явном создании объекта. Автораспаковка — это процесс автоматического извлечения значения упакованного объекта (распаковки) из оболочки типа,

когда нужно получить его значение. Благодаря этому отпадает необходимость вызывать методы вроде `intValue()` или `doubleValue()`.

Внедрение автоматической упаковки и распаковки значительно упрощает реализацию некоторых алгоритмов, исключая необходимость в ручной упаковке и распаковке значений. Это также помогает предотвратить ошибки и очень важно для обобщений, которые оперируют только объектами. И наконец, автоупаковка существенно облегчает работу с каркасом коллекций `Collection Framework`, рассматриваемым в части II.

С появлением автоупаковки отпадает необходимость в ручном создании объектов для заключения примитивных типов в оболочку. Для этого достаточно присвоить значение примитивного типа переменной ссылки на объект оболочки этого типа, а сам этот объект будет создан средствами Java автоматически. В качестве примера ниже приведен современный способ создания объекта типа `Integer` и заключения в его оболочку целочисленного значения `100`. Обратите внимание на то, что объект не создается явно оператором `new`. Это делается в Java автоматически.

```
Integer iOb = 100; // автоупаковка значения типа int
```

Чтобы распаковать объект, достаточно присвоить ссылку на него переменной соответствующего примитивного типа. Например, в следующей строке кода целочисленное значение извлекается из автоматически распаковываемого объекта `iOb`:

```
int i = iOb; // автораспаковка
```

Ниже приведена версия программы из предыдущего примера, переделанная с целью продемонстрировать автоупаковку и автораспаковку.

```
// Продемонстрировать автоупаковку/автораспаковку
class AutoBox {
    public static void main(String args[]) {

        Integer iOb = 100; // автоупаковка значения типа int

        int i = iOb;        // автораспаковка значения типа int

        System.out.println(i + " " + iOb); // выводит значения 100 и 100
    }
}
```

Автоупаковка и методы

Помимо простых случаев присваивания, автоупаковка происходит автоматически всякий раз, когда примитивный тип должен быть преобразован в объект. Автораспаковка происходит всякий раз, когда объект должен быть преобразован в примитивный тип. Таким образом, автоупаковка и автораспаковка может производиться, когда аргумент передается методу или значение возвращается из метода. Рассмотрим следующий пример программы:

```
// Автоупаковка/автораспаковка происходит при передаче
// параметров и возврате значений из методов
```

```
class AutoBox2 {
    // принять параметр типа Integer и вернуть
```



```

// значение типа int;
static int m(Integer v) {
    return v ; // автораспаковка значения типа int
}

public static void main(String args[]) {
    // Передать значение типа int методу m() и присвоить
    // возвращаемое значение объекту типа Integer.
    // Здесь значение 100 аргумента автоматически
    // упаковывается в объект типа Integer.
    // Возвращаемое значение также упаковывается
    // в объект типа Integer.
    Integer iOb = m(100);

    System.out.println(iOb);
}
}

```

Эта программа выводит следующий результат:

100

Обратите внимание на то, что в этой программе метод `m()` объявляется с параметром типа `Integer` и возвращает результат типа `int`. В теле метода `main()` числовое значение `100` передается методу `m()`. А поскольку метод `m()` ожидает получить объект типа `Integer`, то числовое значение `100` автоматически упаковывается. Затем метод `m()` возвращает эквивалент своего аргумента типа `int`. Это приводит к автораспаковке возвращаемого значения `v` типа `int`. Далее это значение присваивается переменной `iOb` типа `Integer` в методе `main()`, что снова приводит к автоупаковке возвращаемого значения типа `int`.

Автоупаковка и автораспаковка в выражениях

В общем, автоупаковка и автораспаковка производятся всякий раз, когда требуется взаимное преобразование значения примитивного типа и объекта оболочки этого типа. Это, как правило, происходит в выражениях, где автоматически распаковывается объект оболочки числового типа. Результат вычисления такого выражения снова упаковывается по мере надобности. Рассмотрим в качестве примера следующую программу:

```

// Автоупаковка/распаковка происходит в выражениях

class AutoBox3 {
    public static void main(String args[]) {

        Integer iOb, iOb2;
        int i;

        iOb = 100;
        System.out.println("Исходное значение iOb: " + iOb);

        // В следующем выражении автоматически распаковывается
        // объект iOb, выполняется приращение получаемого значения,
        // которое затем упаковывается обратно в объект iOb
        ++iOb;
        System.out.println("После ++iOb: " + iOb);
    }
}

```

```
// Здесь объект iOb распаковывается, выражение вычисляется,  
// а результат снова упаковывается и присваивается объекту iOb2  
iOb2 = iOb + (iOb / 3);  
System.out.println("iOb2 после выражения: " + iOb2);  
  
// Здесь вычисляется то же самое выражение,  
// но результат не упаковывается  
i = iOb + (iOb / 3);  
System.out.println("i после выражения: " + i);  
}  
}
```

Ниже приведен результат, выводимый данной программой.

```
Исходное значение iOb: 100  
После ++iOb: 101  
iOb2 после выражения: 134  
i после выражения: 134
```

Обратите особое внимание на следующую строку кода из данной программы:

```
++iOb;
```

В этой строке значение в переменной `iOb` увеличивается на 1. А происходит это следующим образом: объект `iOb` распаковывается, получаемое в итоге значение увеличивается на 1, а результат упаковывается обратно в тот же самый объект.

Автоматическая распаковка позволяет также сочетать числовые объекты разных типов в одном выражении. Как только числовое значение будет распаковано, вступают в действие стандартные правила продвижения и преобразования типов данных. Например, следующая программа написана совершенно верно:

```
class AutoBox4 {  
    public static void main(String args[]) {  
  
        Integer iOb = 100;  
        Double dOb = 98.6;  
  
        dOb = dOb + iOb;  
        System.out.println("dOb после выражения: " + dOb);  
    }  
}
```

Ниже приведен результат, выводимый данной программой.

```
dOb после выражения: 198.6
```

Как видите, оба объекта (из переменных `dOb` типа `Double` и `iOb` типа `Integer`) участвуют в сложении, а результат повторно упаковывается и сохраняется в объекте `dOb`.

Благодаря автоупаковке появляется возможность применять числовые объекты типа `Integer` для управления оператором `switch`. Рассмотрим в качестве примера следующий фрагмент кода:

```
Integer iOb = 2;  
  
switch(iOb) {  
    case 1: System.out.println("один");  
        break;
```

```

    case 2: System.out.println("два");
        break;
    default: System.out.println("ошибка");
}

```

Когда вычисляется оператор `switch`, объект `iOb` распаковывается и возвращается его значение типа `int`. Как показывают приведенные выше примеры программ, благодаря автоупаковке и автораспаковке применение числовых объектов в выражениях становится интуитивно понятным и значительно упрощается. Прежде для написания подобного кода приходилось выполнять приведение типов и вызывать методы вроде `intValue()`.

Автоупаковка и распаковка значений из классов `Boolean` и `Character`

Как упоминалось ранее, в Java также поддерживаются оболочки для типов `boolean` и `char` в классах `Boolean` и `Character` соответственно. Автоупаковка и автораспаковка применимы к оболочкам и этих типов данных. Рассмотрим в качестве примера следующую программу:

```

// Автоупаковка/распаковка значений из классов Boolean и Character

class AutoBox5 {
    public static void main(String args[]) {

        // Автоупаковка/распаковка логического значения типа boolean
        Boolean b = true;

        // объект b автоматически распаковывается,
        // когда он употребляется в условном операторе if
        if(b) System.out.println("b равно true");

        // Автоупаковка/распаковка значения типа char
        Character ch = 'x'; // упаковать значение типа char
        char ch2 = ch;      // распаковать значение типа char

        System.out.println("ch2 равно " + ch2);
    }
}

```

Ниже приведен результат, выводимый данной программой.

```

b равно true
ch2 равно x

```

Самое важное и достойное упоминания в этой программе — это автораспаковка объекта `b` в условном операторе `if`. Напомним, что в результате вычисления условного выражения, управляющего оператором `if`, должно возвращаться логическое значение типа `boolean`. Благодаря автораспаковке логическое значение типа `boolean`, содержащееся в объекте `b`, автоматически распаковывается при вычислении условного выражения. Таким образом, с появлением автоупаковки и распаковки появилась возможность применять объекты типа `Boolean` для управления условным оператором `if`.

Благодаря автоупаковке и автораспаковке объект типа `Boolean` можно также применять для управления операторами всех циклов в Java. Когда объект типа `Boolean` применяется в качестве условия в операторе цикла `while`, `for` или `do/while`, он автоматически распаковывается в свой эквивалент типа `boolean`. Например, следующий фрагмент кода теперь вполне допустим:

```
Boolean b;
// ...
while(b) { // ...
```

Автоупаковка и автораспаковка помогают предотвратить ошибки

Помимо тех удобств, которые предоставляют автоупаковка и распаковка, эти языковые средства могут также оказать помощь в предотвращении ошибок. Рассмотрим в качестве примера следующую программу:

```
// Ошибка, порождаемая ручной распаковкой
class UnboxingError {
    public static void main(String args[]) {

        Integer iOb = 1000;           // автоупаковка значения 1000

        int i = iOb.byteValue(); // ручная распаковка значения
                                // как относящегося к типу byte !!!
        System.out.println(i);      // значение 1000 не выводится !
    }
}
```

Эта программа выводит не предполагаемое значение **1000**, а **-24**! Дело в том, что значение в оболочке объекта `iOb` распаковывается вручную при вызове метода `byteVal()`, что приводит к усечению значения **1000**, хранящегося в этом объекте. В итоге получилось неверное значение **-24**, которое было присвоено переменной `i`. Автораспаковка предотвращает подобные ошибки, поскольку значение из объекта `iOb` всегда будет автоматически распаковываться в значение, совместимое с типом `int`.

Предупреждение

Благодаря внедрению автоупаковки и автораспаковки возникает соблазн применять исключительно объекты оболочек типов наподобие `Integer` или `Double`, пренебрегая примитивными типами данных. Теперь можно, например, написать такой код:

```
// Неудачное применение автоупаковки и автораспаковки!
Double a, b, c;

a = 10.0;
b = 4.0;

c = Math.sqrt(a*a + b*b);

System.out.println("Гипотенуза равна " + c);
```

В данном примере кода в объектах типа `Double` хранятся значения, которые используются для вычисления гипотенузы прямоугольного треугольника. Несмотря на то что этот код формально правильный и вполне работоспособный, он все же демонстрирует весьма неудачное применение автоупаковки и автораспаковки. Такой код менее эффективен, чем аналогичный код, в котором применяется примитивный тип `double`. Дело в том, что автоупаковка и автораспаковка влекут за собой дополнительные издержки, которых можно избежать, применяя элементарные типы данных.

Аннотации (метаданные)

С версии JDK 5 в Java поддерживается языковое средство, позволяющее встраивать справочную информацию в исходные файлы. Эта информация называется *аннотацией* и не меняет порядок выполнения программы. Это означает, что аннотация сохраняет неизменной семантику программы. Но эта информация может быть использована различными инструментальными средствами на стадии разработки или развертывания прикладных программ на Java. Например, аннотация может обрабатываться генераторами исходного кода. Для обозначения этого языкового средства служит также термин *метаданные*, но более описательный термин *аннотация* употребляется намного чаще.

Основы аннотирования программ

Аннотации создаются с помощью механизма, основанного на интерфейсе. Начнем их рассмотрение с простого примера. Ниже приведено объявление аннотации `MyAnno`.

```
// Простой тип аннотации.  
@interface MyAnno {  
    String str();  
    int val();  
}
```

Прежде всего обратите внимание на знак `@`, предваряющий ключевое слово `interface`. Этим компилятору указывается, что объявлен тип аннотации. Далее обратите внимание на два метода — `str()` и `val()`. Все аннотации состоят только из объявлений методов. Но тела этих методов в них не определяются. Вместо этого они реализуются средствами Java. Более того, эти методы ведут себя аналогично полям, как станет ясно в дальнейшем.

Объявление аннотации не может включать в себя ключевое слово `extends`. Но все аннотации автоматически расширяют интерфейс `Annotation`. Это означает, что `Annotation` является суперинтерфейсом для всех аннотаций. Он объявлен в пакете `java.lang.annotation`. В интерфейсе `Annotation` переопределяются методы `hashCode()`, `equals()` и `toString()`, определенные в классе `Object`. В нем также объявляется метод `annotationType()`, возвращающий объект типа `Class`, представляющий вызывающую аннотацию.

Как только аннотация будет объявлена, ею можно воспользоваться для аннотирования любых элементов прикладного кода. До версии JDK 8 аннотации можно

было использовать только в объявлениях, с чего и будет начато их подробное рассмотрение. А в версии JDK 8 появилась возможность аннотировать применение типов данных, как поясняется далее в этой главе. Но обе разновидности аннотаций применяются одним и тем же способом. Аннотацию можно связать с любым объявлением. Например, аннотировать можно классы, методы, поля, параметры и константы перечислимого типа. Аннотированной может быть даже сама аннотация. Но в любом случае аннотация предшествует остальной части объявления.

Когда применяется аннотация, ее членам присваиваются соответствующие значения. В качестве примера ниже приведен вариант применения аннотации `MyAnno` к объявлению метода.

```
// Аннотирование метода
@MyAnno(str = "Пример аннотации", val = 100)
public static void myMeth() { // ...
```

Эта аннотация связана с методом `myMeth()`. Обратите особое внимание на ее синтаксис. Сразу за именем аннотации, которому предшествует знак `@`, следует список инициализируемых ее членом в скобках. Чтобы установить значение члена аннотации, достаточно присвоить его имени данного члена. Таким образом, в данном примере строка "Пример аннотации" присваивается члену `str` аннотации `MyAnno`. Обратите также внимание на то, что в этом присваивании нет никаких скобок после имени члена `str`. Когда члену аннотации присваивается значение, используется только его имя. В данном контексте члены похожи на поля.

Правила удержания аннотаций

Прежде чем продолжить рассмотрение аннотаций, следует обсудить *правила удержания аннотаций*. Правила удержания определяют момент, когда аннотация отбрасывается. В Java определены три такие правила, инкапсулированные в перечисление `java.lang.annotation.RetentionPolicy`. Это правила `SOURCE`, `CLASS` и `RUNTIME`.

Аннотации по правилу удержания `SOURCE` содержатся только в исходном файле и отбрасываются при компиляции. Аннотации по правилу удержания `CLASS` сохраняются в файле с расширением `.class` во время компиляции. Но они недоступны для виртуальной машины JVM во время выполнения. Аннотации по правилу удержания `RUNTIME` сохраняются в файле с расширением `.class` во время компиляции и остаются доступными для виртуальной машины JVM во время выполнения. Это означает, что правило удержания `RUNTIME` предоставляет аннотации наиболее высокую степень сохраняемости.

На заметку! Аннотации объявлений локальных переменных не удерживаются в файле с расширением `.class`.

Правило удержания аннотации задается с помощью одной из встроенных аннотаций Java: `@Retention`, общая форма которой приведена ниже.

```
@Retention (правило_удержания)
```

Здесь *правило_удержания* должно быть обозначено одной из описанных ранее констант. Если для аннотации не указано никакого правила удержания, то применяется правило удержания `CLASS`.

В следующем примере аннотации `MyAnno` правило удержания `RUNTIME` устанавливается с помощью аннотации `@Retention`. Это означает, что аннотация `MyAnno` будет доступна для виртуальной машины JVM во время выполнения программы.

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str();
    int val();
}
```

Получение аннотаций во время выполнения с помощью рефлексии

Аннотации предназначены в основном для использования в инструментальных средствах разработки и развертывания прикладных программ на Java. Но если они задают правило удержания `RUNTIME`, то могут быть опрошены во время выполнения в любой программе на Java с помощью рефлексии. *Рефлексия* — это языковое средство для получения сведений о классе во время выполнения программы. Прикладной программный интерфейс (API) для рефлексии входит в состав пакета `java.lang.reflect`. Пользоваться рефлексией можно самыми разными способами, но мы, к сожалению, не имеем здесь возможности, чтобы рассмотреть их. Тем не менее обратимся к нескольким примерам применения рефлексии, имеющим отношение к аннотациям.

Первый шаг с целью воспользоваться рефлексией состоит в получении объекта типа `Class`. Этот объект представляет класс, аннотацию которого требуется получить. А `Class` относится к числу встроенных в Java классов и определен в пакете `java.lang`. Более подробно он рассматривается в части II данной книги. Имеются разные способы получения объекта типа `Class`. Самый простой из них — вызвать метод `getClass()`, определенный в классе `Object`. Его общая форма приведена ниже. Этот метод возвращает объект типа `Class`, который представляет вызывающий объект.

```
final Class<?> getClass()
```

На заметку! Обратите внимание на знаки `<?>`, следующие за именем `Class` в приведенном выше объявлении метода `getClass()`. Это обозначение имеет отношение к обобщениям в Java. Метод `getClass()` и несколько других связанных с рефлексией методов, обсуждаемых в этой главе, используются в обобщениях, подробнее описываемых в главе 14. Но для того чтобы уяснить основополагающие принципы рефлексии, разбираться в обобщениях совсем не обязательно.

Имея в своем распоряжении объект типа `Class`, можно воспользоваться его методами для получения сведений о различных элементах, объявленных в классе, включая и его аннотацию. Если требуются аннотации, связанные с определенным элементом, объявленным в классе, сначала следует получить объект, представляющий этот элемент. Например, класс `Class` предоставляет (среди прочего) методы `getMethod()`, `getField()` и `getConstructor()`, возвращающие сведения о методе, поле и конструкторе соответственно. Эти методы возвращают объекты типа `Method`, `Field` и `Constructor`.

Чтобы понять этот процесс, рассмотрим в качестве примера получение аннотаций, связанных с методом. Для этого сначала получается объект типа `Class`, представляющий класс, затем вызывается метод `getMethod()` для этого объекта с указанным именем искомого метода. У метода `getMethod()` имеется следующая общая форма:

```
Method getMethod(String имя_метода, Class<?> ... типы_параметров)
```

Имя искомого метода передается в качестве аргумента *имя_метода*. Если этот метод принимает аргументы, то объекты типа `Class`, представляющие их типы, должны быть также указаны в качестве аргумента *типы_параметров*. Обратите внимание на то, что аргумент *типы_параметров* представляет собой список аргументов переменной длины. Это позволяет задать столько типов параметров, сколько требуется, в том числе и не указывать их вообще. Метод `getMethod()` возвращает объект типа `Method`, который представляет метод. Если метод не удастся найти, то генерируется исключение типа `NoSuchMethodException`.

Из объекта типа `Class`, `Method`, `Field` или `Constructor` можно получить конкретные аннотации, связанные с этим объектом, вызвав метод `getAnnotation()`. Его общая форма приведена ниже.

```
<A extends Annotation> getAnnotation(Class<A> тип_аннотации)
```

Здесь параметр *тип_аннотации* обозначает объект типа `Class`, представляющий требующуюся аннотацию. Этот метод возвращает ссылку на аннотацию. Используя эту ссылку, можно получить значения, связанные с членами аннотации. Метод `getAnnotation()` возвращает пустое значение `null`, если аннотация не найдена. В этом случае у искомой аннотации отсутствует аннотация `@Retention`, устанавливающая правило удержания `RUNTIME`. Ниже приведен пример программы, подытоживающий все сказанное выше. В этой программе рефлексия применяется для вывода аннотации, связанной с конкретным методом.

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// Объявление типа аннотации
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str();
    int val();
}

class Meta {
    // аннотировать метод
    @MyAnno(str = "Пример аннотации", val = 100)
    public static void myMeth() {
        Meta ob = new Meta();

        // получить аннотацию из метода
        // и вывести значения ее членов
        try {
            // сначала получить объект типа Class,
            // представляющий данный класс
            Class<?> c = ob.getClass();

            // затем получить объект типа Method,
            // представляющий данный метод
            Method m = c.getMethod("myMeth");
```



```

        // далее получить аннотацию для данного класса
        MyAnno anno = m.getAnnotation(MyAnno.class);

        // и наконец, вывести значения членов аннотации
        System.out.println(anno.str() + " " + anno.val());
    } catch (NoSuchMethodException exc) {
        System.out.println("Метод не найден.");
    }
}

public static void main(String args[]) {
    myMeth();
}
}

```

Ниже приведен результат, выводимый данной программой.

Пример аннотации 100

В этой программе рефлексия применяется, как описано выше, для получения и вывода значений переменных `str` и `val` из аннотации `MyAnno`, связанной с методом `myMeth()` из класса `Meta`. Здесь следует обратить особое внимание на следующее. Во-первых, это выражение `MyAnno.class` в следующей строке кода:

```
MyAnno anno = m.getAnnotation(MyAnno.class);
```

Это выражение вычисляется как объект `Class`, относящийся к типу `MyAnno`, т.е. к искомой аннотации, и называется *литералом класса*. Такое выражение можно использовать всякий раз, когда требуется объект `Class` известного класса. Например, в следующем операторе получается объект `Class` для класса `Meta`:

```
Class<?> c = Meta.class;
```

Безусловно, такой подход годится лишь в том случае, если заранее известно имя класса искомого объекта, что возможно далеко не всегда. В общем, литерал класса можно получить для классов, интерфейсов, примитивных типов и массивов. (Напомним, что синтаксис `<?>` имеет отношение к обобщениям в Java, рассматриваемым в главе 14.)

И во-вторых, это способ получения значений, связанных с переменными `str` и `val`, когда они выводятся в следующей строке кода:

```
System.out.println(anno.str() + " " + anno.val());
```

Обратите внимание на то, что для обращения к ним применяется синтаксис вызова методов. Тот же самый подход применяется всякий раз, когда требуется получить член аннотации.

Второй пример применения рефлексии

В предыдущем примере у метода `myMeth()` отсутствовали параметры. Иными словами, когда вызывался метод `getMethod()`, передавалось только имя `myMeth` искомого метода. Но для того, чтобы получить метод, у которого имеются параметры, следует задать объекты класса, представляющие типы этих параметров, в виде аргументов метода `getMethod()`. Например, ниже приведена немного измененная версия программы из предыдущего примера.

```
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str();
    int val();
}

class Meta {

    // У метода myMeth() теперь имеются два аргумента
    @MyAnno(str = "Два параметра", val = 19)
    public static void myMeth(String str, int i)
    {
        Meta ob = new Meta();

        try {
            Class<?> c = ob.getClass();

            // Здесь указываются типы параметров
            Method m = c.getMethod("myMeth", String.class, int.class);

            MyAnno anno = m.getAnnotation(MyAnno.class);

            System.out.println(anno.str() + " " + anno.val());
        } catch (NoSuchMethodException exc) {
            System.out.println("Метод не найден.");
        }
    }

    public static void main(String args[]) {
        myMeth("Тест", 10);
    }
}
```

Ниже приведен результат выполнения этой версии программы.

Два параметра 19

В этой версии метод `myMeth()` принимает параметры типа `String` и `int`. Чтобы получить сведения об этом методе, следует вызвать метод `getMethod()` так, как показано ниже, где объекты `Class`, представляющие типы `String` и `int`, передаются в виде дополнительных аргументов.

```
Method m = c.getMethod("myMeth", String.class, int.class);
```

Получение всех аннотаций

Для того чтобы получить сразу все аннотации, имеющие аннотацию `@Retention` с установленным правилом удержания `RUNTIME` и связанные с искомым элементом, достаточно вызвать метод `getAnnotations()` для этого элемента. Ниже приведена общая форма метода `getAnnotations()`.

```
Annotation[] getAnnotations()
```

Метод `getAnnotations()` возвращает массив аннотаций. Этот метод может быть вызван для объектов типа `Class`, `Method`, `Constructor` и `Field`.

Ниже приведен еще один пример применения рефлексии, демонстрирующий получение всех аннотаций, связанных с классом и методом. В программе из этого примера сначала объявляются две аннотации, которые затем используются для аннотирования класса и метода.

```
// Показать все аннотации для класса и метода
import java.lang.annotation.*;
import java.lang.reflect.*;

@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str();
    int val();
}

@Retention(RetentionPolicy.RUNTIME)
@interface What {
    String description();
}

@What(description = "Аннотация тестового класса")
@MyAnno(str = "Meta2", val = 99)
class Meta2 {

    @What(description = "Аннотация тестового метода")
    @MyAnno(str = "Testing", val = 100)
    public static void myMeth() {
        Meta2 ob = new Meta2();

        try {
            Annotation annos[] = ob.getClass().getAnnotations();

            // вывести все аннотации для класса Meta2
            System.out.println("Все аннотации для класса Meta2:");
            for(Annotation a : annos)
                System.out.println(a);

            System.out.println();

            // вывести все аннотации для метода myMeth()
            Method m = ob.getClass().getMethod("myMeth");
            annos = m.getAnnotations();

            System.out.println("Все аннотации для метода myMeth():");
            for(Annotation a : annos)
                System.out.println(a);
        } catch (NoSuchMethodException exc) {
            System.out.println("Метод не найден.");
        }
    }

    public static void main(String args[]) {
        myMeth();
    }
}
```

Ниже приведен результат выполнения данной программы.

```
Все аннотации для класса Meta2:
@What(description=Аннотация тестового класса)
@MyAnno(str=Meta2, val=99)
```

```

Все аннотации для метода myMeth():
@What(description=Аннотация тестового метода)
@MyAnno(str=Testing, val=100)

```

В этой программе метод `getAnnotations()` используется для получения массива всех аннотаций, связанных с классом `Meta2` и методом `myMeth()`. Как пояснялось ранее, метод `getAnnotations()` возвращает массив объектов типа `Annotation`. Напомним, что `Annotation` является суперинтерфейсом для всех интерфейсов аннотаций и что в нем переопределяется метод `toString()` из класса `Object`. Так, если выводится ссылка на интерфейс `Annotation`, то вызывается его метод `toString()` для создания символьной строки, описывающей аннотацию, что и демонстрирует предыдущий пример.

Интерфейс `AnnotatedElement`

Методы `getAnnotation()` и `getAnnotations()`, использованные в предыдущем примере, определены в интерфейсе `AnnotatedElement`, который входит в состав пакета `java.lang.reflect`. Этот интерфейс поддерживает рефлексия для аннотации и реализуется в классах `Method`, `Field`, `Constructor`, `Class` и `Package`.

Кроме методов `getAnnotation()` и `getAnnotations()`, в интерфейсе `AnnotatedElement` определяются два других метода. Первый метод, `getDeclaredAnnotations()`, имеет следующую общую форму:

```
Annotation[] getDeclaredAnnotations()
```

Данный метод возвращает ненаследуемые аннотации, присутствующие в вызываемом объекте. А второй метод, `isAnnotationPresent()`, имеет такую общую форму:

```
boolean isAnnotationPresent(Class<? extends Annotation> тип_аннотации)
```

Этот метод возвращает логическое значение `true`, если аннотация, заданная в виде аргумента `тип_аннотации`, связана с вызываемым объектом. В противном случае возвращается логическое значение `false`. В версии JDK 8 эти методы дополнены методами `getDeclaredAnnotation()`, `getAnnotationsByType()` и `getDeclaredAnnotationsByType()`. Два последних метода предназначены для повторяющихся аннотаций, которые рассматриваются в конце этой главы.

Использование значений по умолчанию

Для членов аннотации можно указать значения по умолчанию, которые будут выбираться при применении аннотации, если для них явно не заданы другие значения. Чтобы указать значение по умолчанию, достаточно ввести ключевое слово `default` в объявлении члена аннотации. Ниже приведена общая форма для указания значений членов аннотации по умолчанию.

```
тип member() default значение;
```

Здесь `значение` и `тип` должны быть совместимы по типу данных. Ниже показано, как выглядит аннотация `@MyAnno`, переделанная с учетом значений по умолчанию.

```
// Объявление типа аннотации, включающее значения по умолчанию
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str() default "Тестирование";
    int val() default 9000;
}
```

В этом объявлении определяются значения по умолчанию **"Тестирование"** и **9000** для членов `str` и `val` аннотации `@MyAnno` соответственно. Это означает, что значения ни одного из членов аннотации `@MyAnno` указывать необязательно, когда она применяется. Но любому из них или обоим сразу можно, если требуется, присвоить конкретное значение явным образом. Таким образом, имеются четыре способа применения аннотации `@MyAnno`:

```
@MyAnno() // значения str и val принимаются по умолчанию
@MyAnno(str = "Некоторая строка") // значение val - по умолчанию
@MyAnno(val = 100) // значение str - по умолчанию
@MyAnno(str = "Тестирование", val = 100) // значения не по умолчанию
```

В следующем примере программы демонстрируется использование значений членов аннотации по умолчанию:

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// Объявление типа аннотации, включая значения ее членов по умолчанию
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
    String str() default "Тестирование";
    int val() default 9000;
}

class Meta3 {

    // аннотировать метод, используя значения по умолчанию
    @MyAnno()
    public static void myMeth() {
        Meta3 ob = new Meta3();

        // получить аннотацию для данного метода
        // и вывести значения ее членов
        try {
            Class<?> c = ob.getClass();

            Method m = c.getMethod("myMeth");

            MyAnno anno = m.getAnnotation(MyAnno.class);

            System.out.println(anno.str() + " " + anno.val());
        } catch (NoSuchMethodException exc) {
            System.out.println("Метод не найден.");
        }
    }

    public static void main(String args[]) {
        myMeth();
    }
}
```

Ниже приведен результат, выводимый данной программой.

Тестирование 9000

Аннотации-маркеры

Это — специальный вид аннотаций, которые не содержат членов. Единственное назначение *аннотации-маркера* — пометить объявление, для чего достаточно наличия такого маркера, как аннотации. Лучший способ выяснить, присутствует ли в прикладном коде аннотация-маркер, — вызвать метод `isAnnotationPresent()`, определенный в интерфейсе `AnnotatedElement`.

Рассмотрим пример применения аннотации-маркера. У такой аннотации отсутствуют члены, поэтому достаточно определить, присутствует ли она в прикладном коде:

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// Аннотация-маркер
@Retention(RetentionPolicy.RUNTIME)
@interface MyMarker { }

class Marker {
    // аннотировать метод с помощью маркера
    // Обратите внимание на обязательность скобок ()
    @MyMarker
    public static void myMeth() {
        Marker ob = new Marker();

        try {
            Method m = ob.getClass().getMethod("myMeth");

            // определить наличие аннотации
            if(m.isAnnotationPresent(MyMarker.class))
                System.out.println("Аннотация-маркер MyMarker присутствует.");

        } catch (NoSuchMethodException exc) {
            System.out.println("Метод не найден.");
        }
    }

    public static void main(String args[]) {
        myMeth();
    }
}
```

Приведенный ниже результат подтверждает наличие аннотации-маркера `@MyMarker`.

Аннотация-маркер `MyMarker` присутствует.

Применяя аннотацию-маркер `@MyMarker`, совсем не обязательно указывать скобки после ее имени. Это означает, что аннотация-маркер `@MyMarker` применяется просто по ее имени, как показано ниже. И хотя указание скобок после имени аннотации-маркера не считается ошибкой, делать это совсем не обязательно.

`@MyMarker`

Одночленные аннотации

Очевидно, что *одночленная аннотация* состоит из единственного члена. Она действует подобно обычной аннотации, за исключением того, что допускает сокращен-

ную форму указания значения члена. Когда в такой аннотации присутствует только один член, достаточно задать его значение, а когда она применяется, указывать имя ее члена необязательно. Но для применения этой сокращенной формы единственный член аннотации должен иметь имя `value`. Ниже приведен пример программы, демонстрирующий создание и применение одночленной аннотации.

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// Одночленная аннотация
@Retention(RetentionPolicy.RUNTIME)
@interface MySingle {
    int value(); // эта переменная должна иметь имя value
}

class Single {

    // аннотировать метод одночленной аннотацией
    @MySingle(100)
    public static void myMeth() {
        Single ob = new Single();

        try {
            Method m = ob.getClass().getMethod("myMeth");

            MySingle anno = m.getAnnotation(MySingle.class);

            System.out.println(anno.value()); // выводит значение 100
        } catch (NoSuchMethodException exc) {
            System.out.println("Метод не найден.");
        }
    }

    public static void main(String args[]) {
        myMeth();
    }
}
```

Как и предполагалось, эта программа выводит значение **100**. Здесь одночленная аннотация `@MySingle` применяется для аннотирования метода `myMeth()`, как показано ниже. Обратите внимание на то, что указывать операцию присваивания `value =` необязательно.

```
@MySingle(100)
```

Синтаксис одночленных аннотаций можно использовать и в том случае, когда применяется аннотация с другими членами, но все остальные члены должны иметь значения по умолчанию. Например, в приведенном ниже фрагменте кода в аннотацию вводится дополнительный член `xyz` с нулевым значением по умолчанию.

```
@interface SomeAnno {
    int value();
    int xyz() default 0;
}
```

Если же требуется использовать значение по умолчанию для члена `xyz`, можно применить аннотацию `@SomeAnno`, как показано ниже, просто указав значение члена `value` с помощью синтаксиса одночленных аннотаций.

```
@SomeAnno(88)
```

В этом случае член `xyz` по умолчанию принимает нулевое значение, а член `value` — значение `88`. Разумеется, чтобы задать другое значение члена `xyz`, придется инициировать оба члена аннотации явным образом, как показано ниже. Напомним, что для применения одночленной аннотации ее член должен непременно иметь имя `value`.

```
@SomeAnno(value = 88, xyz = 99)
```

Встроенные аннотации

В Java определено немало встроенных аннотаций. Большинство встроенных аннотаций имеют специальное назначение, но девять из них — общее назначение. Следующие четыре аннотации из этих девяти импортируются из пакета `java.lang.annotation`: `@Retention`, `@Documented`, `@Target` и `@Inherited`. А еще пять аннотаций — `@Override`, `@Deprecated`, `@FunctionalInterface`, `@SafeVarargs` и `@SuppressWarnings` — входят в состав пакета `java.lang`. Каждая из них описана ниже.

На заметку! В версии JDK 8 пакет `java.lang.annotation` был дополнен аннотациями `@Repeatable` и `@Native`. Первая из них служит для поддержки повторяющихся аннотаций, как поясняется далее в этой главе, а вторая — для аннотирования полей, доступных из платформенно-ориентированного кода.

Аннотация `@Retention`

Предназначена для применения только в качестве аннотации к другим аннотациям. Она определяет правило удержания, как пояснялось ранее в этой главе.

Аннотация `@Documented`

Служит маркерным интерфейсом, сообщающим инструментальному средству разработки, что аннотация должна быть документирована. Она предназначена для применения только в качестве аннотации к объявлению другой аннотации.

Аннотация `@Target`

Задаёт типы элементов, к которым можно применять аннотацию. Она предназначена для применения только в качестве аннотации к другим аннотациям. Аннотация `@Target` принимает один аргумент, который должен быть константой из перечисления `ElementType`. Этот аргумент задаёт типы объявляемых элементов, к которым можно применять аннотацию. Эти константы приведены в табл. 12.1 вместе с типами объявляемых элементов, к которым они относятся.

В аннотации `@Target` можно задать одно или несколько значений этих констант. Чтобы задать несколько значений, их следует указать списком, заключив в фигурные скобки. Например, чтобы указать, что аннотация применяется только к полям и локальным переменным, достаточно определить следующую аннотацию `@Target`:

```
@Target( { ElementType.FIELD, ElementType.LOCAL_VARIABLE } )
```


Таблица 12.1. Константы из перечисления `ElementType`

Целевая константа	Объявляемый элемент, к которому можно применять аннотацию
<code>ANNOTATION_TYPE</code>	Другая аннотация
<code>CONSTRUCTOR</code>	Конструктор
<code>FIELD</code>	Поле
<code>LOCAL_VARIABLE</code>	Локальная переменная
<code>METHOD</code>	Метод
<code>PACKAGE</code>	Пакет
<code>PARAMETER</code>	Параметр
<code>TYPE</code>	Класс, интерфейс или перечисление
<code>TYPE_PARAMETER</code>	Параметр типа (добавлено в версии JDK 8)
<code>TYPE_USE</code>	Использование типа (добавлено в версии JDK 8)

В отсутствие обозначения `@Target` аннотацию можно применять к любому объявляемому элементу, за исключением параметров типов. Именно поэтому зачастую лучше указывать целевые константы явным образом, чтобы ясно обозначить назначение аннотации.

Аннотация `@Inherited`

Это — аннотация-маркер, которую можно применять только в другом объявлении аннотации. Более того, она оказывает воздействие только на те аннотации, которые будут применяться в объявлениях классов. Аннотация `@Inherited` обуславливает наследование аннотации из суперкласса в подклассе. Так, если конкретная аннотация запрашивается в подклассе, то в отсутствие этой аннотации в подклассе проверяется ее присутствие в суперклассе. Если запрашиваемая аннотация присутствует в суперклассе и аннотирована как `@Inherited`, то она будет возвращена по запросу.

Аннотация `@Override`

Это — аннотация-маркер, которую можно применять только в методах. Метод, аннотированный как `@Override`, должен переопределять метод из суперкласса. Если он этого не сделает, во время компиляции возникнет ошибка. Эта аннотация служит для гарантии того, что метод из суперкласса будет действительно переопределен, а не просто перегружен.

Аннотация `@Deprecated`

Эта аннотация-маркер обозначает, что объявление устарело и должно быть заменено более новой формой.

Аннотация `@FunctionalInterface`

Эта аннотация-маркер внедрена в версии JDK 8 и предназначена для применения в интерфейсах. Она обозначает, что аннотируемый интерфейс явля-

ется *функциональным*, т.е. содержит один и только один абстрактный метод. Функциональные интерфейсы применяются в лямбда-выражениях (подробнее о тех и других речь пойдет в главе 15). Если же аннотируемый интерфейс не является функциональным, то во время компиляции возникает ошибка. Следует, однако, иметь в виду, что для создания функционального интерфейса аннотация `@FunctionalInterface` не требуется. Следовательно, эта аннотация носит исключительно информативный характер.

Аннотация `@SafeVarargs`

Это — аннотация-маркер, применяемая в методах и конструкторах. Она указывает на отсутствие каких-нибудь небезопасных действий, связанных с параметром переменной длины. Эта аннотация служит для подавления непроверяемых предупреждений, возникающих в коде, который в остальном является безопасным, в связи с применением неовеществляемых типов аргументов переменной длины и получением экземпляра параметризованного массива. (Неовеществляемый тип — это, по существу, обобщенный тип, как поясняется в главе 14, посвященной обобщениям.) Эту аннотацию следует применять только к методам или конструкторам с переменным количеством аргументов, объявляемым как `static` или `final`.

Аннотация `@SuppressWarnings`

Эта аннотация обозначает, что следует подавить одно или несколько предупреждений, которые могут быть выданы компилятором. Подавляемые предупреждения указываются по имени в строковой форме.

Типовые аннотации

В версии JDK 8 расширены места, в которых могут применяться аннотации. Раньше аннотации допускались только в объявлениях, как было показано в предыдущих примерах. Но с выпуском версии JDK 8 появилась возможность указывать аннотации везде, где применяются типы данных. Такая расширенная возможность применения аннотаций называется *типовой аннотацией*. Например, аннотировать можно тип, возвращаемый методом, тип объекта по ссылке `this` в теле метода, приведение типов, уровни доступа к массиву, наследуемый класс, оператор `throws`, а также обобщенные типы, включая границы параметров и аргументы обобщенного типа (более подробно обобщения рассматриваются в главе 14).

Типовые аннотации важны потому, что они позволяют выполнять дополнительные проверки прикладного кода различными инструментальными средствами на стадии разработки, чтобы предотвратить ошибки. Следует иметь в виду, что эти проверки, как правило, не производятся компилятором по команде `javac`. Для этой цели служит отдельное инструментальное средство, хотя оно и могло бы действовать в качестве модуля, подключаемого к компилятору.

В типовую компиляцию должна быть включена целевая константа `ElementType.TYPE_USE`. (Как пояснялось ранее, достоверные целевые константы аннотаций указываются с помощью аннотации `@Target`.) Типовая аннотация при-

меняется к тому типу данных, которому она предшествует. Так, если обозначить типовую аннотацию как `@TypeAnno`, то приведенная ниже строка кода считается вполне допустимой. В этой строке кода аннотация `@TypeAnno` аннотирует исключение типа `NullPointerException` в операторе `throws`:

```
void myMeth() throws @TypeAnno NullPointerException { // ...
```

Аннотировать можно также тип объекта по ссылке `this` (так называемого *получателя*). Как вам должно быть уже известно, ссылка `this` является неявным аргументом во всех методах экземпляра и делается на вызывающий объект. Для аннотирования такого типа данных требуется еще одна новая возможность, появившаяся в версии JDK 8. Теперь ссылке `this` можно явным образом объявлять в качестве первого параметра метода. В этом объявлении тип объекта по ссылке `this` должен соответствовать типу его класса, как показано в приведенном ниже примере.

```
class SomeClass { int myMeth(SomeClass this, int i, int j) { // ...
```

В данном примере типом объекта по ссылке `this` является класс `SomeClass`, поскольку метод `myMeth()` определен в этом классе. С помощью такого объявления теперь можно аннотировать тип объекта по ссылке `this`. Так, если снова обозначить типовую аннотацию как `@TypeAnno`, то приведенная ниже строка кода считается вполне допустимой.

```
int myMeth(@TypeAnno SomeClass this, int i, int j) { // ...
```

Однако если объект по ссылке `this` не аннотируется, то объявлять его совсем не обязательно. Ведь если объект по ссылке `this` не объявляется, он все равно передается неявным образом. И это положение в версии JDK 8 не изменилось. Кроме того, явное объявление объекта по ссылке `this` никоим образом не меняет сигнатуру метода, поскольку такое объявление все равно делается неявно по умолчанию. Опять же объект следует объявлять по ссылке `this` лишь в том случае, если к нему требуется применить аннотацию. И в этом случае ссылка `this` *должна* быть указана в качестве первого параметра метода.

В приведенном ниже примере программы демонстрируется ряд мест, где можно применять типовую аннотацию. В этой программе определяется ряд аннотаций, среди которых имеются типовые аннотации. Имена и целевые константы аннотаций приведены в табл. 12.2.

Таблица 12.2. Имена и целевые константы аннотаций

Аннотация	Целевая константа
<code>@TypeAnno</code>	<code>ElementType.TYPE_USE</code>
<code>@MaxLen</code>	<code>ElementType.TYPE_USE</code>
<code>@NotZeroLen</code>	<code>ElementType.TYPE_USE</code>
<code>@Unique</code>	<code>ElementType.TYPE_USE</code>
<code>@What</code>	<code>ElementType.TYPE_PARAMETER</code>
<code>@EmptyOK</code>	<code>ElementType.FIELD</code>
<code>@Recommended</code>	<code>ElementType.METHOD</code>

Следует заметить, что аннотации `@EmptyOK`, `@Recommended` и `@What` не являются типовыми. Они включены в табл. 12.2 лишь для целей сравнения. Особый интерес представляет аннотация `@What`, которая применяется для аннотирования объявляемого параметра типа и является еще одним новым средством аннотирования, внедренным в версии JDK 8. Применение каждой аннотации поясняется в комментариях к приведенному ниже примеру программы.

```
// Продемонстрировать применение нескольких типовых аннотаций
import java.lang.annotation.*;
import java.lang.reflect.*;

// Аннотация-маркер, которую можно применить к типу данных
@Target(ElementType.TYPE_USE)
@interface TypeAnno { }

// Еще одна аннотация-маркер, которую можно применить к типу данных
@Target(ElementType.TYPE_USE)
@interface NotZeroLen { }

// И еще одна аннотация-маркер, которую можно применить к типу данных
@Target(ElementType.TYPE_USE)
@interface Unique { }

// Параметризованная аннотация, которую можно применить к типу данных
@Target(ElementType.TYPE_USE)
@interface MaxLen {
    int value();
}

// Аннотация, которую можно применить к параметру типа
@Target(ElementType.PARAMETER)
@interface What {
    String description();
}

// Аннотация, которую можно применить в объявлении поля
@Target(ElementType.FIELD)
@interface EmptyOK { }

// Аннотация, которую можно применить в объявлении метода
@Target(ElementType.METHOD)
@interface Recommended { }

// применить аннотацию в параметре типа
class TypeAnnoDemo<@What(description = "Данные обобщенного типа") T> {

    // применить типовую аннотацию в конструкторе
    public @Unique TypeAnnoDemo() {}

    // аннотировать тип (в данном случае – String), но не поле
    @TypeAnno String str;

// аннотировать тест поля
    @EmptyOK String test;

    // применить типовую аннотацию для аннотирования
    // ссылки this на объект (получатель)
    public int f(@TypeAnno TypeAnnoDemo<T> this, int x) {
        return 10;
    }
}
```

```

}

// аннотировать возвращаемый тип
public @TypeAnno Integer f2(int j, int k) {
    return j+k;
}

// аннотировать объявление метода
public @Recommended Integer f3(String str) {
    return str.length() / 2;
}

// применить типовую аннотацию в операторе throws
public void f4() throws @TypeAnno NullPointerException {
    // ...
}

// аннотировать уровни доступа к массиву
String @MaxLen(10) [] @NotZeroLen [] w;

// аннотировать тип элемента массива
@TypeAnno Integer[] vec;

public static void myMeth(int i) {

    // применить типовую аннотацию в аргументе типа
    TypeAnnoDemo<@TypeAnno Integer> ob =
        new TypeAnnoDemo<@TypeAnno Integer>();

    // применить типовую аннотацию в операторе new
    @Unique TypeAnnoDemo<Integer> ob2 =
        new @Unique TypeAnnoDemo<Integer>();

    Object x = new Integer(10);
    Integer y;

    // применить типовую аннотацию в приведении типов
    y = (@TypeAnno Integer) x;
}

public static void main(String args[]) {
    myMeth(10);
}

// применить типовую аннотацию в выражении наследования
class SomeClass extends @TypeAnno TypeAnnoDemo<Boolean> {}
}

```

Большинство аннотаций в приведенном выше примере программы самоочевидны, тем не менее некоторые из них требуют дополнительных разъяснений. Это относится прежде всего к сравнению аннотации типа, возвращаемого методом, с аннотацией объявления метода. Обратите особое внимание на объявление следующих двух методов в рассматриваемом здесь примере программы:

```

// аннотировать возвращаемый тип
public @TypeAnno Integer f2(int j, int k) {
    return j+k;
}

// аннотировать объявление метода

```

```
public @Recommended Integer f3(String str) {
    return str.length() / 2;
}
```

Как видите, в обоих случаях аннотация предшествует типу, возвращаемому методом (в данном случае — `Integer`). Но в обоих случаях аннотируются два разных элемента. В первом случае аннотация `@TypeAnno` аннотирует тип, возвращаемый методом `f2()`, поскольку в ней указана целевая константа `ElementType.TYPE_USE`, а это означает, что она служит аннотацией к использованию типов. А во втором случае аннотация `@Recommended` аннотирует объявление самого метода, поскольку в ней указана целевая константа `ElementType.METHOD`. В итоге аннотация `@Recommended` применяется к объявлению метода, а не к типу, возвращаемому этим методом. Таким образом, указание целевой константы позволяет устранить кажущуюся неоднозначность в толковании аннотации объявления метода и аннотации типа, возвращаемого этим методом.

В отношении аннотирования типа, возвращаемого методом, следует также заметить, что аннотировать возвращаемый тип `void` нельзя. Не меньший интерес вызывают и аннотации полей, как показано ниже.

```
// аннотировать тип (в данном случае — String), но не поле
@TypeAnno String str;

// аннотировать тест поля
@EmptyOK String test;
```

В данном случае аннотация `@TypeAnno` аннотирует тип `String`, тогда как аннотация `@EmptyOK` — тест поля. Несмотря на то что обе аннотации предшествуют всему объявлению, у них разные целевые константы, указываемые в зависимости от типа целевого элемента. Так, если в аннотации указана целевая константа `ElementType.TYPE_USE`, то аннотируется тип. А если в ней указана целевая константа `ElementType.FIELD`, то аннотируется поле. Следовательно, рассматриваемый здесь случай похож на описанный выше случай аннотирования метода тем, что в нем отсутствует неоднозначность. Тот же самый механизм позволяет устранить неоднозначность и в аннотациях локальных переменных.

Обратите далее внимание на аннотирование объекта по ссылке `this` (получателя):

```
public int f(@TypeAnno TypeAnnoDemo<T> this, int x) {
```

В данном случае ссылка `this` указывается в качестве первого параметра и относится к типу `TypeAnnoDemo`, т.е. к классу, членом которого является метод `f()`. Как пояснялось ранее, начиная с версии `JDK 8` ссылку `this` можно явно указывать в качестве параметра в объявлении метода экземпляра ради применения типовой аннотации.

И наконец, рассмотрим аннотирование уровней доступа к методу в следующей строке кода:

```
String @MaxLen(10) [] @NotZeroLen [] w;
```

В этом объявлении аннотация `@MaxLen` аннотирует тип первого уровня доступа к методу, тогда как аннотация `@NotZeroLen` — тип второго уровня. А в приведенном ниже объявлении аннотируется элемент массива типа `Integer`.

```
@TypeAnno Integer[] vec;
```

Повторяющиеся аннотации

Еще одна новая возможность, появившаяся в версии JDK 8, позволяет повторять аннотации в одном и том же элементе. Такие аннотации называются *повторяющимися*. Для того чтобы сделать аннотацию повторяющейся, ее следует снабдить аннотацией `@Repeatable`, определенной в пакете `java.lang.annotation`. В ее поле `value` указывается тип контейнера для повторяющейся аннотации. Такой контейнер указывается в виде аннотации, для которой поле `value` является массивом типа повторяющейся аннотации. Следовательно, чтобы сделать аннотацию повторяющейся, прежде нужно создать контейнерную аннотацию, а затем указать ее тип в качестве аргумента аннотации `@Repeatable`.

Для доступа к повторяющимся аннотациями с помощью такого метода, как, например, `getAnnotation()`, следует воспользоваться контейнерной, а не самой повторяющейся аннотацией. Именно такой подход и демонстрируется в приведенном ниже примере программы. В этой программе представленная ранее версия аннотации `MyAnno` преобразуется в повторяющуюся аннотацию, а затем демонстрируется ее применение.

```
// Продемонстрировать применение повторяющейся аннотации

import java.lang.annotation.*;
import java.lang.reflect.*;

// сделать аннотацию MyAnno повторяющейся
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(MyRepeatedAnnos.class)
@interface MyAnno {
    String str() default "Тестирование";
    int val() default 9000;
}

// Это контейнерная аннотация
@Retention(RetentionPolicy.RUNTIME)
@interface MyRepeatedAnnos {
    MyAnno[] value();
}

class RepeatAnno {

    // повторить аннотацию MyAnno в методе myMeth()
    @MyAnno(str = "Первая аннотация", val = -1)
    @MyAnno(str = "Вторая аннотация", val = 100)
    public static void myMeth(String str, int i)
    {
        RepeatAnno ob = new RepeatAnno();

        try {
            Class<?> c = ob.getClass();

            // получить аннотации для метода myMeth()
            Method m = c.getMethod("myMeth", String.class, int.class);

            // вывести повторяющиеся аннотации MyAnno
            Annotation anno = m.getAnnotation(MyRepeatedAnnos.class);
            System.out.println(anno);
        }
    }
}
```

```

    } catch (NoSuchMethodException exc) {
        System.out.println("Метод не найден.");
    }
}

public static void main(String args[]) {
    myMeth("тест", 10);
}
}

```

Ниже приведен результат, выводимый данной программой.

```

@MyRepeatedAnnos (value=[@MyAnno(str=Первая аннотация, val=-1),
@MyAnno(str=Вторая аннотация, val=100)])

```

Как пояснялось ранее, чтобы сделать аннотацию `MyAnno` повторяющейся, ее нужно снабдить аннотацией `@Repeatable`, указывающей ее контейнерную аннотацию, которая называется `MyRepeatedAnnos`. В данной программе для доступа к повторяющимся аннотациям вызывается метод `getAnnotation()`, которому передается класс контейнерной аннотации, а не самой повторяющейся аннотации. Как следует из результата выполнения данной программы, повторяющиеся аннотации разделяются запятой. Они не возвращаются по отдельности.

Еще один способ получить повторяющиеся аннотации состоит в том, чтобы воспользоваться одним из новых методов, оперирующих непосредственно повторяющейся аннотацией и внедренных в интерфейс `AnnotatedElement` в версии JDK 8. Это методы `getAnnotationsByType()` и `getDeclaredAnnotationsByType()`. Ниже приведена общая форма первого из этих методов.

```

<T extends Annotation> T[ ] getAnnotationsByType(Class<T> тип_аннотации)

```

Этот метод возвращает массив аннотаций, имеющих тип `тип_аннотации` и связанных с вызывающим объектом. Если же аннотации отсутствуют, этот массив будет иметь нулевую длину. Ниже приведен пример применения метода `getAnnotationsByType()` для получения повторяющихся аннотаций `MyAnno`, представленных в предыдущем примере программы.

```

Annotation[] annos = m.getAnnotationsByType(MyAnno.class);
for(Annotation a : annos)
    System.out.println(a);

```

В данном примере тип повторяющейся аннотации `MyAnno` передается методу `getAnnotationsByType()`. Возвращаемый в итоге массив содержит все экземпляры аннотации `MyAnno`, связанные с методом `myMeth()` (в данном примере их два). Каждая повторяющаяся аннотация доступна в массиве по индексу. В данном случае каждая аннотация `MyAnno` выводится при выполнении цикла в стиле `for each`.

Некоторые ограничения на аннотации

Существует ряд ограничений, накладываемых на объявления аннотаций. Во-первых, одна аннотация не может наследовать другую. Во-вторых, все методы, объявленные в аннотации, должны быть без параметров. Кроме того, они должны возвращать один из перечисленных ниже типов:

- примитивный тип наподобие `int` или `double`;
- объект класса `String` или `Class`;
- перечислимый тип;
- тип другой аннотации;
- массив одного из предыдущих типов.

Аннотации не могут быть обобщенными. Иными словами, они не могут принимать параметры типа. (Обобщения рассматриваются в главе 14.) И наконец, при объявлении методов в аннотациях нельзя указывать оператор `throws`.

Ввод-вывод, апплеты и прочие вопросы

Эта глава посвящена двум наиболее важным пакетам в Java: `io` и `applet`. Пакет `io` поддерживает базовую систему ввода-вывода в Java, включая файловый ввод-вывод, а пакет `applet` — апплеты. Поддержка ввода-вывода и апплетов осуществляется библиотеками базового прикладного программного интерфейса (API), а не ключевыми словами языка Java. По этой причине углубленное обсуждение этих вопросов приведено в части II данной книги, где рассматриваются классы прикладного программного интерфейса Java API. А в этой главе представлены основы этих двух подсистем с целью показать, каким образом они интегрированы в язык Java и вписываются в общий контекст программирования на нем и его исполняющую среду. В главе рассматриваются также оператор `try` с ресурсами и недавно внедренные ключевые слова Java: `transient`, `volatile`, `instanceof`, `native`, `strictfp` и `assert`. И в завершение описаны статический импорт и особенности применения ключевого слова `this`, а также дано введение в компактные профили, внедренные в версии JDK 8.

Основы ввода-вывода

Читая предыдущие 12 глав этой книги, вы, вероятно, обратили внимание на то, что в приведенных до сих пор примерах программ было задействовано не так много операций ввода-вывода. По существу, никаких методов ввода-вывода, кроме `print()` и `println()`, в этих примерах не применялось. Причина этого проста: большинство реальных прикладных программ на Java не являются текстовыми консольными программами, а содержат графический пользовательский интерфейс (ГПИ), построенный на основе библиотек AWT, Swing или JavaFX для взаимодействия с пользователем, или же они являются веб-приложениями. Текстовые консольные программы отлично подходят в качестве учебных примеров, но они имеют весьма незначительное практическое применение. К тому же поддержка консольного ввода-вывода в Java ограничена и не очень удобна в употреблении — даже в простейших программах. Таким образом, текстовый консольный ввод-вывод не имеет большого практического значения для программирования на Java.

Несмотря на все сказанное выше, в Java предоставляется сильная и универсальная поддержка файлового и сетевого ввода-вывода. Система ввода-вывода в Java целостна и последовательна. Если усвоить ее основы, то овладеть всем остальным

будет очень просто. Здесь дается лишь общий обзор ввода-вывода, а подробное его описание приводится в главах 20 и 21.

Потоки ввода-вывода

В программах на Java создаются потоки ввода-вывода. *Поток ввода-вывода* — это абстракция, которая поставляет или потребляет информацию. Поток ввода-вывода связан с физическим устройством через систему ввода-вывода в Java. Все потоки ввода-вывода ведут себя одинаково, несмотря на отличия в конкретных физических устройствах, с которыми они связаны. Таким образом, одни и те же классы и методы ввода-вывода применимы к разнотипным устройствам. Это означает, что абстракция потока ввода может охватывать разные типы ввода: из файла на диске, клавиатуры или сетевого соединения. Аналогично поток вывода может обращаться к консоли, файлу на диске или сетевому соединению. Потоки ввода-вывода предоставляют ясный способ организации ввода-вывода, избавляя от необходимости разбираться в отличиях, например, клавиатуры от сети. В языке Java потоки ввода-вывода реализуются в пределах иерархии классов, определенных в пакете `java.io`.

На заметку! Помимо потокового ввода-вывода, определенного в пакете `java.io`, в Java предоставляется также буферный и канальный ввод-вывод, определенный в пакете `java.nio` и его подчиненных пакетах. Эти разновидности ввода-вывода рассматриваются в главе 21.

Потоки ввода-вывода байтов и символов

В Java определяются два вида потоков ввода-вывода: байтов и символов. *Потоки ввода-вывода байтов* предоставляют удобные средства для управления вводом и выводом отдельных байтов. Эти потоки используются, например, при чтении и записи двоичных данных. *Потоки ввода-вывода символов* предоставляют удобные средства управления вводом и выводом отдельных символов. С этой целью в них применяется кодировка в Юникоде, допускающая интернационализацию. Кроме того, потоки ввода-вывода символов оказываются порой более эффективными, чем потоки ввода-вывода байтов.

В первоначальной версии Java 1.0 потоки ввода-вывода символов отсутствовали, и поэтому весь ввод-вывод имел байтовую организацию. Потоки ввода-вывода символов были внедрены в версии Java 1.1, сделав не рекомендованным к употреблению некоторые классы и методы, поддерживавшие ввод-вывод с байтовой организацией. Устаревший код, в котором не применяются потоки ввода-вывода байтов, встречается все реже, но иногда он все еще применяется. Как правило, устаревший код приходится обновлять, если это возможно, чтобы воспользоваться преимуществами потоков ввода-вывода символов.

Следует также иметь в виду, что на самом низком уровне весь ввод-вывод по-прежнему имеет байтовую организацию. А потоки ввода-вывода символов лишь предоставляют удобные и эффективные средства для обращения с символами. В последующих разделах дается краткий обзор потоков ввода-вывода с байтовой и символьной организацией.

Классы потоков ввода-вывода байтов

Потоки ввода-вывода байтов определены в двух иерархиях классов. На вершине этих иерархий находятся абстрактные классы `InputStream` и `OutputStream`. У каждого из этих абстрактных классов имеется несколько конкретных подклассов, в которых учитываются отличия разных устройств, в числе файлов на диске, сетевых соединений и даже буферов памяти. Классы потоков ввода-вывода байтов из пакета `java.io` перечислены в табл. 13.1. Одни из этих классов описываются ниже, а другие — в части II данной книги. Не следует, однако, забывать о необходимости импортировать пакет `java.io`, чтобы воспользоваться классами потоков ввода-вывода.

Таблица 13.1. Классы потоков ввода-вывода байтов из пакета `java.io`

Класс потока ввода-вывода	Назначение
<code>BufferedInputStream</code>	Буферизированный поток ввода
<code>BufferedOutputStream</code>	Буферизированный поток вывода
<code>ByteArrayInputStream</code>	Поток ввода, читающий байты из массива
<code>ByteArrayOutputStream</code>	Поток вывода, записывающий байты в массив
<code>DataInputStream</code>	Поток ввода, содержащий методы для чтения данных стандартных типов, определенных в Java
<code>DataOutputStream</code>	Поток вывода, содержащий методы для записи данных стандартных типов, определенных в Java
<code>FileInputStream</code>	Поток ввода, читающий данные из файла
<code>FileOutputStream</code>	Поток вывода, записывающий данные в файл
<code>FilterInputStream</code>	Реализует абстрактный класс <code>InputStream</code>
<code>FilterOutputStream</code>	Реализует абстрактный класс <code>OutputStream</code>
<code>InputStream</code>	Абстрактный класс, описывающий поток ввода
<code>ObjectInputStream</code>	Поток ввода объектов
<code>ObjectOutputStream</code>	Поток вывода объектов
<code>OutputStream</code>	Абстрактный класс, описывающий поток вывода
<code>PipedInputStream</code>	Канал ввода
<code>PipedOutputStream</code>	Канал вывода
<code>PrintStream</code>	Поток вывода, содержащий методы <code>print()</code> и <code>println()</code>
<code>PushbackInputStream</code>	Поток ввода, поддерживающий возврат одного байта обратно в поток ввода
<code>SequenceInputStream</code>	Поток ввода, состоящий из двух и более потоков ввода, данные из которых читаются по очереди

В абстрактных классах `InputStream` и `OutputStream` определяется ряд ключевых методов, реализуемых в других классах потоков ввода-вывода. Наиболее важными среди них являются методы `read()` и `write()`, читающие и записыва-

ющие байты данных соответственно. Оба эти метода объявлены как абстрактные в классах `InputStream` и `OutputStream`, а в производных классах они переопределяются.

Классы потоков ввода-вывода символов

Потоки ввода-вывода символов также определены в двух иерархиях классов. На вершине этих иерархий находятся два абстрактных класса — `Reader` и `Writer`. Эти абстрактные классы управляют потоками символов в Юникоде. Для каждого из них в Java предусмотрен ряд конкретных подклассов. Классы потоков ввода-вывода символов перечислены в табл. 13.2.

Таблица 13.2. Классы потоков ввода-вывода символов из пакета `java.io`

Класс потока ввода-вывода	Назначение
<code>BufferedReader</code>	Буферизированный поток ввода символов
<code>BufferedWriter</code>	Буферизированный поток вывода символов
<code>CharArrayReader</code>	Поток ввода, читающий символы из массива
<code>CharArrayWriter</code>	Поток вывода, записывающий символы в массив
<code>FileReader</code>	Поток ввода, читающий символы из файла
<code>FileWriter</code>	Поток вывода, записывающий символы в файл
<code>FilterReader</code>	Фильтрованный поток чтения
<code>FilterWriter</code>	Фильтрованный поток записи
<code>InputStreamReader</code>	Поток ввода, преобразующий байты в символы
<code>LineNumberReader</code>	Поток ввода, подсчитывающий строки
<code>OutputStreamWriter</code>	Поток вывода, преобразующий символы в байты
<code>PipedReader</code>	Канал ввода
<code>PipedWriter</code>	Канал вывода
<code>PrintWriter</code>	Поток вывода, содержащий методы <code>print()</code> и <code>println()</code>
<code>PushbackReader</code>	Поток ввода, позволяющий возвращать символы обратно в поток ввода
<code>Reader</code>	Абстрактный класс, описывающий поток ввода символов
<code>StringReader</code>	Поток ввода, читающий символы из строки
<code>StringWriter</code>	Поток вывода, записывающий символы в строку
<code>Writer</code>	Абстрактный класс, описывающий поток вывода символов

В абстрактных классах `Reader` и `Writer` определяется ряд ключевых методов, реализуемых в других классах потоков ввода-вывода. Наиболее важными среди них являются методы `read()` и `write()`, читающие и записывающие байты данных соответственно. Оба эти метода объявлены как абстрактные в классах `Reader` и `Writer`, а в производных классах они переопределяются.

Предопределенные потоки ввода-вывода

Как вам должно быть уже известно, все программы на Java автоматически импортируют пакет `java.lang`. В этом пакете определен класс `System`, инкапсулирующий некоторые свойства исполняющей среды Java. Используя некоторые из его методов, можно, например, получить текущее время и настройки различных параметров, связанных с системой. Класс `System` содержит также три переменные предопределенных потоков ввода-вывода: `in`, `out` и `err`. Эти переменные объявлены в классе `System` как `public`, `static` и `final`. Это означает, что они могут быть использованы в любой другой части прикладной программы без обращения к конкретному объекту класса `System`.

Переменная `System.out` ссылается на стандартный поток вывода. По умолчанию это консоль. Переменная `System.in` ссылается на стандартный поток ввода, которым по умолчанию является клавиатура. А переменная `System.err` ссылается на стандартный поток вывода ошибок, которым по умолчанию также является консоль. Но эти стандартные потоки могут быть перенаправлены на любое совместимое устройство ввода-вывода.

Переменная `System.in` содержит объект типа `InputStream`, а переменные `System.out` и `System.err` содержат объекты типа `PrintStream`. Это потоки ввода-вывода байтов, хотя они, как правило, используются для чтения символов с консоли и записи символов на консоль. Как будет показано далее, их можно, если требуется, заключить в оболочки потоков ввода-вывода символов.

В примерах, приведенных в предыдущих главах, использовался стандартный поток вывода `System.out`. Аналогичным образом можно воспользоваться и стандартным потоком вывода `System.err`. Как поясняется в следующем разделе, пользоваться потоком `System.in` немного сложнее.

Чтение данных, вводимых с консоли

Организовать ввод данных с консоли в версии Java 1.0 можно было только с помощью потока ввода байтов. Такое по-прежнему возможно и теперь, но для коммерческого применения чтение данных, вводимых с консоли, предпочтительнее организовать с помощью потока ввода символов. Это значительно упрощает возможности интернационализации и сопровождения разрабатываемых программ.

В Java данные, вводимые с консоли, читаются из стандартного потока ввода `System.in`. Чтобы получить поток ввода символов, присоединив его к консоли, следует заключить стандартный поток ввода `System.in` в оболочку объекта класса `BufferedReader`, поддерживающего буферизованный поток ввода. Ниже приведен чаще всего используемый конструктор этого класса.

`BufferedReader (Reader поток_чтения_вводимых_данных)`

Здесь параметр `поток_чтения_вводимых_данных` обозначает поток, который связывается с создаваемым экземпляром класса `BufferedReader`. Класс `Reader` является абстрактным. Одним из производных от него конкретных подклассов является класс `InputStreamReader`, преобразующий байты в символы. Для получе-

ния объекта типа `InputStreamReader`, связанного со стандартным потоком ввода `System.in`, служит следующий конструктор:

```
InputStreamReader(InputStream поток_ввода)
```

Переменная `System.in` ссылается на объект класса `InputStream` и поэтому должна быть указана в качестве параметра *поток_ввода*. В конечном итоге получается приведенная ниже строка кода, в которой создается объект типа `BufferedReader`, связанный с клавиатурой. После выполнения этой строки кода переменная экземпляра `br` будет содержать поток ввода символов, связанный с консолью через стандартный поток ввода `System.in`.

```
BufferedReader br = new BufferedReader(new  
    InputStreamReader(System.in));
```

Чтение символов

Для чтения символа из потока ввода типа `BufferedReader` служит метод `read()`. Ниже показана версия метода `read()`, которая будет использоваться в приведенных далее примерах программ.

```
int read() throws IOException
```

Каждый раз, когда вызывается метод `read()`, он читает символ из потока ввода и возвращает его в виде целочисленного значения. По достижении конца потока возвращается значение `-1`. Как видите, метод `read()` может сгенерировать исключение типа `IOException`.

В приведенном ниже примере программы демонстрируется применение метода `read()` для чтения символов с консоли до тех пор, пока пользователь не введет символ "q". Следует заметить, что любые исключения, возникающие при вводе-выводе, просто генерируются в методе `main()`. Такой подход распространен при чтении данных с консоли в простых примерах программ, аналогичных представленным в этой книге, но в более сложных прикладных программах исключения можно обрабатывать явным образом.

```
// Использовать класс BufferedReader для чтения символов с консоли  
import java.io.*;
```

```
class BRRead {  
    public static void main(String args[]) throws IOException  
    {  
        char c;  
        BufferedReader br = new BufferedReader(new  
            InputStreamReader(System.in));  
        System.out.println("Введите символы, 'q' – для выхода.");  
        // читать символы  
        do {  
            c = (char) br.read();  
            System.out.println(c);  
        } while(c != 'q');  
    }  
}
```

Ниже приведен пример выполнения данной программы.

Введите символы, 'q' — для выхода.

```
123abcq
1
2
3
a
b
c
q
```

Результат выполнения данной программы может выглядеть не совсем так, как предполагалось, потому что стандартный поток ввода `System.in` по умолчанию является буферизованным построчно. Это означает, что никакие вводимые данные на самом деле не передаются программе до тех пор, пока не будет нажата клавиша <Enter>. Нетрудно догадаться, что эта особенность делает метод `read()` малоприменимым для организации ввода с консоли в диалоговом режиме.

Чтение символьных строк

Для чтения символьных строк с клавиатуры служит версия метода `readLine()`, который является членом класса `BufferedReader`. Его общая форма приведена ниже. Как видите, этот метод возвращает объект типа `String`.

String readLine() throws IOException

В приведенном ниже примере программы демонстрируется применение класса `BufferedReader` и метода `readLine()`. Эта программа читает и выводит текстовые строки текста до тех пор, пока не будет введено слово “стоп”.

// Чтение символьных строк с консоли средствами класса **BufferedReader**

```
import java.io.*;
```

```
class BRReadLines {
    public static void main(String args[]) throws IOException
    {
        // создать поток ввода типа BufferedReader,
        // используя стандартный поток ввода System.in
        BufferedReader br = new BufferedReader(new
            InputStreamReader(System.in));

        String str;
        System.out.println("Введите строки текста.");
        System.out.println("Введите 'стоп' для завершения.");
        do {
            str = br.readLine();
            System.out.println(str);
        } while(!str.equals("стоп"));
    }
}
```

В следующем примере программы демонстрируется простейший текстовый редактор. С этой целью сначала создается массив объектов типа `String`, а затем читаются текстовые строки, каждая из которых сохраняется в элементе массива. Чтение производится до 100 строк или до тех пор, пока не будет введено слово “стоп”. Для чтения данных с консоли применяется класс `BufferedReader`.


```
// Простейший текстовый редактор
import java.io.*;

class TinyEdit {
    public static void main(String args[]) throws IOException
    {
        // создать поток ввода типа BufferedReader,
        // используя стандартный поток ввода System.in
        BufferedReader br = new BufferedReader(new
            InputStreamReader(System.in));
        String str[] = new String[100];
        System.out.println("Введите строки текста.");
        System.out.println("Введите 'стоп' для завершения.");
        for(int i=0; i<100; i++) {
            str[i] = br.readLine();
            if(str[i].equals("стоп")) break;
        }
        System.out.println("\nСодержимое вашего файла:");
        // вывести текстовые строки
        for(int i=0; i<100; i++) {
            if(str[i].equals("стоп")) break;
            System.out.println(str[i]);
        }
    }
}
```

Ниже приведен пример выполнения данной программы.

```
Введите строки текста.
Введите 'стоп' для завершения.
Это строка один.
Это строка два.
Java упрощает работу со строками.
Просто создайте объекты типа String.
стоп
Содержимое вашего файла:
Это строка один.
Это строка два.
Java упрощает работу со строками.
Просто создайте объекты типа String.
```

Запись данных, выводимых на консоль

Вывод данных на консоль проще всего организовать с помощью упоминавшихся ранее методов `print()` и `println()`, которые применяются в большинстве примеров из этой книги. Эти методы определены в классе `PrintStream` (он является типом объекта, на который ссылается переменная `System.out`). Несмотря на то что стандартный поток `System.out` служит для вывода байтов, его можно вполне применять в простых программах для вывода данных. Тем не менее в следующем разделе описывается альтернативный ему поток вывода символов.

Класс `PrintStream` описывает поток вывода и является производным от класса `OutputStream`, поэтому в нем реализуется также низкоуровневый метод `write()`. Следовательно, метод `write()` можно применять для записи данных, выводимых на консоль. Ниже приведена простейшая форма метода `write()`, определенного в классе `PrintStream`.

```
void write(int байтовое_значение)
```

Этот метод записывает байт, передаваемый в качестве параметра *байтовое_значение*. Несмотря на то что параметр *байтовое_значение* объявлен как целочисленный, записываются только 8 его младших бит. Ниже приведен короткий пример, в котором метод `write()` применяется для вывода на экран буквы "А" с последующим переводом строки.

```
// Продемонстрировать применение метода System.out.write()
class WriteDemo {
    public static void main(String args[]) {
        int b;

        b = 'A';
        System.out.write(b);
        System.out.write('\n');
    }
}
```

Пользоваться методом `write()` для вывода данных на консоль приходится нечасто, хотя иногда это и удобно. Ведь намного проще применять для этой цели методы `print()` и `println()`.

Класс `PrintWriter`

Несмотря на то что стандартным потоком `System.out` вполне допустимо пользоваться для вывода данных на консоль, он все же подходит в большей степени для отладки или примеров программ, аналогичных представленным в данной книге. А для реальных программ рекомендуемым средством вывода данных на консоль служит поток записи, реализованный в классе `PrintWriter`, относящемся к категории символьных классов. Применение такого класса для консольного вывода упрощает интернационализацию прикладных программ.

В классе `PrintWriter` определяется несколько конструкторов. Ниже приведен один из тех конструкторов, которые применяются в рассматриваемых далее примерах.

```
PrintWriter(OutputStream поток_вывода, boolean очистка)
```

Здесь параметр *поток_вывода* обозначает объект типа `OutputStream`, а параметр *очистка* — очистку потока вывода всякий раз, когда вызывается (среди прочих) метод `println()`. Если параметр *очистка* принимает логическое значение `true`, то очистка потока вывода происходит автоматически, а иначе — вручную.

В классе `PrintWriter` поддерживаются методы `print()` и `println()`. Следовательно, их можно использовать таким же образом, как и в стандартном потоке вывода `System.out`. Если аргумент этих методов не относится к простому типу, то для объекта типа `PrintWriter` сначала вызывается метод `toString()`, а затем выводится результат.

Чтобы вывести данные на консоль, используя класс `PrintWriter`, следует указать стандартный поток `System.out` для вывода и его автоматическую очистку.

Например, в следующей строке кода создается объект типа `PrintWriter`, который связывается с консольным выводом:

```
PrintWriter pw = new PrintWriter(System.out, true);
```

В приведенном ниже примере программы демонстрируется применение класса `PrintWriter` для управления выводом данных на консоль.

```
// Продемонстрировать применение класса PrintWriter
import java.io.*;

public class PrintWriterDemo {
    public static void main(String args[]) {
        PrintWriter pw = new PrintWriter(System.out, true);

        pw.println("Это строка");
        int i = -7;
        pw.println(i);
        double d = 4.5e-7;
        pw.println(d);
    }
}
```

Ниже приведен результат, выводимый данной программой.

```
Это строка
-7
4.5E-7
```

Напомним, что стандартный поток `System.out` вполне пригоден для вывода простого текста на консоль на стадии овладения языком Java или отладки прикладных программ, тогда как класс `PrintWriter` упрощает интернационализацию реальных программ. Но поскольку применение класса `PrintWriter` не дает никаких преимуществ в примерах простых программ, то для вывода данных на консоль будет и далее применяться стандартный поток `System.out`.

Чтение и запись данных в файлы

В Java предоставляется немало классов и методов, позволяющих читать и записывать данные в файлы. Прежде всего, следует заметить, что тема ввода-вывода данных в файлы весьма обширна и подробно обсуждается в части II данной книги. А в этом разделе будут представлены основные способы чтения и записи данных в файл. И хотя для этой цели применяются потоки ввода-вывода байтов, упоминаемые здесь способы нетрудно приспособить и под потоки ввода-вывода символов.

Для ввода-вывода данных в файлы чаще всего применяются классы `FileInputStream` и `FileOutputStream`, которые создают потоки ввода-вывода байтов, связанные с файлами. Чтобы открыть файл для ввода-вывода данных, достаточно создать объект одного из этих классов, указав имя файла в качестве аргумента конструктора. У обоих классов имеются и дополнительные конструкторы, но в представленных далее примерах будут употребляться только следующие конструкторы:

```
FileInputStream(String имя_файла) throws FileNotFoundException
FileOutputStream(String имя_файла) throws FileNotFoundException
```

где параметр *имя_файла* обозначает имя того файла, который требуется открыть. Если при создании потока ввода файл не существует, то генерируется исключение типа `FileNotFoundException`. А если при создании потока вывода файл нельзя открыть или создать, то и в этом случае генерируется исключение типа `FileNotFoundException`. Класс исключения `FileNotFoundException` является производным от класса `IOException`. Когда файл открыт для вывода, любой файл, существовавший ранее под тем же самым именем, уничтожается.

На заметку! В тех случаях, когда присутствует диспетчер защиты, некоторые файловые классы, в том числе `FileInputStream` и `FileOutputStream`, генерируют исключение типа `SecurityException`, если при попытке открыть файл обнаруживается нарушение защиты. По умолчанию в прикладных программах, запускаемых на выполнение по команде `java`, диспетчер защиты не применяется. Поэтому в примерах, демонстрирующих организацию ввода-вывода в данной книге, вероятность генерирования исключения типа `SecurityException` не отслеживается. Но в других видах прикладных программ (например, апплетах) диспетчер защиты обычно применяется, и поэтому операции ввода-вывода данных в файлы вполне могут привести в них к исключению типа `SecurityException`. В таком случае следует организовать соответствующую обработку этого исключения.

Завершив работу с файлом, его нужно закрыть. Для этой цели служит метод `close()`, реализованный в классах `FileInputStream` и `FileOutputStream`:

```
void close() throws IOException
```

Закрытие файла высвобождает выделенные для него системные ресурсы, позволяя использовать их для других файлов. Неудачный исход закрытия файла может привести к «утечкам памяти», поскольку неиспользуемые ресурсы оперативной памяти останутся выделенными.

На заметку! Начиная с версии JDK 7 метод `close()` определяется в интерфейсе `AutoCloseable` из пакета `java.lang`. Интерфейс `AutoCloseable` наследует от интерфейса `Closeable` из пакета `java.io`. Оба интерфейса реализуются классами потоков ввода-вывода, включая классы `FileInputStream` и `FileOutputStream`.

Следует заметить, что имеются два основных способа закрытия файла, когда он больше не нужен. При первом, традиционном способе метод `close()` вызывается явно, когда файл больше не нужен. Именно такой способ применялся во всех версиях Java до JDK 7, и поэтому он присутствует во всем коде, написанном до версии JDK 7. А при втором способе применяется оператор `try` с ресурсами, внедренный в версии JDK 7. Этот оператор автоматически закрывает файл, когда он больше не нужен. И в этом случае отпадает потребность в явных вызовах метода `close()`. Но поскольку существует немалый объем кода, который написан до версии JDK 7 и все еще эксплуатируется и сопровождается, то по-прежнему важно знать и владеть традиционным способом закрытия файлов. Поэтому сначала будет рассмотрен именно этот способ, а новый, автоматизированный способ описывается в следующем разделе.

Чтобы прочитать данные из файла, можно воспользоваться версией метода `read()`, определенной в классе `FileInputStream`. Та его версия, которая применяется в представленных далее примерах, выглядит следующим образом:

```
int read() throws IOException
```

Всякий раз, когда вызывается метод `read()`, он выполняет чтение одного байта из файла и возвращает его в виде целочисленного значения. А если достигнут конец файла, то возвращается значение `-1`. Этот метод может сгенерировать исключение типа `IOException`.

В приведенном ниже примере программы метод `read()` применяется для ввода из файла, содержащего текст в коде ASCII, который затем выводится на экран. Имя файла указывается в качестве аргумента командной строки.

```
/* Отображение содержимого текстового файла.
   Чтобы воспользоваться этой программой, укажите
   имя файла, который требуется просмотреть.
   Например, чтобы просмотреть файл TEST.TXT,
   введите в командной строке следующую команду:

   java ShowFile TEST.TXT
*/

import java.io.*;

class ShowFile {
    public static void main(String args[])
    {
        int i;
        FileInputStream fin;

        // сначала убедиться, что имя файла указано
        if(args.length != 1) {
            System.out.println("Использование: ShowFile имя_файла");
            return;
        }

        // Попытка открыть файл
        try {
            fin = new FileInputStream(args[0]);
        } catch(FileNotFoundException e) {
            System.out.println("Невозможно открыть файл");
            return;
        }

        // Теперь файл открыт и готов к чтению.
        // Далее из него читаются символы до тех пор,
        // пока не встретится признак конца файла
        try {
            do {
                i = fin.read();
                if(i != -1) System.out.print((char) i);
            } while(i != -1);
        } catch(IOException e) {
            System.out.println("Ошибка чтения из файла");
        }

        // закрыть файл
        try {
```

```
        fin.close();
    } catch(IOException e) {
        System.out.println("Ошибка закрытия файла");
    }
}
```

Обратите внимание в данном примере программы на блок операторов `try/catch`, обрабатывающий ошибки, которые могут произойти при вводе-выводе. Каждая операция ввода-вывода проверяется на наличие исключений, и если исключение возникает, то оно обрабатывается. В простых программах или примерах кода исключения, возникающие в операциях ввода-вывода, как правило, генерируются в методе `main()`, как это делалось в предыдущих примерах ввода-вывода на консоль. Кроме того, в реальном прикладном коде иногда оказывается полезно, чтобы исключение распространялось в вызывающую часть программы, уведомляя ее о неудачном исходе операции ввода-вывода. Но ради демонстрации в большинстве примеров файлового ввода-вывода, представленных в данной книге, все исключения, возникающие в операциях ввода-вывода, обрабатываются явным образом.

В приведенном выше примере поток ввода закрывается после чтения из файла, но имеется и другая возможность, которая нередко оказывается удобной. Она подразумевает вызов метода `close()` из блока оператора `finally`. В таком случае все методы, получающие доступ к файлу, содержатся в блоке оператора `try`, тогда как блок оператора `finally` служит для закрытия файла. Таким образом, файл будет закрыт независимо от того, как завершится блок оператора `try`. Если обратиться к приведенному выше примеру, то в свете только что сказанного блок оператора `try`, где читаются данные из файла, может быть переписан следующим образом:

```
try {
    do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
    } while(i != -1);
} catch(IOException e) {
    System.out.println("Ошибка чтения из файла");
} finally {
    // закрыть файл при выходе из блока оператора try
    try {
        fin.close();
    } catch(IOException e) {
        System.out.println("Ошибка закрытия файла");
    }
}
```

В данном случае это не так важно. Тем не менее одно из преимуществ такого подхода состоит в том, что если выполнение кода, в котором происходит обращение к файлу, прекращается из-за каких-нибудь исключений, не связанных с операциями ввода-вывода, то файл все равно будет закрыт в блоке оператора `finally`.

Иногда проще заключить все части программы, открывающие файл и получающие доступ к его содержимому, в один блок оператора `try`, вместо того чтобы разделять его на два блока, а затем закрыть файл в блоке оператора `finally`. В качестве иллюстрации ниже показан другой способ написания программы `ShowFile` из предыдущего примера.

/* Отображение содержимого текстового файла.
 Чтобы воспользоваться этой программой, укажите
 имя файла, который требуется просмотреть.
 Например, чтобы просмотреть файл **TEST.TXT**,
 введите в командной строке следующую команду:

```
java ShowFile TEST.TXT
```

В этом варианте программы код, открывающий и получающий
 доступ к файлу, заключен в один блок оператора **try**.
 Файл закрывается в блоке оператора **finally**.

```
*/
import java.io.*;

class ShowFile {
    public static void main(String args[])
    {
        int i;
        FileInputStream fin = null;

        // сначала убедиться, что имя файла указано
        if(args.length != 1) {
            System.out.println("Использование: ShowFile имя_файла");
            return;
        }

        // В следующем коде сначала открывается файл, а затем
        // из него читаются символы до тех пор, пока не встретится
        // признак конца файла
        try {
            fin = new FileInputStream(args[0]);

            do {
                i = fin.read();
                if(i != -1) System.out.print((char) i);
            } while(i != -1);

        } catch(FileNotFoundException e) {
            System.out.println("Файл не найден.");
        } catch(IOException e) {
            System.out.println("Произошла ошибка ввода-вывода");
        } finally {
            // закрыть файл в любом случае
            try {
                if(fin != null) fin.close();
            } catch(IOException e) {
                System.out.println("Ошибка закрытия файла");
            }
        }
    }
}
```

Как видите, при таком подходе объект `fin` инициализируется пустым значением `null`. А в блоке оператора `finally` файл закрывается только в том случае, если объект `fin` не содержит пустое значение `null`. Такой подход оказывается работоспособным потому, что объект `fin` не будет содержать пустое значение `null` только том случае, если файл успешно открыт. Таким образом, метод `close()` не вызывается, если при открытии файла возникает исключение.

Последовательность операторов `try/catch` в приведенном выше примере можно сделать более краткой. Класс исключения `FileNotFoundException` является производным от класса `IOException`, и поэтому обрабатывать отдельно его исключение совсем не обязательно. В качестве примера ниже приведена переделанная последовательность операторов `try/catch` без перехвата исключения типа `FileNotFoundException`. В данном случае отображается стандартное сообщение об исключительной ситуации, описывающее возникшую ошибку.

```
try {
    fin = new FileInputStream(args[0]);

    do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
    } while(i != -1);

} catch(IOException e) {
    System.out.println("Ошибка ввода-вывода: " + e);
} finally {
    // закрыть файл в любом случае
    try {
        if(fin != null) fin.close();
    } catch(IOException e) {
        System.out.println("Ошибка закрытия файла");
    }
}
```

При таком подходе любая ошибка, в том числе и ошибка открытия файла, обрабатывается одним оператором `catch`. Благодаря своей краткости такой подход употребляется в большинстве примеров организации ввода-вывода, представленных в данной книге. Но этот подход не годится в тех случаях, когда требуется иначе отреагировать на неудачный исход открытия файла, например, когда пользователь может неправильно ввести имя файла. В таком случае можно было бы, например, запросить правильное имя файла, прежде чем переходить к блоку оператора `try`, где происходит обращение к файлу.

Для записи в файл можно воспользоваться методом `write()`, определенным в классе `FileOutputStream`. В своей простейшей форме этот метод выглядит следующим образом:

```
void write(int байтовое_значение) throws IOException
```

Этот метод записывает в файл байт, переданный ему в качестве параметра `байтовое_значение`. Несмотря на то что параметр `байтовое_значение` объявлен как целочисленный, в файл записываются только его младшие восемь бит. Если при записи возникает ошибка, генерируется исключение типа `IOException`. В следующем примере программы метод `write()` применяется для копирования файла:

```
/* Копирование файла.
   Чтобы воспользоваться этой программой, укажите
   имена исходного и целевого файлов.
   Например, чтобы скопировать файл FIRST.TXT в файл SECOND.TXT,
   введите в командной строке следующую команду:

   java CopyFile FIRST.TXT SECOND.TXT
*/
```



```

import java.io.*;

class CopyFile {
    public static void main(String args[]) throws IOException
    {
        int i;
        FileInputStream fin = null;
        FileOutputStream fout = null;

        // сначала убедиться, что указаны имена обоих файлов
        if(args.length != 2) {
            System.out.println("Использование: CopyFile откуда куда");
            return;
        }

        // копировать файл
        try {
            // попытаться открыть файлы
            fin = new FileInputStream(args[0]);
            fout = new FileOutputStream(args[1]);

            do {
                i = fin.read();
                if(i != -1) fout.write(i);
            } while(i != -1);

        } catch(IOException e) {
            System.out.println("Ошибка ввода-вывода: " + e);
        } finally {
            try {
                if(fin != null) fin.close();
            } catch(IOException e2) {
                System.out.println("Ошибка закрытия файла ввода");
            }
            try {
                if(fout != null) fout.close();
            } catch(IOException e2) {
                System.out.println("Ошибка закрытия файла вывода");
            }
        }
    }
}

```

Обратите внимание на то, что в данном примере программы при закрытии файлов используются два отдельных блока оператора `try`. Этим гарантируется, что оба файла будут закрыты, даже если при вызове метода `fin.close()` будет сгенерировано исключение. Обратите также внимание на то, что все потенциальные ошибки ввода-вывода обрабатываются в двух приведенных выше программах с помощью исключений, в отличие от других языков программирования, где для уведомления о файловых ошибках используются коды ошибок. Исключения в Java не только упрощают обращение с файлами, но и позволяют легко отличать условие достижения конца файла от файловых ошибок во время ввода.

Автоматическое закрытие файла

В примерах программ из предыдущего раздела метод `close()` вызывался явно, чтобы закрыть файл, как только он окажется ненужным. Как упоминалось ранее,

такой способ закрытия файлов применялся до версии JDK 7. И хотя он все еще допустим и применим, в версии JDK 7 появилась новая возможность, предлагающая иной способ управления такими ресурсами, как потоки ввода-вывода в файлы: автоматическое завершение процесса. Эту возможность иногда еще называют *автоматическим управлением ресурсами* (ARM), и основывается она на усовершенствованной версии оператора `try`. Главное преимущество автоматического управления ресурсами заключается в предотвращении ситуаций, когда файл (или другой ресурс) не освобождается по невнимательности, если он больше не нужен. Как пояснялось ранее, если забыть по какой-нибудь причине закрыть файл, это может привести к утечке памяти и другим осложнениям.

Как упоминалось выше, автоматическое управление ресурсами основывается на усовершенствованной форме оператора `try`. Ниже приведена его общая форма.

```
try (спецификация_ресурса) {  
    // использование ресурса  
}
```

Здесь *спецификация_ресурса* обозначает оператор, объявляющий и инициализирующий такой ресурс, как поток ввода-вывода в файл. Он состоит из объявления переменной, где переменная инициализируется ссылкой на управляемый объект. По завершении блока оператора `try` ресурс автоматически освобождается. Для файла это означает, что он автоматически закрывается, а следовательно, отпадает необходимость вызывать метод `close()` явно. Безусловно, эта новая форма оператора `try` может также включать в себя операторы `finally` и `catch`. Она называется оператором `try с ресурсами`.

Оператор `try с ресурсами` применяется только с теми ресурсами, которые реализуют интерфейс `AutoCloseable`, определенный в пакете `java.lang`. В этом интерфейсе определяется метод `close()`. Интерфейс `AutoCloseable` наследуется интерфейсом `Closeable` из пакета `java.io`. Оба интерфейса реализуются классами потоков ввода-вывода. Таким образом, оператор `try с ресурсами` может быть использован при обращении с потоками ввода-вывода, включая и потоки ввода-вывода в файлы. В качестве первого примера автоматического закрытия файла рассмотрим переделанную версию представленной ранее программы `ShowFile`.

```
/* В этой версии программы ShowFile оператор try с ресурсами  
   применяется для автоматического закрытия файла
```

```
Примечание: для выполнения этого кода требуется версия JDK 7  
*/
```

```
import java.io.*;  
  
class ShowFile {  
    public static void main(String args[])  
    {  
        int i;  
  
        // сначала убедиться, что имя файла указано  
        if(args.length != 1) {  
            System.out.println("Использование: ShowFile имя_файла");  
            return;  
        }  
    }  
}
```

```
// Ниже оператор try с ресурсами применяется
// сначала для открытия, а затем для автоматического
// закрытия файла по завершении блока этого оператора
try(FileInputStream fin = new FileInputStream(args[0])) {

    do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
    } while(i != -1);

} catch(FileNotFoundException e) {
    System.out.println("Файл не найден.");
} catch(IOException e) {
    System.out.println("Произошла ошибка ввода-вывода");
}
}
```

В приведенном выше примере программы обратите особое внимание на открытие файла в блоке оператора **try**:

```
try(FileInputStream fin = new FileInputStream(args[0])) {
```

Как видите, в той части оператора **try**, где указывается спецификация ресурса, объявляется экземпляр **fin** класса **FileInputStream**, которому затем присваивается ссылка на файл, открытый его конструктором. Таким образом, в данной версии программы переменная **fin** является локальной по отношению к блоку оператора **try**, в начале которого она создается. По завершении блока **try** поток ввода-вывода, связанный с переменной **fin**, автоматически закрывается в результате неявного вызова метода **close()**. Этот метод не нужно вызывать явно, а следовательно, исключается возможность просто забыть закрыть файл. В этом и состоит главное преимущество применения оператора **try** с ресурсами.

Однако ресурс, объявляемый в операторе **try**, неявно считается завершенным. Это означает, что присвоить ресурс после того, как он был создан, нельзя. Кроме того, область действия ресурса ограничивается пределами оператора **try** с ресурсами.

В одном операторе **try** можно организовать управление несколькими ресурсами. Для этого достаточно указать спецификацию каждого ресурса через точку с запятой. Примером тому служит приведенная ниже версия программы **CopyFile**, переделанная таким образом, чтобы использовать один оператор **try** с ресурсами для управления переменными **fin** и **fout**.

```
/* Версия программы CopyFile, использующая оператор try с ресурсами.
   Она демонстрирует управление двумя ресурсами (в данном случае –
   файлами) в одном операторе try
*/

import java.io.*;

class CopyFile {
    public static void main(String args[]) throws IOException
    {
        int i;

        // сначала убедиться, что заданы оба файла
        if(args.length != 2) {
            System.out.println("Использование: CopyFile откуда куда");
        }
    }
}
```

```
        return;
    }

    // открыть два файла и управлять ними в операторе try
    try (FileInputStream fin = new FileInputStream(args[0]);
        FileOutputStream fout = new FileOutputStream(args[1]))
    {
        do {
            i = fin.read();
            if(i != -1) fout.write(i);
        } while(i != -1);

    } catch(IOException e) {
        System.out.println("Ошибка ввода-вывода: " + e);
    }
}
```

Обратите внимание на то, как файлы ввода и вывода открываются в блоке оператора `try`.

```
try (FileInputStream fin = new FileInputStream(args[0]);
    FileOutputStream fout = new FileOutputStream(args[1]))
{
    // ...
}
```

По завершении этого блока оператора `try` будут закрыты оба ресурса в переменных `fin` и `fout`. Если сравнить эту версию программы с предыдущей, то можно заметить, что она значительно короче. Возможность упростить исходный код является дополнительным преимуществом автоматического управления ресурсами.

У оператора `try` с ресурсами имеется еще одна особенность, о которой стоит упомянуть. В общем, когда выполняется блок оператора `try`, существует вероятность того, что исключение, возникающее в блоке оператора `try`, приведет к другому исключению, которое произойдет в тот момент, когда ресурс закрывается в блоке оператора `finally`. Если это “обычный” оператор `try`, то первоначальное исключение теряется, будучи вытесненным вторым исключением. А если используется оператор `try` с ресурсами, то второе исключение *подавляется*, но не теряется. Вместо этого оно добавляется в список подавленных исключений, связанных с первым исключением. Доступ к списку подавленных исключений может быть получен с помощью метода `getSuppressed()`, определенного в классе `Throwable`.

Благодаря упомянутым выше преимуществам оператор `try` с ресурсами применяется во многих, но не во всех примерах программ, представленных в данной книге. В некоторых примерах по-прежнему применяется традиционный способ закрытия ресурсов. И на то есть несколько причин. Во-первых, существует немалый объем широко распространенного и эксплуатируемого кода, в котором применяется традиционный способ и с которым хорошо знакомы все программирующие на Java. Во-вторых, не все разрабатываемые проекты будут немедленно переведены на новую версию JDK. Некоторые программисты, вероятно, какое-то время продолжат работать в среде, предшествующей версии JDK 7, где рассматриваемая здесь улучшенная форма оператора `try` недоступна. И наконец, возможны случаи, когда явное закрытие ресурса вручную оказывается лучше, чем автоматическое. По этим причинам в некоторых примерах из этой книги будет и далее применяться

ся традиционный способ явного вызова метода `close()` для закрытия ресурсов вручную. Помимо того, что эти примеры программ демонстрируют традиционный способ управления ресурсами, они могут быть откомпилированы и запущены всеми читателями данной книги во всех системах, которыми они пользуются.

Помните! Для целей демонстрации в некоторых примерах программ из данной книги намеренно применяется традиционный способ закрытия файлов, широко распространенный в унаследованном коде. Но при написании нового кода рекомендуется применять новый способ автоматического управления ресурсами, который поддерживается в только что описанном операторе `try` с ресурсами.

Основы создания апплетов

Во всех предыдущих примерах демонстрировались программы, относящиеся к категории консольных прикладных программ на Java. Но это лишь одна из категорий прикладных программ на Java. К другой категории относятся апплеты. Как упоминалось в главе 1, *апплет* — это небольшая прикладная программа, находящаяся на веб-сервере, откуда она загружается, автоматически устанавливается и запускается как составная часть веб-документа. Как только апплет появится у клиента, он получает ограниченный доступ к ресурсам, чтобы предоставить графический пользовательский интерфейс (ГПИ) и выполнить различные вычисления, не подвергая клиента риску вирусной атаки или нарушения целостности его данных.

Начнем рассмотрение апплетов с простейшего примера, приведенного ниже.

```
import java.awt.*;
import java.applet.*;

public class SimpleApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("Простейший апплет", 20, 20);
    }
}
```

Этот апплет начинается с двух операторов `import`. Первый из них импортирует классы библиотеки Abstract Window Toolkit (AWT). Апплеты взаимодействуют с пользователем через библиотеку AWT, а не через классы консольного ввода-вывода. Библиотека AWT поддерживает элементарный оконный ГПИ и служит здесь в качестве введения в программирование апплетов. Нетрудно догадаться, что библиотека AWT значительно крупнее и сложнее, и полное обсуждение ее возможностей занимает несколько глав в части II данной книги. Правда, приведенный выше простой апплет весьма ограниченно использует возможности библиотеки AWT. (Для поддержки ГПИ в апплетах можно также пользоваться библиотекой Swing, но такой подход рассматривается далее в книге.) Второй оператор `import` импортирует пакет `applet`, в котором находится класс `Applet`. Каждый создаваемый апплет должен быть подклассом, прямо или косвенно производным от класса `Applet`.

В следующей строке кода из рассматриваемого здесь примера простейшего апплета объявляется класс `SimpleApplet`. Этот класс должен быть объявлен открытым (`public`), чтобы быть доступным для кода за пределами апплета.

В классе `SimpleApplet` объявляется метод `paint()`, который определен в библиотеке `AWT` и должен быть переопределен в апплете. Метод `paint()` вызывается всякий раз, когда апплет должен перерисовать результат, выводимый графическим способом. Такая ситуация может возникнуть по ряду причин. Например, окно, в котором запущен апплет, может быть перекрыто другим окном, а затем вновь открыто или же свернуто, а затем развернуто. Метод `paint()` вызывается в том случае, когда апплет начинает свое выполнение. Независимо от конкретной причины, всякий раз, когда апплет должен перерисовать выводимый графическим способом результат, вызывается метод `paint()`, принимающий единственный параметр типа `Graphics`. Этот параметр содержит графический контекст, описывающий графическую среду, в которой действует апплет. Графический контекст используется всякий раз, когда из апплета требуется вывести результат.

В методе `paint()` вызывается метод `drawString()` из класса `Graphics`. Этот метод выводит строку на позиции, задаваемой координатами `X,Y`. Он имеет следующую общую форму:

```
void drawString(String сообщение, int x, int y)
```

где параметр *сообщение* обозначает символьную строку, которая должна быть выведена начиная с позиции, определяемой координатами `x,y`. В Java верхний левый угол окна имеет координаты `0,0`. В результате вызова метода `drawString()` в апплете выводится символьная строка "Простейший апплет", начиная с позиции, определяемой координатами `20,20`.

Обратите внимание на то, что в апплете отсутствует метод `main()`. В отличие от обычных программ на Java, выполнение апплета не начинается с метода `main()`. На самом деле этого метода нет у большинства апплетов. Вместо этого выполнение апплета начинается, когда имя его класса передается средству просмотра апплетов или сетевому браузеру.

После ввода исходного кода апплета `SimpleApplet` его компиляция выполняется таким же образом, как и обычных программ. Но запуск апплета `SimpleApplet` осуществляется иначе. По существу, запустить апплет можно двумя способами:

- в совместимом с Java браузере;
- средством просмотра апплетов, например, стандартным инструментальным средством **appletviewer**. Это средство выполняет апплет в своем окне. Обычно это самый быстрый и простой способ проверки работоспособности апплета.

Каждый из этих способов запуска апплетов подробно описывается ниже.

Для того чтобы выполнить апплет в веб-браузере, достаточно, в частности, создать короткий HTML-файл, который должен содержать соответствующий дескриптор. В настоящее время компания Oracle рекомендует использовать для этой цели дескриптор `APPLET`, хотя может быть также использован дескриптор `OBJECT`. Подробнее о методиках развертывания апплетов речь пойдет в главе 23. В качестве примера ниже приведен HTML-файл с дескриптором `APPLET` для запуска апплета `SimpleApplet`.

```
<applet code="SimpleApplet" width=200 height=60>
</applet>
```

Атрибуты `width` и `height` разметки этого HTML-файл обозначают размеры области отображения, используемой апплетом. (У дескриптора `APPLET` имеются и другие параметры, более подробно рассматриваемые в части II.) Создав рассматриваемый здесь HTML-файл, следует запустить браузер, а затем загрузить в нем этот файл, в результате чего и будет выполнен апплет `SimpleApplet`.

На заметку! Начиная с обновления 21 версии Java 7, апплеты в Java должны подписываться во избежание предупреждений о нарушении защиты при их выполнении в браузере, хотя иногда требуется предотвратить выполнение апплета. К этим переменам особенно чувствительны апплеты, хранящиеся в локальной файловой системе, в том числе и те, что получаются в результате компиляции примеров из данной книги. Так, для выполнения локального апплета в браузере, скорее всего, придется настроить параметры защиты на панели управления Java (Java Control Panel). На момент написания этой книги в компании Oracle рекомендовали пользоваться не локальными апплетами, а теми, что выполняются через веб-сервер. А в будущем ожидается, что выполнение неподписанных локальных апплетов будет блокироваться. В конечном итоге подписание апплетов, распространяемых через Интернет, станет обязательным. И хотя принципы и методики подписания апплетов (и других видов прикладных программ на Java) выходят за рамки рассмотрения этой книги, на веб-сайте компании Oracle можно найти немало подробных сведений по данному вопросу. И, наконец, чтобы опробовать представленные здесь примеры апплетов, проще всего воспользоваться упомянутым ранее средством просмотра апплетов `appletviewer`.

Чтобы опробовать апплет `SimpleApplet` в средстве просмотра апплетов, следует также выполнить приведенный выше HTML-файл. Так, если этот HTML-файл называется `RunApp.html`, то для его запуска на выполнение достаточно ввести в командной строке следующую команду:

```
C:\>appletviewer RunApp.html
```

Но существует более удобный способ, ускоряющий проверку апплетов. Для этого достаточно ввести в начале исходного файла апплета комментарий, содержащий дескриптор `APPLET`. В итоге исходный код апплета будет документирован прототипом необходимых операторов HTML, и скомпилированный апплет можно будет проверить, запуская средство просмотра апплетов вместе с исходным файлом апплета. Если применяется именно такой способ проверки апплетов, то исходный файл апплета `SimpleApplet` должен выглядеть следующим образом:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleApplet" width=200 height=60>
</applet>
*/

public class SimpleApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("Простейший апплет", 20, 20);
    }
}
```

Подобным способом можно очень быстро проходить стадии разработки апплетов, если выполнить следующие действия.

- Отредактировать файл исходного кода апплета.
- Откомпилировать исходный код апплета.
- Запустить средство просмотра апплетов вместе исходным файлом апплета. Средство просмотра апплетов обнаружит дескриптор `APPLET` в комментарии к исходному коду апплета и запустит его на выполнение.

Окно апплета `SimpleApplet`, отображаемое средством просмотра апплетов, приведено на рис. 13.1. Разумеется, фрейм средства просмотра апплетов может отличаться своим внешним видом в зависимости от конкретной исполняющей среды. Поэтому моментальные снимки экранов, приведенные в данной книге, отражают разные исполняющие среды.

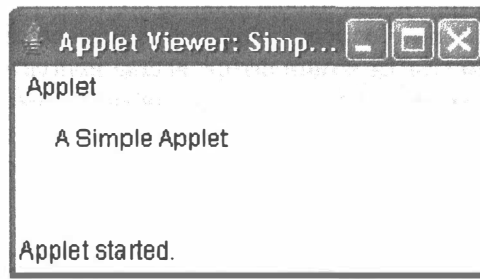


Рис. 13.1. Окно апплета `SimpleApplet`, отображаемое средством просмотра апплетов

Вопросы создания апплетов будут обсуждаться далее в данной книге, а до тех пор укажем главные их особенности, о которых нужно знать теперь.

- Апплеты не нуждаются в методе `main()`.
- Апплеты должны запускаться из средства просмотра апплетов или совместимого с Java веб-браузера.
- Пользовательский ввод-вывод в апплетах не организуется с помощью классов потоков ввода-вывода. Вместо этого в апплетах применяется ГПИ, предоставляемый соответствующими библиотеками.

Модификаторы доступа `transient` и `volatile`

В языке Java определяются два интересных модификатора доступа: `transient` и `volatile`. Эти модификаторы предназначены для особых случаев. Когда переменная-экземпляр объявлена как `transient`, ее значение не должно сохраняться, когда сохраняется объект:

```
class T {  
    transient int a; // не сохранится  
    int b;           // сохранится  
}
```


Если в данном примере кода объект типа `T` записывается в область постоянного хранения, то содержимое переменной `a` не должно сохраняться, тогда как содержимое переменной `b` должно быть сохранено.

Модификатор доступа `volatile` сообщает компилятору, что модифицируемая им переменная может быть неожиданно изменена в других частях программы. Одна из таких ситуаций возникает в многопоточных программах, где иногда у двух или более потоков исполнения имеется совместный доступ к одной и той же переменной. Из соображений эффективности, в каждом потоке может храниться своя закрытая копия этой переменной. Настоящая (или *главная*) копия переменной обновляется в разные моменты, например, при входе в синхронизированный метод. Такой подход вполне работоспособен, но не всегда оказывается достаточно эффективным. Иногда требуется, чтобы главная копия переменной постоянно отражала ее текущее состояние. И для этого достаточно объявить переменную как `volatile`, сообщив тем самым компилятору всегда использовать главную копию этой переменной (или хотя бы поддерживать любые закрытые ее копии обновляемыми по главной копии, и наоборот). Кроме того, доступ к главной копии переменной должен осуществляться в том же порядке, что и к любой закрытой копии.

Применение оператора `instanceof`

Иногда тип объекта полезно выяснить во время выполнения. Например, в одном потоке исполнения объекты разных типов могут формироваться, а в другом потоке исполнения — использоваться. В таком случае удобно выяснить тип каждого объекта, получаемого в обрабатывающем потоке исполнения. Тип объекта во время выполнения не менее важно выяснить и в том случае, когда требуется приведение типов. В Java неправильное приведение типов влечет за собой появление ошибки во время выполнения. Большинство ошибок приведения типов может быть выявлено на стадии компиляции. Но приведение типов в пределах иерархии классов может стать причиной ошибок, которые обнаруживаются только во время выполнения. Например, суперкласс `A` может порождать два подкласса: `B` и `C`. Следовательно, приведение объекта класса `B` или `C` к типу `A` вполне допустимо, но приведение объекта класса `B` к типу `C` (и наоборот) — неверно. А поскольку объект типа `A` может ссылаться на объекты типа `B` и `C`, то как во время выполнения узнать, на какой именно тип делается ссылка перед тем, как выполнить приведение к типу `C`? Это может быть объект типа `A`, `B` или `C`. Если это объект типа `B`, то во время выполнения будет сгенерировано исключение. Для разрешения этого вопроса в Java предоставляется оператор времени выполнения `instanceof`, который имеет следующую общую форму:

ссылка_на_объект `instanceof` **тип**

где *ссылка_на_объект* обозначает ссылку на экземпляр класса, а *тип* — конкретный тип этого класса. Если *ссылка_на_объект* относится к указанному типу или может быть приведена к нему, то вычисление оператора `instanceof` дает в итоге логическое значение `true`, а иначе — логическое значение `false`. Таким образом, оператор `instanceof` — это средство, с помощью которого программа может по-

лучить сведения об объекте во время выполнения. В следующем примере программы демонстрируется применение оператора `instanceof`:

```
// Продемонстрировать применение оператора instanceof
class A {
    int i, j;
}

class B {
    int i, j;
}

class C extends A {
    int k;
}

class D extends A {
    int k;
}

class InstanceOf {
    public static void main(String args[]) {
        A a = new A();
        B b = new B();
        C c = new C();
        D d = new D();
        if(a instanceof A)
            System.out.println("a является экземпляром A");
        if(b instanceof B)
            System.out.println("b является экземпляром B");
        if(c instanceof C)
            System.out.println("c является экземпляром C");
        if(c instanceof A)
            System.out.println("c можно привести к типу A");

        if(a instanceof C)
            System.out.println("a можно привести к типу C");

        System.out.println();

        // сравнить с порожденными типами
        A ob;

        ob = d; // ссылка на объект d
        System.out.println("ob теперь ссылается на d");
        if(ob instanceof D)
            System.out.println("ob является экземпляром D");

        System.out.println();

        ob = c; // ссылка на c
        System.out.println("ob теперь ссылается на c");

        if(ob instanceof D)
            System.out.println("ob можно привести к типу D");
        else
            System.out.println("ob нельзя привести к типу D");

        if(ob instanceof A)
            System.out.println("ob можно привести к типу A");

        System.out.println();
    }
}
```

```
// все объекты могут быть приведены к типу Object
if(a instanceof Object)
    System.out.println("a можно привести к типу Object");
if(b instanceof Object)
    System.out.println("b можно привести к типу Object");
if(c instanceof Object)
    System.out.println("c можно привести к типу к Object");
if(d instanceof Object)
    System.out.println("d можно привести к типу к Object");
}
}
```

Эта программа выводит следующий результат:

```
a является экземпляром A
b является экземпляром B
c является экземпляром C
с можно привести к типу A
```

```
ob теперь ссылается на d
ob является экземпляром D
```

```
ob теперь ссылается на c
ob нельзя привести к типу D
ob можно привести к типу A
```

```
a можно привести к типу Object
b можно привести к типу Object
c можно привести к типу Object
d можно привести к типу Object
```

Большинство программ не нуждается в операторе `instanceof`, поскольку типы объектов обычно известны заранее. Но этот оператор может пригодиться при разработке обобщенных процедур, оперирующих объектами из сложной иерархии классов.

Модификатор доступа `strictfp`

С появлением версии Java 2 модель вычислений с плавающей точкой стала чуть менее строгой. В частности, эта модель не требует теперь округления некоторых промежуточных результатов вычислений. В ряде случаев это предотвращает переполнение. Объявляя класс, метод или интерфейс с модификатором доступа `strictfp`, можно гарантировать, что вычисления с плавающей точкой будут выполняться таким же образом, как и в первых версиях Java. Если класс объявляется с модификатором доступа `strictfp`, все его методы автоматически модифицируются как `strictfp`.

Например, в приведенной ниже строке кода компилятору Java сообщается, что во всех методах, определенных в классе `MyClass`, следует использовать исходную модель вычислений с плавающей точкой. Откровенно говоря, большинству программистов вряд ли понадобится модификатор доступа `strictfp`, поскольку он касается лишь небольшой категории задач.

```
strictfp class MyClass { //...
```

Платформенно-ориентированные методы

Иногда, хотя и редко, возникает потребность вызвать подпрограмму, написанную на другом языке, а не на Java. Как правило, такая подпрограмма существует в виде исполняемого кода для ЦП и той среды, в которой приходится работать, т.е. в виде платформенно-ориентированного кода. Такую подпрограмму, возможно, потребуется вызвать для повышения скорости выполнения. С другой стороны, может возникнуть потребность работать со специализированной сторонней библиотекой, например, с пакетом статистических расчетов. Но поскольку программы на Java компилируются в байт-код, который затем интерпретируется (или динамически компилируется во время выполнения) исполняющей системой Java, то вызвать подпрограмму в платформенно-ориентированном коде из программы на Java, на первый взгляд, невозможно. К счастью, этот вывод оказывается ложным. Для объявления платформенно-ориентированных методов в Java предусмотрено ключевое слово `native`. Однажды объявленные как `native`, эти методы могут быть вызваны из прикладной программы на Java таким же образом, как и вызывается любой другой метод.

Чтобы объявить платформенно-ориентированный метод, его имя следует предварить модификатором доступа `native`, но не определять тело метода, как показано ниже.

```
public native int meth() ;
```

Объявив платформенно-ориентированный метод, нужно написать его и предпринять ряд относительно сложных шагов, чтобы связать его с кодом Java. Большинство платформенно-ориентированных методов пишутся на C. Механизм интеграции кода на C и программы на Java называется *интерфейсом JNI* (Java Native Interface). Подробное описание интерфейса JNI выходит за рамки данной книги, но предложенное ниже краткое описание дает достаточное представление для простого применения этого интерфейса.

На заметку! Конкретные действия, которые следует предпринять, зависят от применяемой среды Java, а также от языка, используемого для реализации платформенно-ориентированных методов. Приведенный ниже пример опирается на среду Windows для реализации платформенно-ориентированного метода на языке C. Кроме того, в рассматриваемом здесь примере применяется динамически подключаемая библиотека, но в версии JDK 8 появилась возможность создавать и статически подключаемую библиотеку.

Самый простой способ понять процесс — рассмотреть его на конкретном примере. Ниже представлена короткая программа, в которой используется платформенно-ориентированный метод `test()`. Можете ввести ее исходный текст.

```
// Простой пример применения платформенно-ориентированного метода
public class NativeDemo {
    int i;
    public static void main(String args[]) {
        NativeDemo ob = new NativeDemo();

        ob.i = 10;
```

```

        System.out.println("Содержимое переменной ob.i перед вызовом
                           платформенно-ориентированного метода:" + ob.i);
        ob.test(); // вызвать платформенно-ориентированный метод
        System.out.println("Содержимое переменной ob.i после вызова
                           платформенно-ориентированного метода:" + ob.i);
    }

    // объявить платформенно-ориентированный метод
    public native void test() ;

    // загрузить библиотеку DLL, содержащую статический метод
    static {
        System.loadLibrary("NativeDemo");
    }
}

```

Обратите внимание на то, что метод `test()` объявлен как `native` и не имеет тела. Это метод, который будет реализован далее на языке C. Обратите также внимание на блок оператора `static`. Как пояснялось ранее, блок, объявленный как `static`, выполняется только один раз при запуске программы, а точнее говоря, при первой загрузке ее класса. В данном случае он служит для загрузки динамически подключаемой библиотеки, которая содержит реализацию метода `test()`. (Ниже будет показано, как создать такую библиотеку.)

Динамически подключаемая библиотека загружается методом `loadLibrary()`, входящим в состав класса `System`. Общая форма этого метода приведена ниже.

```
static void loadLibrary(String имя_файла)
```

Здесь параметр *имя_файла* обозначает символьную строку, в которой задается имя файла, содержащего библиотеку. Для среды Windows предполагается, что этот файл имеет расширение `.DLL`.

Введя исходный текст рассматриваемой здесь программы, скомпилируйте ее, чтобы получить файл `NativeDemo.class`. Затем воспользуйтесь инструментальным средством **javah.exe**, чтобы создать заголовочный файл `NativeDemo.h` (это инструментальное средство входит в состав JDK). Файл `NativeDemo.h` вам предстоит включить в свою реализацию метода `test()`. Чтобы получить файл `NativeDemo.h`, выполните следующую команду:

```
javah -jni NativeDemo
```

Эта команда создает файл `NativeDemo.h`, который должен быть включен в исходный файл C, реализующий метод `test()`. Ниже приведен результат выполнения приведенной выше команды.

```

/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class NativeDemo */

#ifndef _Included_NativeDemo
#define _Included_NativeDemo
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class: NativeDemo
 * Method: test

```

```

* Signature: ()V
*/
JNIEXPORT void JNICALL Java_NativeDemo_test(JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif

```

Обратите особое внимание на следующую строку кода, которая определяет прототип создаваемой вами функции `test()`:

```
JNIEXPORT void JNICALL Java_NativeDemo_test(JNIEnv *, jobject);
```

Следует иметь в виду, что функция будет называться `Java_NativeDemo_test()`. Именно так и следует называть реализуемую вами платформенно-ориентированную функцию. Это означает, что вместо функции `test()` вы создаете на C функцию `Java_NativeDemo_test()`. Часть `NativeDemo` в префиксе добавляется, поскольку она обозначает, что метод `test()` является членом класса `NativeDemo`. Не следует забывать, что в другом классе можно объявить свой метод `test()`, совершенно не похожий на тот, который объявлен в классе `NativeDemo`. Поэтому для различения разных версий этого метода в его имя в качестве префикса включается имя класса. Как правило, платформенно-ориентированным функциям присваивается имя, содержащее в качестве префикса имя класса, в котором он объявлен.

Создав требующийся заголовочный файл, можете написать свою реализацию метода `test()` и сохранить ее в файле `NativeDemo.c`. Ниже приведен пример реализации этого метода на C.

```

/* Этот файл содержит версию метода test() на C.*/

#include <jni.h>
#include "NativeDemo.h"
#include <stdio.h>

JNIEXPORT void JNICALL Java_NativeDemo_test(JNIEnv *env, jobject obj)
{
    jclass cls;
    jfieldID fid;
    jint i;

    printf("Запуск платформенно-ориентированного метода.\n");
    cls = (*env)->GetObjectClass(env, obj);
    fid = (*env)->GetFieldID(env, cls, "i", "I");

    if(fid == 0) {
        printf("Невозможно получить поле id.\n");
        return;
    }
    i = (*env)->GetIntField(env, obj, fid);
    printf("i = %d\n", i);
    (*env)->SetIntField(env, obj, fid, 2*i);
    printf("Завершение платформенно-ориентированного метода.\n");
}

```

Как видите, этот файл включает в себя заголовок `jni.h`, содержащий интерфейсную информацию. Этот файл поставляется вместе с компилятором Java. Напомним, что заголовочный файл `NativeDemo.h` создан ранее по команде `javah`.

В этой функции метод `GetObjectClass()` служит для получения структуры `C`, содержащей данные о классе `NativeDemo`. Метод `GetFieldID()` возвращает структуру `C`, содержащую данные о поле `i` из этого класса. А метод `GetIntField()` извлекает исходное значение этого поля и сохраняет его обновленное значение (дополнительные методы, управляющие другими типами данных, см. в файле `jni.h`).

Создав исходный файл `NativeDemo.c`, его следует скомпилировать, а затем сформировать динамически подключаемую библиотеку. Для того чтобы сделать это с помощью компилятора Microsoft C/C++, введите в командной строке следующую команду, дополнительно указав, если потребуется, путь к файлу `jni.h` и его подчиненному файлу `jni_md.h`:

```
cl /LD NativeDemo.c
```

По этой команде будет создан файл `NativeDemo.dll`. И только после этого можете запустить программу на Java, которая выдаст следующий результат:

```
Содержимое переменной ob.i перед вызовом
платформенно-ориентированного метода: 10
Запуск платформенно-ориентированного метода.
i = 10
Завершение платформенно-ориентированного метода.
Содержимое переменной ob.i после вызова
платформенно-ориентированного метода: 20
```

Трудности, связанные с платформенно-ориентированными методами

Платформенно-ориентированные методы выглядят многообещающе, поскольку позволяют получить доступ к существующей базе библиотечных подпрограмм, а также надеяться на высокую скорость работы программ. Но с этими методами связаны две значительные трудности.

- **Потенциальный риск нарушения безопасности.** Платформенно-ориентированный метод выполняет конкретный машинный код, поэтому он может получить доступ к любой части системы. Это означает, что платформенно-ориентированный код не ограничивается только исполняющей средой Java, а следовательно, он несет в себе, например, угрозу заражения вирусами. Именно по этой причине в апплетах нельзя использовать платформенно-ориентированные методы. Кроме того, загрузка динамически подключаемых библиотек может быть ограничена и произведена только с разрешения диспетчера защиты.
- **Потеря переносимости.** Платформенно-ориентированный код содержится в динамически подключаемой библиотеке, поэтому он должен быть представлен на той машине, которая выполняет программу на Java. Более того, каждый платформенно-ориентированный метод зависит от конкретного процессора и операционной системы, а следовательно, каждая динамически подключаемая библиотека оказывается непереносимой. Таким

образом, прикладная программа на Java, в которой применяются платформенно-ориентированные методы, может выполняться только на той машине, на которой установлена совместимая динамически подключаемая библиотека.

Применение платформенно-ориентированных методов должно быть ограничено, поскольку они делают прикладную программу на Java непереносимой и представляют существенный риск нарушения безопасности.

Применение ключевого слова `assert`

Еще одним относительно новым дополнением языка Java является ключевое слово `assert`. Оно используется на стадии разработки программ для создания так называемых *утверждений* — условий, которые должны быть истинными во время выполнения программы. Например, в программе может быть метод, который всегда возвращает положительное целое значение. Его можно проверить утверждением, что возвращаемое значение больше нуля, используя оператор `assert`. Если во время выполнения программы условие оказывается истинным, то никаких действий больше не выполняется. Но если условие окажется ложным, то генерируется исключение типа `AssertionError`. Утверждения часто применяются с целью проверить, что некоторое ожидаемое условие действительно выполняется. В коде окончательной версии программы утверждения, как правило, отсутствуют.

Ключевое слово `assert` имеет две формы. Первая его форма выглядит следующим образом:

```
assert условие;
```

где *условие* обозначает выражение, в результате вычисления которого должно быть получено логическое значение. Если это логическое значение `true`, то утверждение истинно и никаких действий больше не выполняется. Если же вычисление условия дает логическое значение `false`, то утверждение не подтверждается и по умолчанию генерируется объект исключения типа `AssertionError`.

Ниже приведена вторая форма оператора `assert`.

```
assert условие: выражение;
```

В этой версии *выражение* обозначает значение, которое передается конструктору класса исключения `AssertionError`. Это значение преобразуется в строковую форму и выводится, если утверждение не подтверждается. Как правило, в качестве *выражения* задается символьная строка, но, в общем, разрешается любое выражение, кроме типа `void`, при условии, что оно допускает приемлемое строковое преобразование.

Ниже приведен пример программы, демонстрирующий применение оператора `assert`. В этом примере проверяется, что метод `getnum()` возвращает положительное значение.

```
// Продемонстрировать применение оператора assert  
class AssertDemo {
```



```

static int val = 3;

// вернуть целочисленное значение
static int getnum() {
    return val--;
}

public static void main(String args[])
{
    int n;

    for(int i=0; i < 10; i++) {
        n = getnum();

        assert n > 0; // не подтвердится, если n == 0

        System.out.println("n равно " + n);
    }
}

```

Чтобы разрешить проверку утверждений во время выполнения, следует указать параметр **-ea** в командной строке. Например, для проверки утверждений в классе `AssertDemo` нужно ввести следующую команду:

```
java -ea AssertDemo
```

После компиляции и запуска только что описанным образом эта программа выводит следующий результат:

```

n равно 3
n равно 2
n равно 1
Exception in thread "main" java.lang.AssertionError
    at AssertDemo.main(AssertDemo.java:17)
<Исключение в потоке "main" java.lang.AssertionError
    при вызове AssertDemo.main(AssertDemo.java:17)>

```

В методе `main()` выполняются повторяющиеся вызовы метода `getnum()`, возвращающего целочисленное значение. Это значение присваивается переменной `n`, а затем проверяется в операторе `assert`, как показано ниже.

```
assert n > 0; // не подтвердится, если n == 0
```

Исход вычисления этого оператора окажется неудачным, т.е. утверждение не подтвердится, если значение переменной `n` будет равно нулю, что произойдет после четвертого вызова метода `getnum()`. И когда это произойдет, будет сгенерировано исключение.

Как пояснялось выше, имеется возможность задать сообщение, которое выводится, если утверждение не подтверждается. Так, если подставить следующее утверждение в исходный код из предыдущего примера программы:

```
assert n > 0 : "n отрицательное!";
```

то будет выдан такой результат:

```

n равно 3
n равно 2
n равно 1

```

```
Exception in thread "main" java.lang.AssertionError : n отрицательное!  
at AssertDemo.main(AssertDemo.java:17)
```

Для правильного понимания утверждений важно иметь в виду следующее: на них нельзя полагаться для выполнение каких-нибудь конкретных действий в программе. Дело в том, что отлаженный код окончательной версии программы будет выполняться с отключенным режимом проверки утверждений. Рассмотрим в качестве примера следующий вариант предыдущей программы:

```
// Неудачное применение оператора assert!!!  
class AssertDemo {  
    // получить генератор случайных чисел  
    static int val = 3;  
  
    // вернуть целочисленное значение  
    static int getnum() {  
        return val--;  
    }  
  
    public static void main(String args[])  
    {  
        int n = 0;  
  
        for(int i=0; i < 10; i++) {  
            assert (n = getnum()) > 0; // Неудачная идея!  
  
            System.out.println("n is " + n);  
        }  
    }  
}
```

В этой версии программы вызов метода `getnum()` перенесен в оператор `assert`. И хотя такой прием оказывается вполне работоспособным, когда активизирован режим проверки утверждений, отключение этого режима приведет к неправильной работе программы, потому что вызов метода `getnum()` так и не произойдет! По существу, значение переменной `n` должно быть теперь инициализировано, поскольку компилятор выявит, что это значение может и не быть присвоено в операторе `assert`.

Утверждения являются полезным нововведением в Java, потому что они упрощают проверку ошибок, часто выполняемую на стадии разработки. Так, если до появления утверждений нужно было проверить, что переменная `n` имеет в приведенном выше примере программы положительное значение, то пришлось бы написать последовательность кода, аналогичную следующей:

```
if(n < 0) {  
    System.out.println("n отрицательное!");  
    return; // или сгенерировать исключение  
}
```

А с появлением утверждений для той же самой проверки требуется только одна строка кода. Более того, строки кода с оператором `assert` не придется удалять из окончательного варианта прикладного кода.

Параметры включения и отключения режима проверки утверждений

При выполнении кода можно отключить все утверждения, указав параметр `-da`. Включить или отключить режим проверки утверждений можно для конкретного пакета (и всех его внутренних пакетов), указав его имя и три точки после параметра `-ea` или `-da`. Например, чтобы включить режим проверки утверждений в пакете `MyPack`, достаточно ввести следующее:

```
-ea:MyPack...
```

А для того чтобы отключить режим проверки утверждений, ввести следующее:

```
-da:MyPack...
```

Кроме того, класс можно указать с параметром `-ea` или `-da`. В качестве примера ниже показано, как включить режим проверки утверждений отдельно в классе `AssertDemo`.

```
-ea:AssertDemo
```

Статический импорт

В Java имеется языковое средство, расширяющее возможности ключевого слова `import` и называемое *статическим импортом*. Оператор `import`, предваряемый ключевым словом `static`, можно применять для импорта статических членов класса или интерфейса. Благодаря статическому импорту появляется возможность ссылаться на статические члены непосредственно по именам, не уточняя их именем класса. Это упрощает и сокращает синтаксис, требующийся для работы со статическими членами.

Чтобы стала понятнее польза от статического импорта, начнем с примера, в котором он *не* используется. В приведенной ниже программе вычисляется гипотенуза прямоугольного треугольника. С этой целью вызываются два статических метода из встроенного в Java класса `Math`, входящего в пакет `java.lang`. Первый из них — метод `Math.pow()` — возвращает числовое значение, возведенное в указанную степень, а второй — метод `Math.sqrt()` — возвращает квадратный корень числового значения аргумента.

```
// Вычислить длину гипотенузы прямоугольного треугольника
class Hypot {
    public static void main(String args[]) {
        double side1, side2;
        double hypot;
        side1 = 3.0;
        side2 = 4.0;

        // Обратите внимание на то, что имена методов sqrt() и pow()
        // должны быть уточнены именем их класса - Math
        hypot = Math.sqrt(Math.pow(side1, 2) +
                           Math.pow(side2, 2));

        System.out.println("При заданной длине сторон " +
```

```

        sidel + " и " + side2 +
        "гипотенуза равна " + hypot);
    }
}

```

Методы `pow()` и `sqrt()` являются статическими, поэтому они должны быть вызваны с указанием имени их класса — `Math`. Это приводит к следующему громоздкому коду вычисления гипотенузы:

```

hypot = Math.sqrt(Math.pow(sidel, 2) +
    Math.pow(side2, 2));

```

Как показывает данный простой пример, очень неудобно указывать каждый раз имя класса при вызове методов `pow()` и `sqrt()` или любых других встроенных в Java методов, выполняющих математические операции наподобие `sin()`, `cos()` и `tan()`.

Подобных неудобств можно избежать, если воспользоваться статическим импортом, как показано в приведенной ниже версии программы из предыдущего примера.

```

// Воспользоваться статическим импортом для доступа
// к встроенным в Java методам sqrt() и pow()
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;

// вычислить гипотенузу прямоугольного треугольника
class Hypot {
    public static void main(String args[]) {
        double sidel, side2;
        double hypot;

        sidel = 3.0;
        side2 = 4.0;

        // Здесь методы sqrt() и pow() можно вызывать
        // непосредственно, опуская имя их класса
        hypot = sqrt(pow(sidel, 2) + pow(side2, 2));

        System.out.println("При заданной длине сторон " +
            sidel + " и " + side2 +
            " гипотенуза равна " + hypot);
    }
}

```

В этой версии программы имена методов `sqrt()` и `pow()` становятся видимыми благодаря оператору статического импорта, как показано ниже.

```

import static java.lang.Math.sqrt;
import static java.lang.Math.pow;

```

После этих операторов больше нет нужды уточнять имена методов `pow()` и `sqrt()` именем их класса. Таким образом, вычисление гипотенузы может быть выражено более удобным способом, как показано ниже. Как видите, эта форма не только упрощает код, но и делает его более удобочитаемым.

```

hypot = sqrt(pow(sidel, 2) + pow(side2, 2));

```

Имеются две основные формы оператора `import static`. Первая форма, употреблявшаяся в предыдущем примере программы, делает видимым единственное имя. В общем виде эта форма статического импорта такова:

```
import static пакет.имя_типа.имя_статического_члена;
```

где *имя_типа* обозначает имя класса или интерфейса, который содержит требуемый статический член. Полное имя его пакета указано в части *пакет*, а имя члена — в части *имя_статического_члена*.

Вторая форма статического импорта позволяет импортировать все статические члены данного класса или интерфейса. В общем виде эта форма выглядит следующим образом:

```
import static пакет.имя_типа.*;
```

Если предполагается применять много статических методов или полей, определенных в классе, то эта форма позволяет сделать их доступными, не вызывая каждый из них в отдельности. Так, в предыдущем примере программы с помощью единственного оператора `import` можно сделать доступными методы `pow()` и `sqrt()`, а также *все остальные* статические члены класса `Math`, как показано ниже.

```
import static java.lang.Math.*;
```

Разумеется, статический импорт не ограничивается только классом `Math` или его методами. Например, в следующей строке становится доступным статическое поле `System.out`.

```
import static java.lang.System.out;
```

После этого оператора данные можно выводить на консоль, не уточняя стандартный поток вывода `out` именем его класса `System`, как показано ниже.

```
out.println("Импортировав стандартный поток вывода System.out, " +  
            "можно пользоваться им непосредственно.");
```

Тем не менее приведенный выше способ импорта стандартного потока вывода `System.out` столь же удобен, сколь и полемичен. Несмотря на то что такой способ сокращает исходный текст программы, тем, кто его читает, не вполне очевидно, что `out` обозначает `System.out`. Следует также иметь в виду, что, помимо импорта статических членов классов и интерфейсов, определенных в прикладном программном интерфейсе `Java API`, импортировать статическим способом можно также статические члены своих собственных классов и интерфейсов.

Каким бы удобным ни был статический импорт, очень важно не злоупотреблять им. Не следует забывать, что библиотечные классы `Java` объединяются в пакеты для того, чтобы избежать конфликтов пространств имен и непреднамеренного сокрытия прочих имен. Если статический член используется в прикладной программе только один или два раза, то его лучше *не* импортировать. К тому же некоторые статические имена, как, например, `System.out`, настолько привычны и узнаваемы, что их вряд ли стоит вообще импортировать. Статический импорт следует оставить на тот случай, если статические члены применяются многократно, как, например, при выполнении целого ряда математических вы-

числений. В сущности, этим языковым средством стоит пользоваться, но только не злоупотреблять им.

Вызов перегружаемых конструкторов по ссылке `this()`

Пользуясь перегружаемыми конструкторами, иногда удобно вызывать один конструктор из другого. Для этого в Java имеется еще одна форма ключевого слова `this`. В общем виде эта форма выглядит следующим образом:

`this(список_аргументов)`

По ссылке `this()` сначала выполняется перегружаемый конструктор, который соответствует заданному `списку_аргументов`, а затем — любые операторы, находящиеся в теле исходного конструктора, если таковые имеются. Вызов конструктора по ссылке `this()` должен быть первым оператором в конструкторе.

Чтобы стало понятнее, как пользоваться ссылкой `this()`, обратимся к краткому примеру. Рассмотрим сначала приведенный ниже пример класса, в котором ссылка `this()` не используется.

```
class MyClass {
    int a;
    int b;

    // инициализировать поля a и b по отдельности
    MyClass(int i, int j) {
        a = i;
        b = j;
    }

    // инициализировать поля a и b одним и тем же значением
    MyClass(int i) {
        a = i;
        b = i;
    }

    // присвоить полям a и b нулевое значение по умолчанию
    MyClass() {
        a = 0;
        b = 0;
    }
}
```

Этот класс содержит три конструктора, каждый из которых инициализирует значения полей `a` и `b`. Первому конструктору передаются отдельные значения для инициализации полей `a` и `b`. Второй конструктор принимает только одно значение и присваивает его обоим полям, `a` и `b`. А третий присваивает полям `a` и `b` нулевое значение по умолчанию.

Используя ссылку `this()`, приведенный выше класс `MyClass` можно переписать следующим образом:

```

class MyClass {
    int a;
    int b;

    // инициализировать поля a и b по отдельности
    MyClass(int i, int j) {
        a = i;
        b = j;
    }

    // инициализировать поля a и b одним и тем же значением
    MyClass(int i) {
        this(i, i); // по этой ссылке вызывается
                    // конструктор MyClass(i, i);
    }

    // присвоить полям a и b нулевое значение по умолчанию
    MyClass() {
        this(0); // a по этой ссылке вызывается
                 // конструктор MyClass(0)
    }
}

```

В этой версии класса `MyClass` значения полям `a` и `b` в действительности присваиваются только в конструкторе `MyClass(int, int)`. А два других конструктора просто вызывают первый конструктор (прямо или косвенно) по ссылке `this()`. Рассмотрим, например, что произойдет, если выполнить следующий оператор:

```
MyClass mc = new MyClass(8);
```

Вызов конструктора `MyClass(8)` приводит к выполнению ссылки `this(8,8)`, которая преобразуется в вызов конструктора `MyClass(8,8)`, поскольку именно эта версия конструктора класса `MyClass` соответствует списку аргументов, передаваемому по ссылке `this()`. А теперь рассмотрим следующий оператор, в котором используется конструктор по умолчанию:

```
MyClass mc2 = new MyClass();
```

В данном случае происходит обращение по ссылке `this(0)`, приводящее к вызову конструктора `MyClass(0)`, поскольку именно эта версия конструктора соответствует списку параметров. Разумеется, конструктор `MyClass(0)` вызывает далее конструктор `MyClass(0,0)`, как пояснялось выше.

Одной из причин, по которой стоит вызывать перегружаемые конструкторы по ссылке `this()`, служит потребность избежать дублирования кода. Зачастую сокращение дублированного кода ускоряет загрузку классов, поскольку объектный код становится компактнее. Это особенно важно для программ, доставляемых через Интернет, когда время их загрузки критично. Применение ссылки `this()` позволяет также оптимально структурировать прикладной код, когда конструкторы содержат большой объем дублированного кода.

Следует, однако, иметь в виду, что конструкторы, вызывающие другие конструкторы по ссылке `this()`, выполняются медленнее, чем те, что содержат весь код, необходимый для инициализации. Дело в том, что механизм вызова и возвращения, используемый при вызове второго конструктора, требует дополнительных издержек. Если класс предполагается употреблять для создания

небольшого количества объектов или если его конструкторы, вызывающие друг друга по ссылке `this()`, будут использоваться редко, то снижение производительности во время выполнения, скорее всего, окажется незначительным. Но если во время выполнения программы предполагается создание большого количества (порядка тысяч) объектов такого класса, то дополнительные издержки, связанные с вызовом одних конструкторов из других по ссылке `this()`, могут отрицательно сказаться на производительности. Создание объектов затрагивает всех пользователей такого класса, и поэтому придется тщательно взвесить преимущества более быстрой загрузки в сравнении с увеличением времени на создание объекта.

Следует также иметь в виду, что вызов очень коротких конструкторов, как, например, из класса `MyClass`, по ссылке `this()` зачастую лишь незначительно увеличивает размер объектного кода. (В некоторых случаях никакого уменьшения объема объектного кода вообще не происходит.) Дело в том, что байт-код, который устанавливается и возвращается из вызова конструктора по ссылке `this()`, добавляет инструкции в объектный файл. Поэтому в таких случаях вызов конструктора по ссылке `this()`, несмотря на исключение дублирования кода, не даст значительной экономии времени загрузки, но может повлечь за собой дополнительные издержки на создание каждого объекта. Поэтому применение ссылки `this()` больше всего подходит для вызова тех конструкторов, которые содержат большой объем кода инициализации, а не тех, которые просто устанавливают значения в нескольких полях.

Вызывая конструкторы по ссылке `this()`, следует учитывать следующее. Во-первых, при вызове конструктора по ссылке `this()` нельзя использовать переменные экземпляра класса этого конструктора. И во-вторых, в одном и том же конструкторе нельзя использовать ссылки `super()` и `this()`, поскольку каждая из них должна быть первым оператором в конструкторе.

Компактные профили Java API

В версии JDK 8 внедрено средство, позволяющее организовать подмножества библиотеки прикладных программных интерфейсов API в так называемые *компактные профили*. Они обозначаются следующим образом: `compact1`, `compact2` и `compact3`. Каждый такой профиль содержит подмножество библиотеки. Более того, компактный профиль `compact2` включает в себя весь профиль `compact1`, а компактный профиль `compact3` — весь профиль `compact2`. Следовательно, каждый последующий компактный профиль строится на основании предыдущего. Преимущество компактных профилей заключается в том, что прикладной программе не нужно загружать библиотеку полностью. Применение компактных профилей позволяет сократить размер библиотеки, а следовательно, выполнять некоторые категории прикладных программ на тех устройствах, где отсутствует полная поддержка прикладного программного интерфейса Java API. Благодаря компактным профилям удастся также сократить время, требующееся для загрузки программы. В документации на прикладной программный интерфейс Java API ука-

зывается, к какому именно элементу этого прикладного интерфейса принадлежит компактный профиль, если это вообще имеет место.

На стадии компиляции программы с помощью параметра командной строки **-profile** можно указать, какие именно элементы прикладного программного интерфейса Java API, определяемые компактным профилем, следует использовать в программе. Ниже приведена общая форма команды, по которой программу можно скомпилировать именно таким способом.

```
javac -profile имя_профиля имя_программы
```

Здесь *имя_профиля* обозначает конкретный профиль: compact1, compact2 или compact3. Например:

```
javac -profile compact2 Test.java
```

В этом примере команды компиляции указан профиль compact2. Если же исходный файл программы Test.java содержит элементы прикладного программного интерфейса Java API, отсутствующие в компактном профиле compact2, то во время компиляции возникнет ошибка.

После выхода в 1995 году первоначальной версии 1.0 в Java было внедрено немало новых языковых средств. Одним из наиболее значительных нововведений, повлиявших на дальнейшую судьбу Java, стали *обобщения*. Во-первых, их появление означало внедрение новых синтаксических элементов в язык. Во-вторых, они повлекли за собой изменения во многих классах и методах базового прикладного программного интерфейса API. Ныне обобщения являются неотъемлемой частью программирования на Java, и поэтому требуется ясное понимание этого важного языкового средства. И в этой главе оно исследуется во всех подробностях.

Применение обобщений позволило создавать классы, интерфейсы и методы, работающие безопасным по отношению к типам способом с разнообразными видами данных. Многие алгоритмы логически идентичны, независимо от того, к данным каких типов они применяются. Например, механизм, поддерживающий стеки, является одним и тем же в стеках, хранящих элементы типа Integer, String, Object или Thread. Благодаря обобщениям можно определить алгоритм один раз независимо от конкретного типа данных, а затем применять его к обширному разнообразию типов данных без каких-нибудь дополнительных усилий. Впечатляющая эффективность внедренных в Java обобщений коренным образом изменила способы написания кода на этом языке программирования.

Вероятно, одним из средств Java, которое в наибольшей степени испытало на себе влияние обобщений, является каркас *коллекций* Collections Framework. Этот каркас является частью прикладного программного интерфейса Java API и подробно обсуждается в главе 18, но стоит дать хотя бы краткое пояснение его назначения. *Коллекция* — это группа объектов, а в каркасе коллекций определяется ряд классов для управления такими коллекциями, как, например, списки и отображения. Классы коллекций всегда были способны обращаться с объектами любых типов. Но выгода от внедрения в Java обобщений в том и состоит, что классы коллекций можно теперь применять с полной гарантией типовой безопасности. Таким образом, обобщения являются не только эффективным языковым средством, но и способны значительно усовершенствовать уже существующие средства. Именно поэтому обобщения являются столь значительным дополнением языка Java.

В этой главе описываются синтаксис, теория и практика применения обобщений. В ней будет показано, каким образом обобщения обеспечивают типовую безопасность в некоторых трудных прежде случаях. Прочитав эту главу, вы, скорее всего, захотите ознакомиться с материалом главы 18, посвященной каркасу

коллекций Collections Framework, где вы найдете немало примеров применения обобщений на практике.

Что такое обобщения

По существу, *обобщения* — это *параметризованные типы*. Такие типы важны, поскольку они позволяют объявлять классы, интерфейсы и методы, где тип данных, которыми они оперируют, указан в виде параметра. Используя обобщения, можно, например, создать единственный класс, который будет автоматически обращаться с разнотипными данными. Классы, интерфейсы или методы, оперирующие параметризованными типами, называются *обобщенными*.

Следует заметить, что в Java всегда предоставлялась возможность создавать в той или иной степени обобщенные классы, интерфейсы и методы, оперирующие ссылками типа `Object`. А поскольку класс `Object` служит суперклассом для всех остальных классов, то он позволяет обращаться к объекту любого типа. Следовательно, в старом коде ссылки типа `Object` использовались в обобщенных классах, интерфейсах и методах с целью оперировать разнотипными объектами. Но дело в том, что они не могли обеспечить типовую безопасность.

Именно обобщения внесли в язык типовую безопасность типов, которой так недоставало прежде. Они также упростили процесс выполнения, поскольку теперь нет нужды в явном приведении типов для преобразования объектов типа `Object` в конкретные типы обрабатываемых данных. Благодаря обобщениям все операции приведения типов выполняются автоматически и неявно. Таким образом, обобщения расширили возможности повторного использования кода, позволив делать это легко и безопасно.

На заметку! Программирующим на C++ следует иметь в виду, что обобщения и шаблоны в C++ — это не одно и то же, хотя они и похожи. У этих двух подходов к обобщенным типам есть ряд принципиальных отличий. Если у вас имеется некоторый опыт программирования на C++, не спешите делать поспешные выводы о том, как обобщения действуют в Java.

Простой пример обобщения

Начнем с простого примера обобщенного класса. В приведенной ниже программе определяются два класса. Первый из них — обобщенный класс `Gen`, второй — демонстрационный класс `GenDemo`, в котором используется обобщенный класс `Gen`.

```
// Простой обобщенный класс.
// Здесь T обозначает параметр типа,
// который будет заменен реальным типом
// при создании объекта типа Gen
class Gen<T> {
    T ob; // объявить объект типа T

    // передать конструктору ссылку на объект типа T
    Gen(T o) {
```

```

        ob = o;
    }

    // вернуть объект ob
    T getob() {
        return ob;
    }

    // показать тип T
    void showType() {
        System.out.println("Типом T является " + ob.getClass().getName());
    }
}

// продемонстрировать применение обобщенного класса
class GenDemo {
    public static void main(String args[]) {
        // Создать ссылку типа Gen для целых чисел
        Gen<Integer> iOb;

        // Создать объект типа Gen<Integer> и присвоить
        // ссылку на него переменной iOb. Обратите внимание на
        // применение автоупаковки для инкапсуляции значения 88
        // в объекте типа Integer
        iOb = new Gen<Integer>(88);

        // показать тип данных, хранящихся в переменной iOb
        iOb.showType();

        // получить значение переменной iOb. Обратите внимание на то,
        // что для этого не требуется никакого приведения типов
        int v = iOb.getob();
        System.out.println("Значение: " + v);
        System.out.println();

        // создать объект типа Gen для символьных строк
        Gen<String> strOb = new Gen<String> ("Тест обобщений");

        // показать тип данных, хранящихся в переменной strOb
        strOb.showType();

        // получить значение переменной strOb. И в этом случае
        // приведение типов не требуется
        String str = strOb.getob();
        System.out.println("Значение: " + str);
    }
}

```

Ниже приведен результат, выводимый данной программой.

```

Типом T является java.lang.Integer
Значение: 88

```

```

Типом T является java.lang.String
Значение: Тест обобщений

```

Внимательно проанализируем эту программу. Обратите внимание на объявление класса Gen в следующей строке кода:

```

class Gen<T> {

```

где **T** обозначает имя *параметра типа*. Это имя используется в качестве заполнителя, вместо которого в дальнейшем подставляется имя конкретного типа, передаваемого классу `Gen` при создании объекта. Это означает, что обозначение **T** применяется в классе `Gen` всякий раз, когда требуется параметр типа. Обратите внимание на то, что обозначение **T** заключено в угловые скобки (`<>`). Этот синтаксис может быть обобщен. Всякий раз, когда объявляется параметр типа, он указывается в угловых скобках. В классе `Gen` применяется параметр типа, и поэтому он является обобщенным классом, относящимся к так называемому *параметризованному типу*.

Далее тип **T** используется для объявления объекта `ob`:

```
T ob; // объявить объект типа T
```

Как упоминалось выше, параметр типа **T** — это место для подстановки конкретного типа, который указывается в дальнейшем при создании объекта класса `Gen`. Это означает, что объект `ob` станет объектом того типа, который будет передан в качестве параметра типа **T**. Так, если передать тип `String` в качестве параметра типа **T**, то такой экземпляр объекта `ob` будет иметь тип `String`.

Рассмотрим далее конструктор `Gen()`. Его код приведен ниже.

```
Gen(T o) {
    ob = o;
}
```

Как видите, параметр `o` имеет тип **T**. Это означает, что конкретный тип параметра `o` определяется с помощью параметра типа **T**, передаваемого при создании объекта класса `Gen`. А поскольку параметр `o` и переменная экземпляра `ob` относятся к типу **T**, то они получают одинаковый конкретный тип при создании объекта класса `Gen`.

Параметр типа **T** может быть также использован для указания типа, возвращаемого методом, как показано ниже на примере метода `getob()`. Объект `ob` также относится к типу **T**, поэтому его тип совместим с типом, возвращаемым методом `getob()`.

```
T getob() {
    return ob;
}
```

Метод `showType()` отображает тип **T**, вызывая метод `getName()` для объекта типа `Class`, возвращаемого в результате вызова метода `getClass()` для объекта `ob`. Метод `getClass()` определен в классе `Object`, и поэтому он является членом всех классов. Этот метод возвращает объект типа `Class`, соответствующий типу того класса объекта, для которого он вызывается. В классе `Class` определяется метод `getName()`, возвращающий строковое представление имени класса.

Класс `GenDemo` служит для демонстрации обобщенного класса `Gen`. Сначала в нем создается версия класса `Gen` для целых чисел, как показано ниже.

```
Gen<Integer> iOb;
```

Проанализируем это объявление внимательнее. Обратите внимание на то, что тип `Integer` указан в угловых скобках после слова `Gen`. В данном случае `Integer` — это *аргумент типа*, который передается в качестве параметра типа **T** из класса `Gen`. Это объявление фактически означает создание версии класса `Gen`, где все ссылки

на тип `T` преобразуются в ссылки на тип `Integer`. Таким образом, в данном объявлении объект `ob` относится к типу `Integer`, и метод `getob()` возвращает тип `Integer`.

Прежде чем продолжить дальше, следует сказать, что компилятор Java на самом деле не создает разные версии класса `Gen` или любого другого обобщенного класса. Теоретически это было бы удобно, но на практике дело обстоит иначе. Вместо этого компилятор удаляет все сведения об обобщенных типах, выполняя необходимые операции приведения типов, чтобы сделать поведение прикладного кода таким, как будто создана конкретная версия класса `Gen`. Таким образом, имеется только одна версия класса `Gen`, которая существует в прикладной программе. Процесс удаления обобщенной информации об обобщенных типах называется *стифанием*, и мы еще вернемся к этой теме далее в главе.

В следующей строке кода переменной `iOb` присваивается ссылка на экземпляр целочисленной версии класса `Gen`:

```
iOb = new Gen<Integer>(88);
```

Обратите внимание на то, что, когда вызывается конструктор `Gen()`, аргумент типа `Integer` также указывается. Это необходимо потому, что объект (в данном случае — `iOb`), которому присваивается ссылка, относится к типу `Gen<Integer>`. Следовательно, ссылка, возвращаемая оператором `new`, также должна относиться к типу `Gen<Integer>`. В противном случае во время компиляции возникает ошибка. Например, следующее присваивание вызовет ошибку во время компиляции:

```
iOb = new Gen<Double>(88.0); // ОШИБКА!
```

Переменная `iOb` относится к типу `Gen<Integer>`, поэтому она не может быть использована для присваивания ссылки типа `Gen<Double>`. Такая проверка типа является одним из основных преимуществ обобщений, потому что она обеспечивает типовую безопасность.

На заметку! Как будет показано далее в этой главе, в версии JDK 7 появилась возможность употреблять сокращенный синтаксис для создания экземпляра обобщенного класса. Но ради ясности изложения в данном случае используется полный синтаксис.

Как следует из комментариев к данной программе, в приведенном ниже присваивании выполняется автоупаковка для инкапсуляции значения `88` типа `int` в объекте типа `Integer`.

```
iOb = new Gen<Integer>(88);
```

Такое присваивание вполне допустимо, поскольку обобщение `Gen<Integer>` создает конструктор, принимающий аргумент типа `Integer`. А поскольку предполагается объект типа `Integer`, то значение `88` автоматически упаковывается в этот объект. Разумеется, присваивание может быть написано и явным образом, как показано ниже, но такой его вариант не дает никаких преимуществ.

```
iOb = new Gen<Integer>(new Integer(88));
```

Затем в данной программе отображается тип объекта `ob` в переменной `iOb` (в данном случае — тип `Integer`). А далее получается значение объекта `ob` в следующей строке:

```
int v = iOb.getob();
```

Метод `getob()` возвращает обобщенный тип `T`, который был заменен на тип `Integer` при объявлении переменной экземпляра `iOb`. Поэтому метод `getob()` также возвращает тип `Integer`, который автоматически распаковывается в тип `int` и присваивается переменной `v` типа `int`. Следовательно, тип, возвращаемый методом `getob()`, нет никакой нужды приводить к типу `Integer`. Безусловно, выполнять автоупаковку необязательно, переписав предыдущую строку кода так, как показано ниже. Но автоупаковка позволяет сделать код более компактным.

```
int v = iOb.getob().intValue();
```

Далее в классе `GenDemo` объявляется объект типа `Gen<String>` следующим образом:

```
Gen<String> strOb = new Gen<String>("Тест обобщений");
```

В качестве аргумента типа в данном случае указывается тип `String`, подставляемый вместо параметра типа `T` в обобщенном классе `Gen`. Это, по существу, приводит к созданию строковой версии класса `Gen`, что и демонстрируется в остальной части рассматриваемой здесь программы.

Обобщения действуют только со ссылочными типами

Когда объявляется экземпляр обобщенного типа, аргумент, передаваемый в качестве параметра типа, должен относиться к ссылочному типу, но ни в коем случае не к примитивному типу наподобие `int` или `char`. Например, в качестве параметра `T` классу `Gen` можно передать тип любого класса, но нельзя передать примитивный тип. Таким образом, следующее объявление недопустимо:

```
Gen<int> intOb = new Gen<int>(53); // ОШИБКА! Использовать  
// примитивные типы нельзя!
```

Безусловно, отсутствие возможности использовать примитивный тип не является серьезным ограничением, поскольку можно применять оболочки типов данных (как это делалось в предыдущем примере программы) для инкапсуляции примитивных типов. Более того, механизм автоупаковки и автораспаковки в Java делает прозрачным применение оболочек типов данных.

Обобщенные типы различаются по аргументам типа

В отношении обобщенных типов самое главное понять, что ссылка на одну конкретную версию обобщенного типа несовместима с другой версией того же самого обобщенного типа. Так, если ввести следующую строку кода в предыдущую программу, то при ее компиляции возникнет ошибка:

```
iOb = strOb; // НЕВЕРНО!
```

Несмотря на то что переменные экземпляра `iOb` и `strOb` относятся к типу `Gen<T>`, они являются ссылками на разные типы объектов, потому что их параметры типов отличаются. Этим, в частности, обобщения обеспечивают типовую безопасность, предотвращая ошибки подобного рода.

Каким образом обобщения повышают типовую безопасность

В связи с изложенным выше может возникнуть следующий вопрос: если те же самые функциональные возможности, которые были обнаружены в обобщенном классе `Gen`, могут быть получены и без обобщений, т.е. простым указанием класса `Object` в качестве типа данных и правильным приведением типов, то в чем же польза от того, что класс `Gen` является обобщенным? Дело в том, что обобщения автоматически гарантируют типовую безопасность во всех операциях, где задействован обобщенный класс `Gen`. В процессе его применения исключается потребность в явном приведении и ручной проверке типов в прикладном коде.

Чтобы стали понятнее выгоды от обобщений, рассмотрим сначала следующий пример программы, в которой создает необобщенный эквивалент класса `Gen`:

```
// Класс NonGen – функциональный эквивалент класса Gen без обобщений
class NonGen {
    Object ob; // объект ob теперь имеет тип Object

    // передать конструктору ссылку на объект типа Object
    NonGen(Object o) {
        ob = o;
    }

    // вернуть тип Object
    Object getob() {
        return ob;
    }

    // показать тип объекта ob
    void showType() {
        System.out.println("Объект ob относится к типу " +
            ob.getClass().getName());
    }
}

// продемонстрировать необобщенный класс
class NonGenDemo {
    public static void main(String args[]) {
        NonGen iOb;

        // создать объект типа NonGen и сохранить в нем
        // объект типа Integer. Выполняется автоупаковка
        iOb = new NonGen(88);

        // показать тип данных, хранящихся в переменной iOb
        iOb.showType();

        // получить значение переменной iOb,
        // на этот раз требуется приведение типов
        int v = (Integer) iOb.getob();
        System.out.println("Значение: " + v);
        System.out.println();

        // создать другой объект типа NonGen и
        // сохранить в нем объект типа String
        NonGen strOb = new NonGen("Тест без обобщений");
    }
}
```



```

// показать тип данных, хранящихся в переменной strOb
strOb.showType();

// получить значение переменной strOb,
// И в этом случае потребуется приведение типов
String str = (String) strOb.getOb();
System.out.println("Значение: " + str);

// Этот код компилируется, но он принципиально неверный!
iOb = strOb;
v = (Integer) iOb.getOb(); // Ошибка во время выполнения!
}
}

```

В этой версии программы обращает на себя внимание ряд интересных моментов. Прежде всего, в классе `NonGen` все ссылки на тип `T` заменены ссылками на тип `Object`. Это позволяет хранить в классе `NonGen` объекты любого типа, как и в обобщенном классе `Gen`. Но это не позволяет компилятору Java получить какие-нибудь подлинные сведения о типе данных, фактически сохраняемых в объекте класса `NonGen`, что плохо по двум причинам. Во-первых, для извлечения сохраненных данных требуется явное приведение типов. И во-вторых, многие ошибки несоответствия типов не могут быть обнаружены до времени выполнения. Рассмотрим каждый из этих недостатков подробнее.

Обратите внимание на следующую строку кода:

```
int v = (Integer) iOb.getOb();
```

Метод `getOb()` возвращает тип `Object`, поэтому его нужно привести к типу `Integer`, чтобы выполнить автораспаковку и сохранить значение в переменной `v`. Если убрать приведение типов, программа не скомпилируется. Если в ее версии с обобщениями приведение типов производится неявно, то в версии без обобщений приведение должно быть сделано явно. Это не только неудобно, но и служит потенциальным источником ошибок.

Теперь рассмотрим следующий фрагмент кода в конце данной программы:

```

// Этот код компилируется, но он принципиально неверный!
iOb = strOb;
v = (Integer) iOb.getOb(); // Ошибка во время выполнения!

```

Здесь переменной экземпляра `iOb` присваивается значение переменной экземпляра `strOb`. Но переменная экземпляра `strOb` ссылается на объект, содержащий символьную строку, а не целое число. Такое присваивание синтаксически корректно, потому что все ссылки типа `NonGen` одинаковы и любая ссылка типа `NonGen` может указывать на любой другой объект типа `NonGen`. Но семантически эта операция присваивания неверна, что и отражено в следующей строке кода. Здесь тип, возвращаемый методом `getOb()`, приводится к типу `Integer`, а затем делается попытка присвоить полученное значение переменной `v`. Дело в том, что переменная экземпляра `iOb` теперь ссылается на объект, хранящий данные типа `String`, а не `Integer`. К сожалению, без обобщений компилятор Java просто не в состоянии обнаружить эту ошибку. Вместо этого во время выполнения генерируется исключение при попытке привести к типу `Integer`. Но как вам, должно быть, уже известно, появление ошибок во время выполнения кода неприемлемо!

Подобной бы ошибки не произошло, если бы в данной программе использовались обобщения. Упомянутая выше попытка присваивания разнотипных объектов в обобщенной версии данной программы была бы обнаружена уже на стадии компиляции и не привела бы к серьезной ошибке во время выполнения. Возможность создавать типизированный (т.е. обеспечивающий типовую безопасность) код, в котором ошибки несоответствия типов перехватываются компилятором, является главным преимуществом обобщений. Несмотря на то что пользоваться ссылками на тип `Object` для создания “псевдообобщенного” кода можно было всегда, не следует забывать, что такой код не обеспечивает типовую безопасность, а злоупотребление им приводит к исключениям во время выполнения. Обобщения предотвращают подобные осложнения. По существу, благодаря обобщениям ошибки, возникающие во время выполнения, преобразуются в ошибки, обнаруживаемые во время компиляции. В этом и заключается главное преимущество обобщений.

Обобщенный класс с двумя параметрами типа

Для обобщенного типа можно объявлять не только один параметр. Два или более параметра типа можно указать списком через запятую. Например, приведенный ниже класс `TwoGen` является переделанным вариантом класса `Gen`, принимающим два параметра типа.

```
// Простой обобщенный класс с двумя параметрами типа: T и V
class TwoGen<T, V> {
    T ob1;
    V ob2;

    // передать конструктору ссылки на объекты типа T и V
    TwoGen(T o1, V o2) {
        ob1 = o1;
        ob2 = o2;
    }

    // показать типы T и V
    void showTypes() {
        System.out.println("Тип T: " + ob1.getClass().getName());

        System.out.println("Тип V: " + ob2.getClass().getName());
    }

    T getob1() {
        return ob1;
    }

    V getob2() {
        return ob2;
    }
}

// продемонстрировать применение класса TwoGen
class SimpGen {
    public static void main(String args[]) {

        TwoGen<Integer, String> tgObj =
            new TwoGen<Integer, String>(88, "Обобщения");
    }
}
```

```

// показать типы
tgObj.showTypes();

// Получить и показать значения
int v = tgObj.getob1();
System.out.println("Значение: " + v);

String str = tgObj.getob2();
System.out.println("Значение: " + str);
}
}

```

Ниже приведен пример выполнения данной программы.

```

Тип T: java.lang.Integer
Тип V: java.lang.String
Значение: 88
Значение: Обобщения

```

Обратите внимание на следующее объявление класса TwoGen:

```
class TwoGen<T, V> {
```

В этом объявлении два параметра типа T и V задаются списком через запятую. А поскольку в объявлении этого класса указаны два параметра типа, то при создании объекта типа TwoGen должны быть переданы два аргумента типа, как показано ниже.

```
TwoGen<Integer, String> tgObj =
    new TwoGen<Integer, String> (88, "Обобщения");
```

В этом случае тип Integer подставляется вместо параметра типа T, а тип String — вместо параметра типа V.

В данном примере оба аргумента типа отличаются, тем не менее вполне допустимо передавать в качестве параметров два одинаковых типа. Например, следующая строка кода написана правильно:

```
TwoGen<String, String> x = new TwoGen<String, String> ("A", "B");
```

В этом случае оба аргумента, V и T, будут иметь тип String. Ясно, что если оба аргумента типа всегда одинаковы, то два параметра типа не нужны.

Общая форма обобщенного класса

Синтаксис, представленный в предыдущих примерах программ, может быть обобщен. Ниже показано, как выглядит синтаксис объявления обобщенного класса.

```
class имя_класса<список_параметров_типа> { // ...
```

А синтаксис объявления ссылки на обобщенный класс и создание его экземпляра полностью выглядит следующим образом:

```
имя_класса<список_аргументов_типа> имя_переменной =
    new имя_класса<список_аргументов_типа> (список_аргументов_констант);
```

Ограниченные типы

В предыдущих примерах программ параметры типов могли быть заменены типами любых классов. Это подходит для многих целей, но иногда удобно ограничить перечень типов, передаваемых в качестве параметров. Допустим, требуется создать обобщенный класс с методом, возвращающим среднее значение массива чисел. Более того, с помощью этого класса требуется получить среднее значение из целых чисел, а также чисел с плавающей точкой одинарной и двойной точности. Таким образом, тип числовых данных требуется указать обобщенно, используя параметр типа. Попытайтесь создать такой класс можно, например, следующим образом:

```
// Класс Stats – пример безуспешной попытки создать
// обобщенный класс для вычисления среднего значения
// массива чисел заданного типа
//
// Этот класс содержит ошибку!
class Stats<T> {
    T[] nums; // nums – это массив элементов типа T

    // передать конструктору ссылку на массив значений типа T
    Stats(T[] o) {
        nums = o;
    }

    // вернуть значение типа double в любом случае
    double average() {
        double sum = 0.0;
        for(int i=0; i < nums.length; i++)
            sum += nums[i].doubleValue(); // ОШИБКА!!!
        return sum / nums.length;
    }
}
```

Метод `average()` из класса `Stats` пытается получить версию типа `double` для каждого числа из массива `nums`, вызывая метод `doubleValue()`. Все классы оболочек числовых типов данных, в том числе `Integer` и `Double`, являются подклассами, производными от класса `Number`, а в классе `Number` определяется метод `doubleValue()`, поэтому данный метод доступен во всех классах оболочек числовых типов данных. Но дело в том, что компилятор не может знать, что автор программы намерен создавать объекты типа `Stats`, используя только числовые типы данных. Поэтому, когда класс `Stats` компилируется, выдается сообщение об ошибке, уведомляющее, что метод `doubleValue()` неизвестен. Чтобы устранить эту ошибку, придется найти какой-нибудь способ сообщить компилятору, что в качестве параметра `T` предполагается передавать числовые типы. Кроме того, требуется каким-то образом *гарантировать*, что передаваться будут *только* числовые типы данных.

Для подобных случаев в Java предоставляются *ограниченные типы*. Указывая параметр типа, можно наложить ограничение сверху в виде верхней границы, где объявляется суперкласс, от которого должны быть унаследованы все аргументы типов. С этой целью вместе с параметром указывается ключевое слово `extends`, как показано ниже.

<T extends суперкласс>

Это означает, что параметр типа **T** может быть заменен только указанным суперклассом или его подклассами. Следовательно, суперкласс объявляет верхнюю границу включительно. Наложив ограничение сверху, можно исправить упомянутый выше класс **Stats**, указав класс **Number** в виде верхней границы для параметра типа.

```
// В этой версии класса Stats аргумент типа T должен быть
// классом Number или наследуемым от него классом
class Stats<T extends Number> {
    T[] nums; // массив класса Number или его подкласса

    // передать конструктору ссылку на массив элементов
    // класса Number или его подкласса
    Stats(T[] o) {
        nums = o;
    }

    // вернуть значение типа double в любом случае
    double average() {
        double sum = 0.0;

        for(int i=0; i < nums.length; i++)
            sum += nums[i].doubleValue();

        return sum / nums.length;
    }
}

// продемонстрировать применение класса Stats
class BoundsDemo {
    public static void main(String args[]) {
        Integer inums[] = { 1, 2, 3, 4, 5 };
        Stats<Integer> iob = new Stats<Integer>(inums);
        double v = iob.average();
        System.out.println("Среднее значение iob равно " + v);

        Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
        Stats<Double> dob = new Stats<Double>(dnums);
        double w = dob.average();
        System.out.println("Среднее значение dob равно " + w);

        // Этот код не скомпилируется, так как класс String
        // не является производным от класса Number
        // String str[] = { "1", "2", "3", "4", "5" };
        // Stats<String> strob = new Stats<String>(str);
        // double x = strob.average();
        // System.out.println("Среднее значение strob равно " + v);
    }
}
```

Эта программа выводит следующий результат:

```
Среднее значение iob равно 3.0
Среднее значение dob равно 3.3
```

Обратите внимание на то, что класс **Stats** объявляется теперь следующим образом:

```
class Stats<T extends Number> {
```

Тип `T` ограничивается сверху классом `Number`, а следовательно, компилятору Java теперь известно, что все объекты типа `T` могут вызывать метод `doubleValue()`, поскольку это метод класса `Number`. И само по себе это уже серьезное преимущество. Но кроме того, ограничение параметра типа `T` предотвращает создание нечисловых объектов типа `Stats`. Так, если попытаться убрать комментарии из строк кода в конце данной программы и перекомпилировать ее, то во время компиляции будет выдана ошибка, поскольку класс `String` не является производным от класса `Number`.

В виде ограничения можно накладывать не только тип класса, но и тип интерфейса. Более того, такое ограничение может включать в себя как тип класса, так и типы одного или нескольких интерфейсов. В этом случае тип класса должен быть задан первым. Когда ограничение включает в себя тип интерфейса, допустимы только аргументы типа, реализующие этот интерфейс. Накладывая на обобщенный тип ограничение, состоящее из класса и одного или нескольких интерфейсов, для их объединения следует воспользоваться логической операцией **&**:

```
class Gen<T extends MyClass & MyInterface> { // ...
```

где параметр типа `T` ограничивается классом `MyClass` и интерфейсом `MyInterface`. Таким образом, любой тип, передаваемый параметру `T`, должен быть подклассом, производным от класса `MyClass` и реализующим интерфейс `MyInterface`.

Применение метасимвольных аргументов

Типовая безопасность удобна не только сама по себе, но иногда она позволяет получить идеально подходящие конструкции. Например, в классе `Stats`, рассмотренном в предыдущем разделе, предполагается, что в него требуется ввести метод `sameAvg()`, который определяет, содержат ли два объекта типа `Stats` массивы, дающие одинаковое среднее значение независимо от типа числовых значений в них. Так, если один такой объект содержит значения `1, 0, 2, 0` и `3, 0` типа `double`, а другой — целочисленные значения `2, 1` и `3`, то их среднее значение будет одинаковым. Для того чтобы реализовать метод `sameAvg()`, можно, в частности, передать ему аргумент типа `Stats`, а затем сравнить средние значения в этом методе и вызывающем объекте, возвращая логическое значение `true`, если они равны. Кроме того, необходимо иметь возможность вызывать метод `sameAvg()`, например, следующим образом:

```
Integer inums[] = { 1, 2, 3, 4, 5 };
Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };

Stats<Integer> iob = new Stats<Integer>(inums);
Stats<Double> dob = new Stats<Double>(dnums);

if(iob.sameAvg(dob))
    System.out.println("Средние значения одинаковы.");
else
    System.out.println("Средние значения отличаются.");
```

На первый взгляд, написать метод `sameAvg()` совсем не трудно. Ведь класс `Stats` является обобщенным, и поэтому его метод `average()` может оперировать

разнотипными объектами класса `Stats`. К сожалению, трудности появляются, стоит только попытаться объявить параметр типа `Stats`. Если `Stats` — параметризованный тип, то какой тип параметра следует указать для `Stats` при объявлении параметра этого типа? Вполне подходящим может показаться решение использовать `T` в качестве параметра типа следующим образом:

```
// Это решение не годится!
// Определить равенство двух средних значений
boolean sameAvg(Stats<T> ob) {
    if(average() == ob.average())
        return true;

    return false;
}
```

Недостаток такого решения заключается в том, что оно годится только для другого объекта класса `Stats`, тип которого совпадает с вызывающим объектом. Так, если вызывающий объект относится к типу `Stats<Integer>`, то и параметр `ob` должен относиться к типу `Stats<Integer>`. Его нельзя применять, например, для сравнения среднего значения типа `Stats<Double>` со средним значением типа `Stats<Short>`. Таким образом, данное решение пригодно только в очень ограниченном контексте и не является общим, а следовательно, и обобщенным.

Чтобы создать обобщенную версию метода `sameAvg()`, следует воспользоваться другим средством обобщений Java — *метасимвольным аргументом*. Метасимвольный аргумент обозначается знаком `?` и представляет неизвестный тип. Применяя метасимвольный аргумент, метод `sameAvg()` можно написать, например, следующим образом:

```
// Определить равенство двух средних значений.
// Обратите внимание на применение метасимвола
boolean sameAvg(Stats<?> ob) {
    if(average() == ob.average())
        return true;

    return false;
}
```

где метасимвольный аргумент типа `Stats<?>` совпадает с любым объектом класса `Stats`, что позволяет сравнивать средние значения любых двух объектов класса `Stats`. В следующем примере программы демонстрируется такое применение метасимвольного аргумента.

```
// Применение метасимвола
class Stats<T extends Number> {
    T[] nums; // массив класса Number или его подкласса

    // передать конструктору ссылку на массив
    // элементов класса Number или его подкласса
    Stats(T[] o) {
        nums = o;
    }

    // вернуть значение типа double в любом случае
    double average() {
        double sum = 0.0;
    }
}
```

```
        for(int i=0; i < nums.length; i++)
            sum += nums[i].doubleValue();

        return sum / nums.length;
    }

    // Определить равенство двух средних значений.
    // Обратите внимание на применение метасимвола
    boolean sameAvg(Stats<?> ob) {
        if(average() == ob.average())
            return true;

        return false;
    }
}

// Продемонстрировать применение метасимвола
class WildcardDemo {
    public static void main(String args[]) {
        Integer inums[] = { 1, 2, 3, 4, 5 };
        Stats<Integer> iob = new Stats<Integer>(inums);
        double v = iob.average();
        System.out.println("Среднее значение iob равно " + v);

        Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
        Stats<Double> dob = new Stats<Double>(dnums);
        double w = dob.average();
        System.out.println("Среднее значение dob равно " + w);

        Float fnums[] = { 1.0F, 2.0F, 3.0F, 4.0F, 5.0F };
        Stats<Float> fob = new Stats<Float>(fnums);
        double x = fob.average();
        System.out.println("Среднее значение fob равно " + x);

        // выяснить, какие массивы имеют одинаковые средние значения
        System.out.print("Средние значения iob и dob ");
        if(iob.sameAvg(dob))
            System.out.println("равны.");
        else
            System.out.println("отличаются.");

        System.out.print("Средние iob и fob ");
        if(iob.sameAvg(fob))
            System.out.println("одинаковы.");
        else
            System.out.println("отличаются.");
    }
}
```

Ниже приведен результат выполнения данной программы.

```
Среднее значение iob равно 3.0
Среднее значение dob равно 3.3
Среднее значение fob равно is 3.0
Средние значения iob и dob отличаются.
Средние значения iob и fob одинаковы.
```

И еще одно, последнее замечание: следует иметь в виду, что метасимвол не оказывает никакого влияния на тип создаваемых объектов класса Stats. Это определяется оператором `extends` в объявлении класса Stats. А метасимвол просто совпадает в любом *достоверным* объектом класса Stats.

Ограниченные метасимвольные аргументы

Метасимвольные аргументы могут быть ограничены почти таким же образом, как и параметры типов. Ограничивать метасимвольный аргумент особенно важно при создании обобщенного типа, оперирующего иерархией классов. Чтобы это стало понятнее, обратимся к конкретному примеру. Рассмотрим следующую иерархию классов, инкапсулирующих координаты:

```
// Двумерные координаты
class TwoD {
    int x, y;

    TwoD(int a, int b) {
        x = a;
        y = b;
    }
}

// Трехмерные координаты
class ThreeD extends TwoD {
    int z;

    ThreeD(int a, int b, int c) {
        super(a, b);
        z = c;
    }
}

// Четырехмерные координаты
class FourD extends ThreeD {
    int t;

    FourD(int a, int b, int c, int d) {
        super(a, b, c);
        t = d;
    }
}
```

На вершине этой иерархии находится класс `TwoD`, инкапсулирующий двумерные координаты `XY`. Его наследует класс `ThreeD`, вводя третье измерение и образуя координаты `XYZ`. Класс `ThreeD` наследует от класса `FourD`, вводящего четвертое измерение (время) и порождая четырехмерные координаты.

Ниже приведен обобщенный класс `Coords`, хранящий массив координат.

```
// Этот класс хранит массив координатных объектов
class Coords<T extends TwoD> {
    T[] coords;

    Coords(T[] o) { coords = o; }
}
```

Обратите внимание на то, что в классе `Coords` задается параметр типа, ограниченный классом `TwoD`. Это означает, что любой массив, сохраняемый в объекте типа `Coords`, будет содержать объект класса `TwoD` или любого из его подклассов.

А теперь допустим, что требуется написать метод, выводящий координаты `X` и `Y` каждого элемента массива `coords` в объекте класса `Coords`. Это нетрудно сделать с помощью метасимвола, как показано ниже, поскольку все типы объектов класса `Coords` имеют, как минимум, пару координат (`X` и `Y`).

```
static void showXY(Coords<?> c) {  
    System.out.println("Координаты X Y:");  
    for(int i=0; i < c.coords.length; i++)  
        System.out.println(c.coords[i].x + " " + c.coords[i].y);  
    System.out.println();  
}
```

Класс `Coords` относится к ограниченному обобщенному типу, для которого указан класс `TwoD` в качестве верхней границы, поэтому все объекты, которые можно использовать для создания объекта типа `Coords`, будут массивами класса `TwoD` или наследуемых от него классов. Таким образом, метод `showXY()` может выводить содержимое любого объекта типа `Coords`.

Но что, если требуется создать метод, выводящий координаты `X`, `Y` и `Z` объекта типа `ThreeD` или `FourD`? Дело в том, что не все объекты типа `Coords` будут иметь три координаты, поскольку объект типа `Coords<TwoD>` будет иметь только координаты `X` и `Y`. Как же тогда написать метод, который будет выводить координаты `X`, `Y` и `Z` для объектов типа `Coords<ThreeD>` и `Coords<FourD>`, и в то же время не допустить использование этого метода с объектами типа `Coords<TwoD>`? Ответ заключается в использовании *ограниченных метасимвольных аргументов*.

Ограниченный метасимвол задает верхнюю или нижнюю границу для аргумента типа. Это позволяет ограничить типы объектов, которыми будет оперировать метод. Наиболее распространен метасимвол, который накладывает ограничение сверху и создается с помощью оператора `extends` почти так же, как и ограниченный тип.

Применяя ограниченные метасимволы, нетрудно создать метод, выводящий координаты `X`, `Y` и `Z` для объекта типа `Coords`, если эти три координаты действительно имеются у этого объекта. Например, приведенный ниже метод `showXYZ()` выводит координаты элементов, сохраняемых в объекте типа `Coords`, если эти элементы относятся к классу `ThreeD` (или к наследуемым от него классам).

```
static void showXYZ(Coords<? extends ThreeD> c) {  
    System.out.println("X Y Z Coordinates:");  
    for(int i=0; i < c.coords.length; i++)  
        System.out.println(c.coords[i].x + " " +  
                           c.coords[i].y + " " +  
                           c.coords[i].z);  
    System.out.println();  
}
```

Обратите внимание на то, что оператор `extends` введен в метасимвол при объявлении параметра `c`. Это означает, что метасимвол `?` может совпадать с любым типом, при условии, что он относится к классу `ThreeD` или наследуемому от него классу. Таким образом, оператор `extends` накладывает ограничение сверху на совпадение с метасимволом `?`. Вследствие этого ограничения метод `showXYZ()` может быть вызван со ссылками на объекты типа `Coords<ThreeD>` или `Coords<FourD>`, но не со ссылкой на объект типа `Coords<TwoD>`. Попытка вызвать метод `showXYZ()` со ссылкой на объект типа `Coords<TwoD>` приведет к ошибке во время компиляции. Тем самым обеспечивается типовая безопасность.

Ниже приведена полная версия программы, демонстрирующей действие ограниченных метасимвольных аргументов.

```

// Ограниченные метасимвольные аргументы

// Двумерные координаты
class TwoD {
    int x, y;

    TwoD(int a, int b) {
        x = a;
        y = b;
    }
}

// Трехмерные координаты
class ThreeD extends TwoD {
    int z;

    ThreeD(int a, int b, int c) {
        super(a, b);
        z = c;
    }
}

// Четырехмерные координаты
class FourD extends ThreeD {
    int t;

    FourD(int a, int b, int c, int d) {
        super(a, b, c);
        t = d;
    }
}

// Этот класс хранит массив координатных объектов
class Coords<T extends TwoD> {
    T[] coords;

    Coords(T[] o) { coords = o; }
}

// Продемонстрировать применение ограниченных метасимволов
class BoundedWildcard {
    static void showXY(Coords<?> c) {
        System.out.println("Координаты X Y:");
        for(int i=0; i < c.coords.length; i++)
            System.out.println(c.coords[i].x + " " +
                               c.coords[i].y);
        System.out.println();
    }

    static void showXYZ(Coords<? extends ThreeD> c) {
        System.out.println("Координаты X Y Z:");
        for(int i=0; i < c.coords.length; i++)
            System.out.println(c.coords[i].x + " " +
                               c.coords[i].y + " " +
                               c.coords[i].z);
        System.out.println();
    }

    static void showAll(Coords<? extends FourD> c) {
        System.out.println("Координаты X Y Z T:");
        for(int i=0; i < c.coords.length; i++)

```

```

        System.out.println(c.coords[i].x + " " +
        c.coords[i].y + " " +
        c.coords[i].z + " " +
        c.coords[i].t);
        System.out.println();
    }

    public static void main(String args[]) {
        TwoD td[] = {
            new TwoD(0, 0),
            new TwoD(7, 9),
            new TwoD(18, 4),
            new TwoD(-1, -23)
        };

        Coords<TwoD> tdlocs = new Coords<TwoD>(td);

        System.out.println("Содержимое объекта tdlocs.");
        showXY(tdlocs); // Верно, это тип TwoD
        // showXYZ(tdlocs); // Ошибка, это не тип ThreeD
        // showAll(tdlocs); // Ошибка, это не тип FourD

        // а теперь создать несколько объектов типа FourD
        FourD fd[] = {
            new FourD(1, 2, 3, 4),
            new FourD(6, 8, 14, 8),
            new FourD(22, 9, 4, 9),
            new FourD(3, -2, -23, 17)
        };

        Coords<FourD> fdlocs = new Coords<FourD>(fd);

        System.out.println("Содержимое объекта fdlocs.");
        // Здесь все верно
        showXY(fdlocs);
        showXYZ(fdlocs);
        showAll(fdlocs);
    }
}

```

Результат выполнения этой программы выглядит следующим образом:

Содержимое объекта tdlocs.

Координаты X Y:

```

0 0
7 9
18 4
-1 -23

```

Содержимое объекта fdlocs.

Координаты X Y:

```

1 2
6 8
22 9
3 -2

```

Координаты X Y Z:

```

1 2 3
6 8 14
22 9 4
3 -2 -23

```

```

Координаты X Y Z T:
1 2 3 4
6 8 14 8
22 9 4 9
3 -2 -23 17

```

Обратите внимание на следующие закомментированные строки:

```

// showXYZ(tdlocs); // Ошибка, это не тип ThreeD
// showAll(tdlocs); // Ошибка, это не тип FourD

```

Объект `tdlocs` относится к типу `Coords<TwoD>`, и его нельзя использовать для вызова метода `showXYZ()` или `showAll()`, поскольку этому препятствуют ограниченные метасимвольные аргументы в их объявлении. Чтобы убедиться в этом, попробуйте убрать комментарии из приведенных выше строк кода и скомпилировать данную программу. В итоге вы получите ошибку компиляции из-за несоответствия типов.

В общем, чтобы установить верхнюю границу для метасимвола, следует воспользоваться приведенной ниже формой метасимвольного выражения:

```
<? extends суперкласс>
```

Здесь *суперкласс* обозначает имя класса, который служит верхней границей. Не следует забывать, что это включающее выражение, а следовательно, класс, заданный в качестве верхней границы (т.е. *суперкласс*), также находится в пределах допустимых типов.

Имеется также возможность указать нижнюю границу для метасимвольного аргумента, введя оператор `super` в его объявление. Ниже приведена общая форма наложения ограничения на метасимвольный аргумент снизу.

```
<? super подкласс>
```

В данном случае допустимыми аргументами могут быть только те классы, которые являются суперклассами для указанного *подкласса*. Это исключающее выражение, поскольку оно не включает в себя заданный *подкласс*.

Создание обобщенного метода

Как было показано в предыдущих примерах, в методах обобщенного класса можно использовать параметр типа, а следовательно, они становятся обобщенными относительно параметра типа. Но можно объявить обобщенный метод, в котором непосредственно используется один или несколько параметров типа. Более того, можно объявить обобщенный метод, входящий в необобщенный класс.

Начнем рассмотрение обобщенных методов с конкретного примера. В приведенной ниже программе объявляется необобщенный класс `GenMethDemo`, а в нем — статический обобщенный метод `isIn()`. Этот метод определяет, является ли объект членом массива. Его можно применять к любому типу объектов и массивов, при условии, что массив содержит объекты, совместимые с типом искомого объекта.

```

// Продемонстрировать простой обобщенный метод
class GenMethDemo {

```

```
// определить, содержится ли объект в массиве
static <T extends Comparable<T>, V extends T>
    boolean isIn(T x, V[] y) {
    for(int i=0; i < y.length; i++)
        if(x.equals(y[i])) return true;

    return false;
}

public static void main(String args[]) {
    // применить метод isIn() для целых чисел
    Integer nums[] = { 1, 2, 3, 4, 5 };

    if(isIn(2, nums))
        System.out.println("Число 2 содержится в массиве nums");

    if(!isIn(7, nums))
        System.out.println("Число 7 отсутствует в массиве nums");

    System.out.println();

    // применить метод isIn() для символьных строк
    String strs[] = { "один", "два", "три",
                      "четыре", "пять" };

    if(isIn("два", strs))
        System.out.println("два содержится в массиве strs");

    if(!isIn("семь", strs))
        System.out.println("семь отсутствует в массиве strs");

    // Не скомпилируется! Типы должны быть совместимы
    // if(isIn("два", nums))
    // System.out.println("два содержится в массиве strs ");
}
```

Ниже приведен результат выполнения данной программы.

Число 2 содержится в массиве nums
Число 7 отсутствует массиве в nums

два содержится в массиве strs
семь отсутствует массиве в strs

Рассмотрим метод `isIn()` подробнее. Прежде всего обратите внимание на объявление этого метода в следующей строке кода:

```
static <T extends Comparable<T>, V extends T> boolean isIn(T x, V[] y) {
```

Параметр типа объявляется *до* типа, возвращаемого методом. Обратите также внимание на то, что тип `T` расширяет обобщенный тип `Comparable<T>`, где `Comparable` — это интерфейс, объявляемый в пакете `java.lang`. В классе, реализующем интерфейс `Comparable`, определяются объекты, которые могут быть упорядочены. Следовательно, указание интерфейса `Comparable` в качестве верхней границы гарантирует, что метод `isIn()` вполне применим к объектам, которые можно сравнивать. Интерфейс `Comparable` является обобщенным, а параметр его типа обозначает тип сравниваемых объектов. (Далее в этой главе будет показано, как создается обобщенный интерфейс.) Обратите далее внимание на то, что тип

V ограничен сверху типом T . Это означает, что тип V должен быть тем же типом, что и T , или же типом его подкласса. Такая взаимосвязь подразумевает, что метод `isIn()` может быть вызван только с совместимыми аргументами. И наконец, обратите внимание на то, что метод `isIn()` объявлен как статический, что позволяет вызывать его независимо ни от какого объекта. Следует, однако, иметь в виду, что обобщенные методы могут быть как статическими, так и нестатическими. Никаких ограничений на этот счет не существует.

А теперь обратите внимание на то, что метод `isIn()` вызывается из метода `main()` с нормальным синтаксисом вызовов, не требуя указывать аргументы типа. Дело в том, что типы аргументов различаются автоматически, а типы T и V соответственно подстраиваются. Например, в первом вызове этого метода.

```
if(isIn(2, nums))
```

первый аргумент относится к типу `Integer` (благодаря автоупаковке), поэтому вместо типа T подставляется тип `Integer`. Второй аргумент также относится к типу `Integer`, который подставляется вместо типа V . Во втором вызове данного метода оба аргумента относятся к типу `String`, который и подставляется вместо типов T и V .

Для вызовов большинства обобщенных методов, как правило, достаточно и выведения типов, но если требуется, то аргументы типа можно указать явно. В качестве примера ниже показано, как должен выглядеть первый вызов метода `isIn()`, если явно указаны оба аргумента типа.

```
GenMethDemo.<Integer, Integer>isIn(2, nums)
```

Очевидно, что явное указание аргументов типа в данном случае не дает никаких преимуществ. Более того, выведение типов в отношении методов было усовершенствовано в версии JDK 8. Таким образом, указывать аргументы типа явным образом требуется лишь в крайне редких случаях.

А теперь обратите внимание на приведенный ниже закомментированный код из рассматриваемой здесь программы.

```
// if(isIn("два", nums))
//     System.out.println("два содержится в массиве strs ");
```

Если убрать комментарии из этих строк кода, а затем попытаться скомпилировать данную программу, то возникнет ошибка. Дело в том, что параметр типа V ограничивается типом T в выражении `extends` из объявления параметра типа V . Это означает, что параметр типа V должен иметь тип T или же тип его подкласса. В данном же случае первый аргумент относится к типу `String`, а следовательно, второй аргумент также должен относиться к типу `String`, хотя на самом деле он относится к типу `Integer`, который не является производным от класса `String`. В итоге во время компиляции возникает ошибка несоответствия типов. Такая способность обеспечивать типовую безопасность является одним из самых главных преимуществ обобщенных методов.

Синтаксис, использованный для создания метода `isIn()`, можно обобщить. Ниже приведена общая синтаксическая форма обобщенного метода.

```
<список_параметров_типа> возвращаемый_тип
    для_метода (список_параметров) { // ...
```

В любом случае *список_параметров_типа* обозначает разделяемый запятыми список параметров типа. Обратите внимание на то, что в объявлении обобщенного метода список параметров типа предшествует возвращаемому типу.

Обобщенные конструкторы

Конструкторы также могут быть обобщенными, даже если их классы таковыми не являются. Рассмотрим в качестве примера следующую короткую программу:

```
// Использовать обобщенный конструктор
class GenCons {
    private double val;

    <T extends Number> GenCons(T arg) {
        val = arg.doubleValue();
    }

    void showval() {
        System.out.println("val: " + val);
    }
}

class GenConsDemo {
    public static void main(String args[]) {
        GenCons test = new GenCons(100);
        GenCons test2 = new GenCons(123.5F);

        test.showval();
        test2.showval();
    }
}
```

Эта программа выводит следующий результат:

```
val: 100.0
val: 123.5
```

В конструкторе `GenCons()` задается параметр обобщенного типа, который может быть производным от класса `Number`, поэтому конструктор `GenCons()` можно вызывать с любым числовым типом, включая `Integer`, `Float` или `Double`. И несмотря на то, что класс `GenCons` не является обобщенным, его конструктор обобщен.

Обобщенные интерфейсы

Помимо классов и методов, обобщенными можно объявлять интерфейсы. Обобщенные интерфейсы объявляются таким же образом, как и обобщенные классы. Ниже приведен характерный тому пример. В нем создается обобщенный интерфейс `MinMax`, где объявляются методы `min()` и `max()`, которые, как предполагается, должны возвращать минимальное и максимальное значения из некоторого множества объектов.

```
// Пример применения обобщенного интерфейса
// Обобщенный интерфейс MinMax для определения
```



```
// минимального и максимального значений
interface MinMax<T extends Comparable<T>> {
    T min();
    T max();
}

// реализовать обобщенный интерфейс MinMax
class MyClass<T extends Comparable<T>> implements MinMax<T> {
    T[] vals;

    MyClass(T[] o) { vals = o; }

    // вернуть минимальное значение из массива vals
    public T min() {
        T v = vals[0];

        for(int i=1; i < vals.length; i++)
            if(vals[i].compareTo(v) < 0) v = vals[i];

        return v;
    }

    // вернуть максимальное значение из массива vals
    public T max() {
        T v = vals[0];

        for(int i=1; i < vals.length; i++)
            if(vals[i].compareTo(v) > 0) v = vals[i];

        return v;
    }
}

class GenIFDemo {
    public static void main(String args[]) {
        Integer inums[] = {3, 6, 2, 8, 6};
        Character chs[] = {'b', 'r', 'p', 'w'};

        MyClass<Integer> iob = new MyClass<Integer>(inums);
        MyClass<Character> cob = new MyClass<Character>(chs);

        System.out.println("Максимальное значение в массиве inums: " +
            iob.max());
        System.out.println("Минимальное значение в массиве inums: " +
            iob.min());

        System.out.println("Максимальное значение в массиве chs: " +
            cob.max());
        System.out.println("Минимальное значение в массиве chs: " +
            cob.min());
    }
}
```

Ниже приведен результат выполнения данной программы.

```
Максимальное значение в массиве inums: 8
Минимальное значение в массиве inums: 2
Максимальное значение в массиве chs: w
Минимальное значение в массиве chs: b
```

Большую часть этой программы нетрудно понять, но некоторые ключевые моменты следует все же пояснить. Прежде всего обратите внимание на следующее объявление обобщенного интерфейса MinMax:

```
interface MinMax<T extends Comparable<T>> {
```

Как правило, обобщенный интерфейс объявляется таким же образом, как и обобщенный класс. В данном случае параметр типа `T` ограничивается сверху интерфейсом `Comparable`. Как пояснялось выше, интерфейс `Comparable` определен в пакете `java.lang` для целей сравнения объектов. Параметр его типа обозначает тип сравниваемых объектов.

Затем интерфейс `MinMax` реализуется в классе `MyClass`. Объявление класса `MyClass` выглядит следующим образом:

```
class MyClass<T extends Comparable<T>> implements MinMax<T> {
```

Обратите особое внимание, каким образом параметр типа `T` сначала объявляется в классе `MyClass`, а затем передается интерфейсу `MinMax`. Интерфейсу `MinMax` требуется тип класса, реализующего интерфейс `Comparable`, поэтому в объявлении класса, реализующего этот интерфейс (в данном случае — класса `MyClass`), должно быть наложено такое же ограничение. Более того, однажды наложенное ограничение уже не нужно повторять в операторе `implements`. В действительности это было бы даже неверно. Например, приведенная ниже строка неверна и не может быть скомпилирована. Однажды установленный параметр типа просто передается интерфейсу без последующих видоизменений.

```
// Неверно!
class MyClass<T extends Comparable<T>>
    implements MinMax<T extends Comparable<T>> {
```

Как правило, класс, реализующий обобщенный интерфейс, должен быть также обобщенным — по крайней мере, в тех случаях, когда он принимает параметр типа, передаваемый далее интерфейсу. Например, следующая попытка объявить класс `MyClass` приведет к ошибке:

```
class MyClass implements MinMax<T> { // Неверно!
```

В классе `MyClass` параметр типа не объявляется, поэтому передать его интерфейсу `MinMax` никак нельзя. В данном случае идентификатор параметра типа `T` просто неизвестен, и поэтому компилятор выдаст ошибку. Безусловно, если класс реализует конкретный тип обобщенного интерфейса, то реализующий класс не обязан быть обобщенным, как показано ниже.

```
class MyClass implements MinMax<Integer> { // Верно
```

Обобщенный интерфейс дает два преимущества. Во-первых, он может быть реализован для разных типов данных. И во-вторых, он позволяет наложить ограничения на типы данных, для которых он может быть реализован. В примере интерфейса `MinMax` вместо параметра типа `T` могут быть подставлены только типы классов, реализующих интерфейс `Comparable`.

Ниже приведена общая синтаксическая форма обобщенного интерфейса.

```
interface имя_интерфейса<список_параметров_типа> { // ...
```

Здесь *список_параметров_типа* обозначает разделяемый запятыми список параметров типа. Когда реализуется обобщенный интерфейс, следует указать аргументы типа, как показано ниже.

```
class имя_класса<список_параметров_типа>
    implements имя_интерфейса<список_аргументов_типа> {
```

Базовые типы и унаследованный код

Поддержка обобщений отсутствовала до версии JDK 5, но требовался какой-нибудь способ переноса старого кода, разработанного до появления обобщений. На момент написания этой книги существовал большой объем унаследованного кода, который должен оставаться функциональным и быть совместимым с обобщениями. Код, предшествующий обобщениям, должен иметь возможность нормально взаимодействовать с обобщенным кодом, а обобщенный код — с унаследованным.

Чтобы облегчить плавный переход к обобщениям, в Java допускается применять обобщенные классы без аргументов. Это приводит к созданию *базового типа* (иначе называемого “сырым”) для класса. Этот базовый тип оказывается совместимым с унаследованным кодом, где синтаксис обобщений неизвестен. Главный недостаток применения базового типа заключается в том, что при этом утрачивается типовая безопасность. В приведенном ниже примере базовый тип демонстрируется в действии.

```
// Продемонстрировать базовый тип
```

```
class Gen<T> {  
  
    T ob; // объявить объект типа T  
  
    // передать конструктору ссылку на объект типа T  
    Gen(T o) {  
        ob = o;  
    }  
  
    // вернуть объект ob  
    T getob() {  
        return ob;  
    }  
}
```

```
// Продемонстрировать применение базового типа
```

```
class RawDemo {  
    public static void main(String args[]) {  
  
        // создать объект типа Gen для целых чисел  
        Gen<Integer> iOb = new Gen<Integer>(88);  
  
        // создать объект типа Gen для символьных строк  
        Gen<String> strOb = new Gen<String>("Тест обобщений");  
  
        // создать объект базового типа Gen и присвоить ему  
        // значение типа Double  
        Gen raw = new Gen(new Double(98.6));  
  
        // Требуется приведение типов, поскольку тип неизвестен  
        double d = (Double) raw.getob();  
        System.out.println("Значение: " + d);  
  
        // Применение базовых типов может вызвать исключения во  
        // время выполнения. Ниже представлены некоторые тому примеры.  
        // Следующее приведение типов вызовет ошибку во время выполнения!  
        // int i = (Integer) raw.getob(); // ОШИБКА во время выполнения!  
  
        // Следующее присваивание нарушает типовую безопасность
```

```
strOb = raw; // Верно, но потенциально ошибочно
// String str = strOb.getob(); // ОШИБКА во время выполнения!

// Следующее присваивание также нарушает типовую безопасность
raw = iOb; // Верно, но потенциально ошибочно
// d = (Double) raw.getob(); // ОШИБКА во время выполнения!
}
}
```

С этой программой связано несколько интересных моментов. В следующем объявлении создается класс `Gen` базового типа:

```
Gen raw = new Gen(new Double(98.6));
```

Обратите внимание на то, что никаких аргументов типа в этом объявлении не указывается. По существу, в нем создается объект класса `Gen`, тип `T` которого заменяется типом `Object`.

Базовые типы не обеспечивают нужной безопасности. Это означает, что переменной базового типа можно присвоить ссылку на любой тип объектов класса `Gen`. Возможно и обратное: переменной конкретного типа `Gen` можно присвоить ссылку на объект базового типа `Gen`. Но обе операции потенциально небезопасны, поскольку они выполняются в обход механизма проверки типов.

Недостаток типовой безопасности иллюстрируется закомментированными строками кода в конце данной программы. Рассмотрим каждую из них.

```
// int i = (Integer) raw.getob(); // ОШИБКА во время выполнения!
```

В этой строке кода получается значение объекта `ob` из объекта `raw`, и это значение приводится к типу `Integer`. Дело в том, что объект `raw` содержит значение типа `Double` вместо целочисленного значения. Но этого нельзя обнаружить на стадии компиляции, поскольку тип объекта `raw` неизвестен. Следовательно, выполнение этой строки кода приведет к ошибке во время выполнения.

В следующем фрагменте кода переменной `strOb`, ссылающейся на объект типа `Gen<String>`, присваивается ссылка на объект типа `Gen`:

```
strOb = raw; // Верно, но потенциально ошибочно
// String str = strOb.getob(); // ОШИБКА во время выполнения!
```

Такое присваивание само по себе синтаксически верно, но сомнительно. Переменная `strOb` относится к типу `Gen<String>`, поэтому предполагается, что она содержит символьную строку. Но после присваивания объект, на который ссылается переменная `strOb`, содержит значение типа `Double`. Следовательно, во время выполнения, когда предпринимается попытка присвоить переменной `str` содержимое переменной `strOb`, происходит ошибка, поскольку объект, на который ссылается переменная `strOb`, теперь содержит значение типа `Double`. Таким образом, присваивание базовой ссылки обобщенной ссылке происходит, минуя механизм проверки типов.

Следующий фрагмент кода представляет совершенно противоположный случай:

```
raw = iOb; // Верно, но потенциально ошибочно
// d = (Double) raw.getob(); // ОШИБКА во время выполнения!
```

где обобщенная ссылка присваивается переменной базовой ссылки. И хотя такое присваивание синтаксически верно, оно может также привести к ошибкам, как

показывает вторая строка кода. В данном случае переменная `raw` ссылается на объект, содержащий значение типа `Integer`, но в операции приведения типов предполагается, что он содержит значение типа `Double`. Такую ошибку нельзя предотвратить на стадии компиляции, поэтому она проявляется во время выполнения.

В связи с тем что базовые типы представляют опасность, по команде `javac` выводятся *непроверяемые предупреждения*, когда компилятор обнаруживает, что их применение способно нарушить типовую безопасность. Появление таких предупреждений вызовут следующие строки кода из рассматриваемой здесь программы:

```
Gen raw = new Gen(new Double(98.6));
```

```
strOb = raw; // Верно, но потенциально ошибочно
```

В первой строке кода происходит вызов конструктора `Gen()` без аргумента типа, что приводит к появлению предупреждения. А во второй строке базовая ссылка присваивается переменной обобщенной ссылки, что также приводит к появлению предупреждения.

На первый взгляд может показаться, что приведенная ниже строка кода также должна порождать предупреждение, но этого на самом деле не происходит.

```
raw = iOb; // Верно, но потенциально ошибочно
```

Здесь компилятор не выдает никаких предупреждений, потому что присваивание не вызывает никакой *дополнительной* потери типовой безопасности, кроме той, что уже произошла при создании объекта `raw`.

И еще одно, последнее замечание: применение базовых типов следует ограничивать теми случаями, когда унаследованный код приходится сочетать с новым, обобщенным кодом. Базовые типы служат лишь для переноса кода, а не средством, которое следует применять в новом коде.

Иерархии обобщенных классов

Обобщенные классы могут быть частью иерархии классов, как и любые другие необобщенные классы. Это означает, что обобщенный класс может действовать в качестве суперкласса или подкласса. Главное отличие обобщенных иерархий от необобщенных состоит в том, что в обобщенной иерархии любые аргументы типа, требующиеся обобщенному суперклассу, должны передаваться всеми подклассами вверх по иерархии. Это похоже на порядок передачи аргументов конструкторам вверх по иерархии.

Применение обобщенного суперкласса

Ниже приведен пример иерархии, в которой применяется обобщенный суперкласс.

```
// Простая иерархия обобщенных классов
class Gen<T> {
    T ob;

    Gen(T o) {
```

```

        ob = o;
    }

    // вернуть объект ob
    T getob() {
        return ob;
    }
}

// Подкласс, производный от класса Gen
class Gen2<T> extends Gen<T> {
    Gen2(T o) {
        super(o);
    }
}

```

В этой иерархии класс Gen2 расширяет обобщенный класс Gen. Обратите внимание на объявление класса Gen2 в следующей строке кода:

```
class Gen2<T> extends Gen<T> {
```

Параметр типа T указан в объявлении класса Gen2 и передается классу Gen в выражении extends. Это означает, что тип, передаваемый классу Gen2, будет также передан классу Gen. Например, в объявлении

```
Gen2<Integer> num = new Gen2<Integer>(100);
```

тип Integer передается в качестве параметра типа классу Gen. Таким образом, объект ob в части Gen класса Gen2 будет иметь тип Integer. Следует также иметь в виду, что параметр типа T используется в классе Gen2 только для поддержки его суперкласса Gen. Даже если подкласс обобщенного суперкласса совсем не обязательно должен быть обобщенным, в нем все же должны быть указаны параметры типа, требующиеся его обобщенному суперклассу.

Разумеется, подкласс может, если требуется, быть дополнен и своими параметрами типа. В качестве примера ниже показан вариант предыдущей иерархии, где в класс Gen2 вводится свой параметр типа.

```

// В подкласс могут быть введены свои параметры типа
class Gen<T> {
    T ob; // объявить объект типа T

    // передать конструктору ссылку на объект типа T
    Gen(T o) {
        ob = o;
    }

    // вернуть ссылку ob
    T getob() {
        return ob;
    }
}

// Подкласс, производный от класса Gen, где
// определяется второй параметр типа V
class Gen2<T, V> extends Gen<T> {
    V ob2;

    Gen2(T o, V o2) {
        super(o);
    }
}

```

```

        ob2 = o2;
    }

    V getob2() {
        return ob2;
    }
}
// создать объект типа Gen2
class HierDemo {
    public static void main(String args[]) {

        // создать объекты типа Gen2 для символьных строк целых чисел
        Gen2<String, Integer> x =
            new Gen2<String, Integer>("Значение равно: ", 99);

        System.out.print(x.getob());
        System.out.println(x.getob2());
    }
}

```

Обратите внимание на объявление класса Gen2, приведенное в следующей строке кода:

```
class Gen2<T, V> extends Gen<T> {
```

где T обозначает тип, передаваемый классу Gen, а V — тип, характерный для класса Gen2. Параметр типа V используется при объявлении объекта ob2, а также в качестве типа, возвращаемого методом getob2(). В методе main() создается объект класса Gen2, в котором тип String подставляется вместо параметра типа T, а тип Integer — вместо параметра типа V. Данная программа выдает следующий вполне ожидаемый результат:

Значение равно: 99

Обобщенный подкласс

Суперклассом для обобщенного класса вполне может служить и необобщенный класс. Рассмотрим в качестве примера следующую программу:

```

// Необобщенный класс может быть суперклассом
// для обобщенного подкласса

// Необобщенный класс
class NonGen {
    int num;

    NonGen(int i) {
        num = i;
    }

    int getnum() {
        return num;
    }
}

// Обобщенный подкласс
class Gen<T> extends NonGen {
    T ob; // объявить объект типа T
}

```

```
// передать конструктору объект типа T
Gen(T o, int i) {
    super(i);
    ob = o;
}

// вернуть объект ob
T getob() {
    return ob;
}

// создать объект типа Gen
class HierDemo2 {
    public static void main(String args[]) {

        // создать объект типа Gen для символьных строк
        Gen<String> w = new Gen<String>("Добро пожаловать", 47);

        System.out.print(w.getob() + " ");
        System.out.println(w.getnum());
    }
}
```

Ниже приведен результат выполнения данной программы.

Добро пожаловать 47

Обратите внимание, каким образом в этой программе класс Gen наследуется от класса NonGen:

```
class Gen<T> extends NonGen {
```

Класс NonGen является необобщенным, поэтому никаких аргументов типа в нем не указывается. И даже если в классе Gen объявляется параметр типа T, то он не требуется (и не может быть использован) в классе NonGen. Таким образом, класс Gen наследуется от класса NonGen обычным образом. Никаких специальных условий для этого не требуется.

Сравнение типов в обобщенной иерархии во время выполнения

Напомним, что для получения сведений о типе во время выполнения служит оператор `instanceof`, описанный в главе 13. Как пояснялось ранее, оператор `instanceof` определяет, является ли объект экземпляром класса. Он возвращает логическое значение `true`, если объект относится к указанному типу или может быть приведен к этому типу. Оператор `instanceof` можно применять к объектам обобщенных классов. В следующем примере класса демонстрируются некоторые последствия совместимости типов в обобщенных иерархиях:

```
// Использовать оператор instanceof в иерархии обобщенных классов
class Gen<T> {
    T ob;

    Gen(T o) {
        ob = o;
    }
}
```



```

    }

    // вернуть объект ob
    T getob() {
        return ob;
    }
}

// Подкласс, производный от класса Gen
class Gen2<T> extends Gen<T> {
    Gen2(T o) {
        super(o);
    }
}

// продемонстрировать последствия динамической идентификации
// типов в иерархии обобщенных классов
class HierDemo3 {
    public static void main(String args[]) {
        // создать объект типа Gen для целых чисел
        Gen<Integer> iOb = new Gen<Integer>(88);

        // создать объект типа Gen для целых чисел
        Gen2<Integer> iOb2 = new Gen2<Integer>(99);

        // создать объект типа Gen2 для символьных строк
        Gen2<String> strOb2 = new Gen2<String>("Тест обобщений");

        // проверить, является ли объект iOb2 какой-нибудь
        // из форм класса Gen2
        if(iOb2 instanceof Gen2<?>)
            System.out.println(
                "Объект iOb2 является экземпляром класса Gen2");

        // проверить, является ли объект iOb2 какой-нибудь
        // из форм класса Gen
        if(iOb2 instanceof Gen<?>)
            System.out.println(
                "Объект iOb2 является экземпляром класса Gen");
        System.out.println();

        // проверить, является ли strOb2 объектом класса Gen2
        if(strOb2 instanceof Gen2<?>)
            System.out.println(
                "Объект strOb2 является экземпляром класса Gen2");

        // проверить, является ли strOb2 объектом класса Gen
        if(strOb2 instanceof Gen<?>)
            System.out.println(
                "Объект strOb2 является экземпляром класса Gen");

        System.out.println();

        // проверить, является ли iOb экземпляром класса Gen2,
        // что совсем не так
        if(iOb instanceof Gen2<?>)
            System.out.println(
                "Объект iOb является экземпляром класса Gen2");

        // проверить, является ли iOb экземпляром класса Gen,
        // что так и есть
    }
}

```

```

    if(iOb instanceof Gen<?>)
        System.out.println(
            "Объект iOb является экземпляром класса Gen");

    // Следующий код не скомпилируется, так как сведения об
    // обобщенном типе отсутствуют во время выполнения
    // if(iOb2 instanceof Gen2<Integer>)
    // System.out.println(
    //     "Объект iOb2 является экземпляром класса Gen2<Integer>");
}
}

```

Ниже приведен результат выполнения данной программы.

```

Объект iOb2 является экземпляром класса Gen2
Объект iOb2 является экземпляром класса Gen

Объект strOb2 является экземпляром класса Gen2
Объект strOb2 является экземпляром класса Gen

Объект iOb является экземпляром класса Gen

```

В данной программе класс Gen2 является производным от класса Gen, который обобщен по параметру типа T. В методе main() создаются три объекта. Первый – объект iOb типа Gen<Integer>, второй – объект iOb2 типа Gen2<Integer>, третий – объект strOb2 типа Gen2<String>.

Затем в данной программе выполняются следующие проверки типа объекта iOb2 с помощью оператора instanceof:

```

// проверить, является ли объект iOb2 какой-нибудь из форм класса Gen2
if(iOb2 instanceof Gen2<?>)
    System.out.println("Объект iOb2 является экземпляром класса Gen2");

// проверить, является ли объект iOb2 какой-нибудь из форм класса Gen
if(iOb2 instanceof Gen<?>)
    System.out.println("Объект iOb2 является экземпляром класса Gen");

```

Как показывает результат выполнения данной программы, обе проверки завершаются успешно. В первом случае объект iOb2 проверяется на соответствие обобщенному типу Gen2<?>. Эта проверка завершается успешно, поскольку она просто подтверждает, что объект iOb2 относится какому-то из типов Gen2. С помощью метасимвола оператор instanceof определяет, относится ли объект iOb2 к какому-то из типов Gen2. Далее объект iOb проверяется на принадлежность типу суперкласса Gen<?>. И это верно, поскольку объект iOb2 является некоторой формой суперкласса Gen. В ряде последующих строк кода из метода main() выполняется та же самая последовательность проверок, но уже объекта strOb2 с выводом соответствующих результатов.

Далее объект iOb, являющийся экземпляром суперкласса Gen<Integer>, проверяется в следующих строках кода:

```

// проверить, является ли iOb экземпляром класса Gen2, что совсем не так
if(iOb instanceof Gen2<?>)
    System.out.println("Объект iOb является экземпляром класса Gen2");

// проверить, является ли iOb экземпляром класса Gen, что так и есть
if(iOb instanceof Gen<?>)
    System.out.println("Объект iOb является экземпляром класса Gen");

```

Первый условный оператор `if` возвращает логическое значение `false`, поскольку объект `iOb` не является ни одной из форм типа `Gen2`. Следующая проверка завершается успешно, потому что объект `iOb` относится к одному из типов `Gen`.

А теперь внимательно проанализируем следующие закомментированные строки кода:

```
// Следующий код не скомпилируется, так как сведения об
// обобщенном типе отсутствуют во время выполнения
// if(iOb2 instanceof Gen2<Integer>)
//     System.out.println(
//         "Объект iOb2 является экземпляром класса Gen2<Integer>");
```

Как следует из комментария, эти строки кода не компилируются, потому что в них предпринимается попытка сравнить объект `iOb2` с конкретным типом `Gen2` (в данном случае с типом `Gen2<Integer>`). Напомним, что во время выполнения всякие сведения об обобщенном типе отсутствуют. Таким образом, оператор `instanceof` ни коим образом не может выяснить, является ли объект `iOb2` экземпляром типа `Gen2<Integer>` или нет.

Приведение типов

Тип одного экземпляра обобщенного класса можно привести к другому только в том случае, если они совместимы и их аргументы типа одинаковы. Например, следующее приведение типов из предыдущего примера программы:

```
(Gen<Integer>) iOb2 // допустимо
```

вполне допустимо, потому что объект `iOb2` является экземпляром типа `Gen<Integer>`. А следующее приведение типов:

```
(Gen<Long>) iOb2 // недопустимо
```

недопустимо, поскольку объект `iOb2` не является экземпляром типа `Gen<Long>`.

Переопределение методов в обобщенном классе

Метод из обобщенного класса может быть переопределен, как и любой другой метод. Рассмотрим в качестве примера следующую программу, в которой переопределяется метод `getob()`:

```
// Переопределение обобщенного метода в обобщенном классе
class Gen<T> {
    T ob; // объявить объект типа T

    // передать конструктору ссылку на объект типа T
    Gen(T o) {
        ob = o;
    }

    // вернуть объект ob
    T getob() {
        System.out.print("Метод getob() из класса Gen: ");
        return ob;
    }
}
```

```
// Подкласс, производный от класса Gen и
// переопределяющий метод getob()
class Gen2<T> extends Gen<T> {

    Gen2(T o) {
        super(o);
    }

    // переопределить метод getob()
    T getob() {
        System.out.print("Метод getob() из класса Gen2: ");
        return ob;
    }
}

// продемонстрировать переопределение обобщенных методов
class OverrideDemo {
    public static void main(String args[]) {

        // создать объект типа Gen для целых чисел
        Gen<Integer> iOb = new Gen<Integer> (88);

        // создать объект типа Gen2 для целых чисел
        Gen2<Integer> iOb2 = new Gen2<Integer> (99);

        // создать объект типа Gen2 для символьных строк
        Gen2<String> strOb2 = new Gen2<String> ("Тест обобщений");
        System.out.println(iOb.getob());
        System.out.println(iOb2.getob());
        System.out.println(strOb2.getob());
    }
}
```

Ниже приведен результат выполнения данной программы.

```
Метод getob() из класса Gen: 88
Метод getob() из класса Gen2: 99
Метод getob() из класса Gen2: Тест обобщений
```

Как подтверждает полученный результат, переопределенная версия метода `getob()` вызывается для объекта типа `Gen2`, тогда как для объектов типа `Gen` вызывается его версия из суперкласса.

Выведение типов и обобщения

Начиная с версии JDK 7 можно использовать сокращенный синтаксис для создания экземпляра обобщенного типа. Для начала рассмотрим следующий обобщенный класс:

```
class MyClass<T, V> {
    T ob1;
    V ob2;

    MyClass(T o1, V o2) {
        ob1 = o1;
        ob2 = o2;
    }
}
```

```
// ...
}
```

До версии JDK 7 для получения экземпляра класса MyClass пришлось бы воспользоваться оператором, аналогичным следующему:

```
MyClass<Integer, String> mcOb =
    new MyClass<Integer, String>(98, "Строка");
```

где аргументы типа (Integer и String) определяются дважды: когда объявляется объект mcOb и когда экземпляр класса MyClass создается с помощью оператора new. Со времени появления обобщений в версии JDK 5 эта форма была обязательна для всех версий языка Java вплоть до JDK 7. Хотя в этой форме нет ничего неправильного, она, по существу, несколько более многословна, чем требуется. В операторе new тип аргументов типа может быть без особого труда выведен из типа объекта mcOb, поэтому указывать их во второй раз на самом деле нет никаких причин. Чтобы разрешить эту ситуацию, в версии JDK 7 был внедрен синтаксический элемент, позволяющий избежать повторного указания аргументов типа.

Теперь приведенное выше объявление может быть переписано следующим образом:

```
MyClass<Integer, String> mcOb = new MyClass<>(98, "Строка");
```

Обратите внимание на то, что в правой части приведенного выше оператора, где создается экземпляр, просто указываются угловые скобки (<>), обозначающие пустой список аргументов типа и называемые *ромбовидным оператором*. Этот оператор предписывает компилятору вывести тип аргументов, требующихся конструктору в операторе new. Главное преимущество синтаксиса вывода типов заключается в том, что он короче и иногда значительно сокращает очень длинные операторы объявления.

Обобщим все сказанное выше. Когда выполняется вывод типов, синтаксис объявления для обобщенной ссылки и создания экземпляра имеет приведенную ниже общую форму, где список аргументов типа конструктора в операторе new пуст.

```
имя_класса<список_аргументов_типа> имя_переменной =
    new имя_класса<>(список_аргументов_конструктора);
```

Выведение типов можно также выполнять и при передаче параметров. Так, если в класс MyClass вводится следующий метод:

```
boolean isSame(MyClass<T, V> o) {
    if(ob1 == o.ob1 && ob2 == o.ob2) return true;
    else return false;
}
```

то приведенный ниже вызов считается вполне допустимым.

```
if(mcOb.isSame(new MyClass<>(1, "test"))) System.out.println("Same");
```

В данном случае аргументы типа для аргумента, передаваемого методу isSame(), могут быть выведены из параметров типа.

В связи с тем что синтаксис вывода типов был внедрен в версии JDK 7 и не поддерживается устаревшими компиляторами, в последующих примерах программ будет использоваться полный синтаксис объявления экземпляров обоб-

щенных классов, чтобы эти примеры можно было откомпилировать любым компилятором Java, поддерживающим обобщения. Кроме того, употребление полного синтаксиса позволяет яснее понять, что именно создается, а это очень важно для понимания исходного кода примеров, представленных в данной книге. Но при написании прикладного кода употребление синтаксиса вывода типов значительно упростит объявления.

Стирание

Обычно подробно знать, каким образом компилятор Java преобразует исходный текст программы в объектный код, не нужно. Но что касается обобщений, то некоторое общее представление об этом процессе все же следует иметь, поскольку оно объясняет принцип действия механизма обобщений, а также причины, по которым он иногда ведет себя не совсем обычно. Поэтому ниже описывается вкратце, каким образом обобщения реализованы в языке Java.

Важное ограничение, которое было наложено на способ реализации обобщений в Java, состояло в том, что требовалось обеспечить совместимость с предыдущими версиями Java. Проще говоря, обобщенный код должен быть совместим с прежним кодом, существовавшим до появления обобщений. Это означает, что любые изменения в синтаксисе языка Java или виртуальной машине JVM должны вноситься, не нарушая старый код. Способ, которым обобщения реализуются в Java для удовлетворения этому требованию, называется *стиранием*.

Рассмотрим в общих чертах принцип действия стирания. При компиляции прикладного кода Java все сведения об обобщенных типах удаляются (стираются). Это означает, что параметры типа сначала заменяются их ограничивающим типом, которым является тип `Object`, если никакого явного ограничения не указано. Затем выполняется требуемое приведение типов, определяемое аргументами типа, для обеспечения совместимости с типами, указанными в этих аргументах. Компилятор также обеспечивает эту совместимость типов. Такой подход к обобщениям означает, что никаких сведений о типах во время выполнения не существует. Это просто механизм автоматической обработки исходного кода.

Мостовые методы

Иногда компилятору приходится вводить в класс так называемый *мостовой метод* в качестве выхода из положения в тех случаях, когда результат стирания типов в перегружаемом методе из подкласса не совпадает с тем, что получается при стирании в аналогичном методе из суперкласса. В этом случае создается метод, который использует стирание типов в суперклассе и вызывает соответствующий метод из подкласса с указанным стиранием типов. Безусловно, мостовые методы появляются только на уровне байт-кода, они недоступны для программиста и не могут быть вызваны непосредственно.

Несмотря на то что программистам обычно не приходится иметь дело с мостовыми методами, поскольку они им не особенно нужны, все же полезно рассмотреть ситуацию, в которой они создаются. Рассмотрим следующий пример программы:

```
// Ситуация, в которой создается мостовой метод
class Gen<T> {
    T ob; // объявить объект типа T

    // передать конструктору ссылку на объект типа T
    Gen(T o) {
        ob = o;
    }

    // вернуть объект ob
    T getob() {
        return ob;
    }
}

// Подкласс, производный от класса Gen
class Gen2 extends Gen<String> {

    Gen2(String o) {
        super(o);
    }

    // перегрузить метод getob() для получения символьных строк
    String getob() {
        System.out.print("Вызван метод String getob(): ");
        return ob;
    }
}

// продемонстрировать ситуацию, когда требуется мостовой метод
class BridgeDemo {
    public static void main(String args[]) {

        // создать объект типа Gen2 для символьных строк
        Gen2 strOb2 = new Gen2("Тест обобщений");

        System.out.println(strOb2.getob());
    }
}
```

В этой программе класс `Gen2` расширяет класс `Gen`, но делает это с помощью специальной строковой версии класса `Gen`, как показывает следующее объявление:

```
class Gen2 extends Gen<String> {
```

Более того, в классе `Gen2` переопределяется метод `getob()` с возвращаемым типом `String`, как показано ниже.

```
// перегрузить метод getob() для получения символьных строк
String getob() {
    System.out.print("Вызван метод String getob(): ");
    return ob;
}
```

Все это вполне допустимо. Единственное затруднение состоит в том, что из-за стирания типов ожидаемая форма метода `getob()` будет выглядеть следующим образом:

```
Object getob() { // ...
```

Чтобы разрешить это затруднение, компилятор создает мостовой метод с упомянутой выше сигнатурой, который вызывает строковый вариант метода `getob()`. Так, если исследовать интерфейс класса `Gen2` по команде `javap`, то в нем можно обнаружить следующие методы:

```
class Gen2 extends Gen<java.lang.String> {
    Gen2(java.lang.String);
    java.lang.String getob();
    java.lang.Object getob(); // мостовой метод
}
```

Как видите, в их число входит и мостовой метод. (Комментарий добавлен автором, а не командой `javap`, а результат ее выполнения может меняться в зависимости от используемой версии Java.)

И наконец, обратите внимание на то, что оба варианта метода `getob()` отличаются лишь типом возвращаемого значения. Обычно это приводит к ошибке, но поскольку происходит не в исходном коде, то никаких осложнений не возникает, и виртуальная машина JVM находит правильный выход из данного положения.

Ошибки неоднозначности

Включение в язык обобщений породило *неоднозначность* — новый вид ошибок, от которых приходится защищаться. Ошибки неоднозначности происходят, когда стирание типов приводит к тому, что два внешне разных объявления обобщений разрешают один и тот же стираемый тип, вызывая конфликт. Рассмотрим следующий пример программы, в которой выполняется перегрузка методов:

```
// Неоднозначность возникает в результате стирания
// типов перегружаемых методов
class MyGenClass<T, V> {
    T ob1;
    V ob2;

    // ...

    // Эти два перегружаемых метода неоднозначны и не компилируются
    void set(T o) {
        ob1 = o;
    }

    void set(V o) {
        ob2 = o;
    }
}
```

Обратите внимание на то, что в классе `MyGenClass` объявляются два обобщенных типа — `T` и `V`. В классе `MyGenClass` предпринимается попытка перегрузить метод `set()`, исходя из параметров типа `T` и `V`. На первый взгляд это вполне обоснованно, потому что параметры `T` и `V` вроде бы относятся к разным типам. Но именно здесь возникают две проблемы неоднозначности.

Первая проблема состоит в следующем: судя по тому, как написан класс `MyGenClass`, ничто не требует, чтобы параметры `T` и `V` относились к разным ти-

пам. В принципе объект класса `MyGenClass` вполне возможно создать, например, следующим образом:

```
MyGenClass<String, String> obj = new MyGenClass<String, String>()
```

В этом случае оба параметра типа `T` и `V` будут заменены типом `String`. Благодаря этому оба варианта метода `set()` оказываются одинаковыми, что, безусловно, считается ошибкой.

Вторая, более фундаментальная проблема состоит в том, что стирание типов приводит оба варианта метода `set()` к следующей форме:

```
void set(Object o) { // ...
```

Это означает, что перегрузка метода `set()`, которую пытается осуществить класс `MyGenClass`, в действительности неоднозначна.

Ошибки неоднозначности трудно исправить. Так, если заранее известно, что параметр типа `V` всегда будет получать некоторый подтип класса `Number`, то можно попытаться исправить класс `MyGenClass`, переписав его объявление следующим образом:

```
class MyGenClass<T, V extends Number> { // почти верно!
```

Это исправление позволит скомпилировать класс `MyGenClass`, и можно будет даже получать экземпляры его объектов так, как показано ниже.

```
MyGenClass<String, Number> x = new MyGenClass<String, Number>();
```

Такой код вполне работоспособен, поскольку в Java можно точно определить, какой именно метод следует вызвать. Но неоднозначность возникнет при попытке выполнить следующую строку кода:

```
MyGenClass<Number, Number> x = new MyGenClass<Number, Number>();
```

Если параметры типа `T` и `V` получают тип `Number`, то какой именно вариант метода `set()` следует вызвать? Вызов метода `set()` теперь неоднозначен.

Откровенно говоря, в предыдущем примере было бы лучше объявить два метода с разными именами, вместо того чтобы перегружать метод `set()`. Разрешение неоднозначности зачастую требует реструктуризации кода, потому что неоднозначность свидетельствует о принципиальной проектной ошибке.

Некоторые ограничения, присущие обобщениям

Имеются некоторые ограничения, о которых следует помнить, применяя обобщения. К их числу относится создание объектов по параметрам типа, статических членов, исключений и массивов. Каждое из этих ограничений рассматривается далее по отдельности.

Получить экземпляр по параметру типа нельзя

Создать экземпляр по параметру типа нельзя. Рассмотрим в качестве примера такой класс:

```
// Создать экземпляр типа T нельзя
class Gen<T> {
    T ob;

    Gen() {
        ob = new T(); // Недопустимо!!!
    }
}
```

В данном примере предпринимается недопустимая попытка создать экземпляр типа *T*. Причину недопустимости такой операции нетрудно понять: компилятору неизвестен тип создаваемого объекта. Ведь параметр типа *T* — это просто место для подстановки конкретного типа.

Ограничения на статические члены

Ни в одном из статических членов нельзя использовать параметр типа, объявляемый в его классе. Например, оба статических члена следующего класса недопустимы:

```
class Wrong<T> {
    // Неверно, нельзя создать статические переменные типа T
    static T ob;

    // Неверно, ни в одном из статических методов нельзя
    // использовать параметр типа T
    static T getob() {
        return ob;
    }
}
```

Если объявить статические члены по параметру типа, объявленному в их классе, нельзя, то объявить статические обобщенные методы со своими параметрами типа все же можно, как демонстрировалось в примерах, представленных ранее в этой главе.

Ограничения на обобщенные массивы

Имеются два важных ограничения, накладываемые на обобщения и касающиеся массивов. Во-первых, нельзя создать экземпляр массива, тип элемента которого определяется параметром типа. И во-вторых, нельзя создать массив специфических для типа обобщенных ссылок. Оба эти ограничения демонстрируются в следующем кратком примере программы:

```
// Обобщения и массивы
class Gen<T extends Number> {
    T ob;

    T vals[]; // Верно!

    Gen(T o, T[] nums) {
        ob = o;

        // Этот оператор неверен
        // vals = new T[10]; // нельзя создать массив типа T
    }
}
```

```

        // Тем не менее этот оператор верен
        vals = nums; // можно присвоить ссылку существующему массиву
    }
}

class GenArrays {
    public static void main(String args[]) {
        Integer n[] = { 1, 2, 3, 4, 5 };

        Gen<Integer> iOb = new Gen<Integer>(50, n);

        // Нельзя создать массив специфических для типа обобщенных ссылок
        // Gen<Integer> gens[] = new Gen<Integer>[10]; // Неверно!

        // А это верно!
        Gen<?> gens[] = new Gen<?>[10]; // Верно!
    }
}

```

Как показывает данный пример программы, объявлять ссылку на массив типа *T* допустимо, как это сделано в следующей строке кода:

```
T vals[]; // Верно!
```

Тем не менее нельзя получить экземпляр массива типа *T*, как показано в приведенной ниже закомментированной строке кода.

```
// vals = new T[10]; // нельзя создать массив объектов типа T
```

Создать массив типа *T* нельзя потому, что тип *T* не существует во время выполнения, а следовательно, компилятор не в состоянии выяснить, массив элементов какого типа требуется в действительности создавать.

Тем не менее конструктору *Gen()* можно передать ссылку на совместимый по типу массив при создании объекта и присвоить эту ссылку переменной *vals*, как это делается в приведенной ниже строке кода.

```
vals = nums; // можно присвоить ссылку существующему массиву
```

Это вполне допустимо, поскольку массив, передаваемый классу *Gen*, имеет известный тип, совпадающий с типом *T* на момент создания объекта. Следует, однако, иметь в виду, что в теле метода *main()* нельзя объявить массив ссылок на объекты конкретного обобщенного типа. Так, следующая строка кода не подлежит компиляции:

```
// Gen<Integer> gens[] = new Gen<Integer>[10]; // Неверно!
```

Тем не менее массив ссылок на обобщенный тип *можно* создать, если воспользоваться метасимволом, как показано ниже. Поступить так лучше, чем применять массивы базовых типов. Ведь, по крайней мере, некоторые проверки типов могут быть по-прежнему выполнены компилятором.

```
Gen<?> gens[] = new Gen<?>[10]; // Верно!
```

Ограничения на обобщенные исключения

Обобщенный класс не может расширять класс *Throwable*. Это означает, что создать обобщенные классы исключений нельзя.

В ходе непрерывно продолжающейся разработки и развития в Java были внедрены многие языковые средства, начиная с исходной версии 1.0. Но два из них следует выделить особо, поскольку они коренным образом изменили этот язык, а следовательно, и порядок написания на нем кода. Первым из этих нововведений стали обобщения, внедренные в версии JDK 5 и рассмотренные в главе 14, а вторым — лямбда-выражения, которым посвящена настоящая глава.

Лямбда-выражения и связанные с ними средства, внедренные в версии JDK 8, значительно усовершенствовали язык Java по следующим причинам. Во-первых, они вводят в синтаксис новые элементы, повышающие выразительную силу языка. Они, по существу, упрощают порядок реализации некоторых общих языковых конструкций. И во-вторых, внедрение лямбда-выражений позволило разделить новым возможностям библиотеку прикладного программного интерфейса API. К их числу относится возможность упростить параллельную обработку в многоядерных средах, особенно в циклических операциях, выполняемых в стиле `for each`, а также в новом прикладном программном интерфейсе API потоков ввода-вывода, где поддерживаются конвейерные операции с данными. Внедрение лямбда-выражений послужило также катализатором для внедрения в Java других новых средств, включая методы по умолчанию, рассмотренные в главе 9 и позволяющие определить стандартное поведение интерфейсного метода, а также описываемые в этой главе ссылки на методы, позволяющие ссылаться на метод, не выполняя его.

Помимо преимуществ, которые лямбда-выражения вносят в Java, имеется еще одна причина, по которой они являются очень важным дополнением этого языка программирования. За последние несколько лет лямбда-выражения стали предметом главного внимания в области разработки языков программирования. В частности, они были внедрены в такие языки, как C# и C++. А их включение в состав версии JDK 8 позволило сохранить живой, новаторский характер языка Java, к которому уже привыкли те, кто на нем программирует.

В конечном итоге лямбда-выражения изменили современный вид языка Java таким же образом, как это сделали обобщения несколько лет назад. Проще говоря, внедрение лямбда-выражений окажет влияние практически на всех программирующих на Java. И поэтому они действительно являются очень важным нововведением в Java.

Введение в лямбда-выражения

Особое значение для ясного представления о том, каким образом лямбда-выражения реализованы в Java, имеют две языковые конструкции. Первой из них является само лямбда-выражение, а второй — функциональный интерфейс. Начнем с простого определения каждой из этих конструкций.

Лямбда-выражение, по существу, является анонимным (т.е. безымянным) методом. Но этот метод не выполняется самостоятельно, а служит для реализации метода, определяемого в функциональном интерфейсе. Таким образом, лямбда-выражение приводит к некоторой форме анонимного класса. Нередко лямбда-выражения называют также *замыканиями*.

Функциональным называется такой интерфейс, который содержит один и *только* один абстрактный метод. Как правило, в таком методе определяется предполагаемое назначение интерфейса. Следовательно, функциональный интерфейс представляет единственное действие. Например, стандартный интерфейс `Runnable` является функциональным, поскольку в нем определяется единственный метод `run()`, который, в свою очередь, определяет действие самого интерфейса `Runnable`. Кроме того, в функциональном интерфейсе определяется *целевой тип* лямбда-выражения. В связи с этим необходимо подчеркнуть следующее: лямбда-выражение можно использовать только в том контексте, в котором определен его целевой тип. И еще одно замечание: функциональный интерфейс иногда еще называют *SAM-типом*, где SAM обозначает *Single Abstract Method* — единственный абстрактный метод.

На заметку! В функциональном интерфейсе можно определить любой открытый метод, определенный в классе `Object`, например, метод `equals()`, не воздействуя на состояние его функционального интерфейса. Открытые методы из класса `Object` считаются неявными членами функционального интерфейса, поскольку они автоматически реализуются экземпляром функционального интерфейса.

Рассмотрим подробнее как лямбда-выражения, так и функциональные интерфейсы.

Основные положения о лямбда-выражениях

Лямбда-выражение вносит новый элемент в синтаксис и оператор в язык Java. Этот новый оператор называется *лямбда-оператором*, или операцией “стрелка” (`->`). Он разделяет лямбда-выражение на две части. В левой части указываются любые параметры, требующиеся в лямбда-выражении. (Если же параметры не требуются, то они указываются пустым списком.) А в правой части находится *тело лямбда-выражения*, где указываются действия, выполняемые лямбда-выражением. Операция `->` буквально означает “становиться” или “переходить”.

В Java определены две разновидности тел лямбда-выражений. Одна из них состоит из единственного выражения, а другая — из блока кода. Рассмотрим сначала лямбда-выражения, в теле которых определяется единственное выражение. А лямбда-выражения с блочными телами обсудим далее в этой главе.

Прежде чем продолжить дальше, имеет смысл обратиться к некоторым примерам лямбда-выражений. Рассмотрим сначала самое простое лямбда-выражение, какое только можно написать. В приведенном ниже лямбда-выражении вычисляется значение константы.

```
() -> 123.45
```

Это лямбда-выражение не принимает никаких параметров, а следовательно, список его параметров оказывается пустым. Оно возвращает значение константы **123, 45**. Следовательно, это выражение аналогично вызову следующего метода:

```
double myMeth() { return 123.45; }
```

Разумеется, метод, определяемый лямбда-выражением, не имеет имени. Ниже приведено более интересное лямбда-выражение.

```
() -> Math.random() * 100
```

В этом лямбда-выражении из метода `Math.random()` получается псевдослучайное значение, которое умножается на 100 и затем возвращается результат. И это лямбда-выражение не требует параметров. Если же лямбда-выражению требуются параметры, они указываются списком в левой части лямбда-оператора. Ниже приведен простой пример лямбда-выражения с одним параметром.

```
(n) -> (n % 2) == 0
```

Это выражение возвращает логическое значение `true`, если числовое значение параметра `n` оказывается четным. Тип параметра (в данном случае `n`) можно указывать явно, но зачастую в этом нет никакой нужды, поскольку его тип в большинстве случаев выводится. Как и в именованном методе, в лямбда-выражении можно указывать столько параметров, сколько требуется.

Функциональные интерфейсы

Как пояснялось ранее, функциональным называется такой интерфейс, в котором определяется *единственный* абстрактный метод. Те, у кого имеется предыдущий опыт программирования на Java, могут возразить, что все методы интерфейса неявно считаются абстрактными, но так было до внедрения лямбда-выражений. Как пояснялось в главе 9, начиная с версии JDK 8 для метода, объявляемого в интерфейсе, можно определить стандартное поведение по умолчанию, и поэтому он называется *методом по умолчанию*. Отныне интерфейсный метод считается абстрактным лишь в том случае, если у него отсутствует реализация по умолчанию. А поскольку интерфейсные методы, не определяемые по умолчанию, неявно считаются абстрактными, то их не обязательно объявлять с модификатором доступа `abstract`, хотя это и можно сделать при желании.

Ниже приведен пример объявления функционального интерфейса.

```
interface MyNumber {  
    double getValue();  
}
```

В данном случае метод `getValue()` неявно считается абстрактным и единственным определяемым в интерфейсе `MyNumber`. Следовательно, интерфейс `MyNumber` является функциональным, а его функция определяется методом `getValue()`.

Как упоминалось ранее, лямбда-выражение не выполняется самостоятельно, а скорее образует реализацию абстрактного метода, определенного в функциональном интерфейсе, где указывается его целевой тип. Таким образом, лямбда-выражение может быть указано только в том контексте, в котором определен его целевой тип. Один из таких контекстов создается в том случае, когда лямбда-выражение присваивается ссылке на функциональный интерфейс. К числу других контекстов целевого типа относятся инициализация переменных, операторы `return` и аргументы методов.

Рассмотрим пример, демонстрирующий применение лямбда-выражения в контексте присваивания. С этой целью сначала объявляется ссылка на функциональный интерфейс `MyNumber`, как показано ниже.

```
// создать ссылку на функциональный интерфейс MyNumber  
MyNumber myNum;
```

Затем лямбда-выражение присваивается этой ссылке на функциональный интерфейс следующим образом:

```
// использовать лямбда-выражение в контексте присваивания  
myNum = () -> 123.45;
```

Когда лямбда-выражение появляется в контексте своего целевого типа, автоматически создается экземпляр класса, реализующего функциональный интерфейс, причем лямбда-выражение определяет поведение абстрактного метода, объявляемого в функциональном интерфейсе. А когда этот метод вызывается через свой адресат, выполняется лямбда-выражение. Таким образом, лямбда-выражение позволяет преобразовать сегмент кода в объект.

В предыдущем примере лямбда-выражение становится реализацией метода `getValue()`. В итоге получается значение константы **123,45**, которое выводится на экран следующим образом:

```
// вызвать метод getValue(), реализуемый  
// присвоенным ранее лямбда-выражением  
System.out.println("myNum.getValue());
```

Лямбда-выражение было ранее присвоено переменной `myNum` ссылки на функциональный интерфейс `MyNumber`. Оно возвращает значение константы **123,45**, которое получается в результате вызова метода `getValue()`.

Для того чтобы лямбда-выражение использовалось в контексте своего целевого типа, абстрактный метод и лямбда-выражение должны быть совместимыми по типу. Так, если в абстрактном методе указываются два параметра типа `int`, то и в лямбда-выражении должны быть указаны два параметра, тип которых явно обозначается как `int` или неявно выводится как `int` из самого контекста. В общем, параметры лямбда-выражения должны быть совместимы по типу и количеству с параметрами абстрактного метода. Это же относится и к возвращаемым типам. А любые исключения, генерируемые в лямбда-выражении, должны быть приемлемы для абстрактного метода.

Некоторые примеры лямбда-выражений

Принимая во внимание все сказанное выше, рассмотрим ряд простых примеров, демонстрирующих основные принципы действия лямбда-выражений. В пер-

вом примере программы все приведенные ранее фрагменты кода собраны в единое целое:

```
// Продемонстрировать применение простого лямбда-выражения

// Функциональный интерфейс
interface MyNumber {
    double getValue();
}

class LambdaDemo {
    public static void main(String args[])
    {
        MyNumber myNum; // объявить ссылку на функциональный интерфейс

        // Здесь лямбда-выражение просто является константным выражением.
        // Когда оно присваивается ссылочной переменной myNum, получается
        // экземпляр класса, в котором лямбда-выражение реализует
        // метод getValue() из функционального интерфейса MyNumber
        myNum = () -> 123.45;

        // вызвать метод getValue(), предоставляемый
        // присвоенным ранее лямбда-выражением
        System.out.println("Фиксированное значение: " + myNum.getValue());

        // А здесь используется более сложное выражение
        myNum = () -> Math.random() * 100;

        // В следующих строках кода вызывается лямбда-выражение
        // из предыдущей строки кода
        System.out.println("Случайное значение: " + myNum.getValue());
        System.out.println(
            "Еще одно случайное значение: " + myNum.getValue());

        // Лямбда-выражение должно быть совместимо с абстрактным методом,
        // определяемым в функциональном интерфейсе. Поэтому следующая
        // строка кода ошибочна:
        // myNum = () -> "123.03"; // ОШИБКА!
    }
}
```

Ниже приведен результат, выводимый данной программой.

```
Фиксированное значение: 123.45
Случайное значение: 88.90663650412304
Еще одно случайное значение: 53.00582701784129
```

Как упоминалось ранее, лямбда-выражение должно быть совместимо по типу с абстрактным методом, для реализации которого оно предназначено. Именно поэтому последняя строка кода в приведенном выше примере закомментирована. Ведь значение типа `String` несовместимо с типом `double`, возвращаемым методом `getValue()`.

В следующем примере программы демонстрируется применение лямбда-выражения с параметром:

```
// Продемонстрировать применение лямбда-выражения,
// принимающего один параметр

// Еще один функциональный интерфейс
interface NumericTest {
```



```
boolean test(int n);
}

class LambdaDemo2 {
    public static void main(String args[])
    {
        // Лямбда-выражение, проверяющее, является ли число четным
        NumericTest isEven = (n) -> (n % 2)==0;

        if(isEven.test(10)) System.out.println("Число 10 четное");
        if(!isEven.test(9)) System.out.println("Число 9 нечетное");

        // А теперь воспользоваться лямбда-выражением, в котором
        // проверяется, является ли число неотрицательным
        NumericTest isNonNeg = (n) -> n >= 0;

        if(isNonNeg.test(1)) System.out.println("Число 1 неотрицательное");
        if(!isNonNeg.test(-1)) System.out.println("Число -1 отрицательное");
    }
}
```

Ниже приведен результат, выводимый данной программой.

```
Число 10 четное
Число 9 нечетное
Число 1 неотрицательное
Число -1 отрицательное
```

В данном примере программы демонстрируется главная особенность лямбда-выражений, требующая более подробного рассмотрения. Обратите особое внимание на лямбда-выражение, выполняющее проверку на равенство:

```
(n) -> (n % 2)==0
```

Обратите внимание на то, что тип переменной `n` не указан, но выводится из контекста. В данном случае тип переменной `n` выводится из типа `int` параметра метода `test()`, определяемого в функциональном интерфейсе `NumericTest`. Впрочем, ничто не мешает явно указать тип параметра в лямбда-выражении. Например, следующее лямбда-выражение так же достоверно, как и предыдущее:

```
(int n) -> (n % 2)==0
```

где параметр `n` явно указывается как `int`. Как правило, явно указывать тип параметров лямбда-выражений необязательно, хотя в некоторых случаях это может все же потребоваться.

В данном примере программы демонстрируется еще одна важная особенность лямбда-выражений. Ссылка на функциональный интерфейс может быть использована для выполнения любого совместимого с ней лямбда-выражения. Обратите внимание на то, что в данной программе определяются два разных лямбда-выражения, совместимых с методом `test()` из функционального интерфейса `NumericTest`. В первом лямбда-выражении `isNonNeg` проверяется, является ли числовое значение отрицательным, а во втором лямбда-выражении `isNonNeg` — является ли оно неотрицательным. Но в любом случае проверяется значение параметра `n`. А поскольку каждое из этих лямбда-выражений совместимо с методом `test()`, то оно выполняется по ссылке на функциональный интерфейс `NumericTest`.

Прежде чем продолжить дальше, следует сделать еще одно замечание. Если у лямбда-выражения имеется единственный параметр, его совсем не обязательно заключать в круглые скобки в левой части лямбда-оператора. Например, приведенный ниже способ написания лямбда-выражения также допустим в программах.

```
n -> (n % 2) == 0
```

Ради согласованности в представленных далее примерах программ списки параметров всех лямбда-выражений заключаются в круглые скобки — даже если они содержат единственный параметр. Разумеется, вы вольны выбрать тот способ указания параметров лямбда-выражений, который вам больше по душе.

В приведенном ниже примере программы демонстрируется лямбда-выражение, принимающее два параметра. В данном случае в лямбда-выражении проверяется, является ли одно число множителем другого.

```
// Продемонстрировать применение лямбда-выражения,
// принимающего два параметра

interface NumericTest2 {
    boolean test(int n, int d);
}

class LambdaDemo3 {
    public static void main(String args[])
    {
        // В этом лямбда-выражении проверяется, является ли
        // одно число множителем другого
        NumericTest2 isFactor = (n, d) -> (n % d) == 0;

        if(isFactor.test(10, 2))
            System.out.println("Число 2 является множителем числа 10");

        if(!isFactor.test(10, 3))
            System.out.println("Число 3 не является множителем числа 10");
    }
}
```

Ниже приведен пример выполнения данной программы.

Число 2 является множителем числа 10

Число 3 не является множителем числа 10

В данном примере программы метод `test()` определяется в функциональном интерфейсе `NumericTest2` следующим образом:

```
boolean test(int n, int d);
```

При объявлении метода `test()` указываются два параметра. Следовательно, в лямбда-выражении, совместимом с методом `test()`, следует также указать два параметра. Ниже показано, как это делается.

```
(n, d) -> (n % d) == 0
```

Оба параметра, `n` и `d`, указываются списком через запятую. Данный пример можно обобщить. Всякий раз, когда в лямбда-выражении требуется больше одного параметра, их следует указать списком через запятую, заключив в круглые скобки в левой части лямбда-оператора.

Следует, однако, иметь в виду, что если требуется явно объявить тип одного из параметров лямбда-выражения, то это следует сделать и для всех остальных параметров. Например, следующее лямбда-выражение достоверно:

```
(int n, int d) -> (n % d) == 0
```

А это лямбда-выражение недостоверно:

```
(int n, d) -> (n % d) == 0
```

Блочные лямбда-выражения

Тело лямбда-выражений в предыдущих примерах состояло из единственного выражения. Такая разновидность тел называется *телом выражения*, а лямбда-выражения с телом выражения иногда еще называют *одиночными*. В теле выражения код, указываемый в правой части лямбда-оператора, должен состоять из одного выражения. Несмотря на все удобство одиночных лямбда-выражений, иногда в них требуется вычислять не одно выражение. Для подобных случаев в Java предусмотрена вторая разновидность лямбда-выражений, где код, указываемый в правой части лямбда-оператора, может состоять из нескольких операторов. Такие лямбда-выражения называются *блочными*, а их тело — *телом блока*.

Блочное лямбда-выражение расширяет те виды операций, которые могут выполняться в лямбда-выражении, поскольку оно допускает в своем теле наличие нескольких операторов. Например, в блочном лямбда-выражении можно объявлять переменные, организовывать циклы, указывать операторы выбора `if` и `switch`, создавать вложенные блоки и т.д. Создать блочное лямбда-выражение совсем не трудно. Для этого достаточно заключить тело выражения в фигурные скобки таким же образом, как и любой другой блок кода.

Кроме наличия в теле выражения нескольких операторов, блочные лямбда-выражения применяются точно так же, как и упоминавшиеся ранее одиночные лямбда-выражения. Однако для возврата значения из блочных лямбда-выражений нужно явно указывать оператор `return`. Это нужно делать потому, что тело блочного лямбда-выражения не представляет одиночное выражение.

Ниже приведен пример программы, в котором блочное лямбда-выражение применяется для вычисления и возврата факториала целочисленного значения.

```
// Блочное лямбда-выражение, вычисляющее
// факториал целочисленного значения

interface NumericFunc {
    int func(int n);
}

class BlockLambdaDemo {
    public static void main(String args[])
    {
        // Это блочное лямбда-выражение вычисляет
        // факториал целочисленного значения
        NumericFunc factorial = (n) -> {
            int result = 1;
```

```

        for(int i=1; i <= n; i++)
            result = i * result;

        return result;
    };

    System.out.println("Факториал числа 3 равен " + factorial.func(3));
    System.out.println("Факториал числа 5 равен " + factorial.func(5));
}
}

```

Ниже приведен результат выполнения данной программы.

```

Факториал числа 3 равен 6
Факториал числа 5 равен 120

```

В данном примере программы обратите внимание на то, что в блочном лямбда-выражении объявляется переменная `result`, организуется цикл `for` и указывается оператор `return`. Все эти действия вполне допустимы в теле блочного лямбда-выражения. По существу, тело блока такого выражения аналогично телу метода. Следует также иметь в виду, что когда в лямбда-выражении оказывается оператор `return`, он просто вызывает возврат из самого лямбда-выражения, но не из обьющего его метода.

Ниже приведен еще один пример блочного лямбда-выражения. В данном примере программы изменяется на обратный порядок следования символов в строке.

```

// Блочное выражение, изменяющее на обратный
// порядок следования символов в строке

interface StringFunc {
    String func(String n);
}

class BlockLambdaDemo2 {
    public static void main(String args[])
    {

        // Это блочное выражение изменяет на обратный
        // порядок следования символов в строке
        StringFunc reverse = (str) -> {
            String result = "";
            int i;

            for(i = str.length()-1; i >= 0; i--)
                result += str.charAt(i);

            return result;
        };

        System.out.println("Лямбда обращается на " +
                           reverse.func("Лямбда"));
        System.out.println("Выражение обращается на " +
                           reverse.func("Выражение"));
    }
}

```

Ниже приведен результат выполнения данной программы.

```

Лямбда обращается на адбямл
Выражение обращается на еиненарыв

```

В данном примере программы в функциональном интерфейсе `StringFunc` объявляется метод `func()`, принимающий параметр типа `String` и возвращающий значение типа `String`. Следовательно, в лямбда-выражении `reverse` тип параметра `str` должен быть выведен как `String`. Обратите внимание на то, что метод `charAt()` вызывается для параметра `str` как для объекта. И это вполне допустимо, поскольку этот параметр имеет тип `String` благодаря выведению типов.

Обобщенные функциональные интерфейсы

Указывать параметры типа в самом лямбда-выражении нельзя. Следовательно, лямбда-выражение не может быть обобщенным. (Безусловно, все лямбда-выражения проявляют в той или иной мере свойства, подобные обобщениям, благодаря выведению типов.) А вот функциональный интерфейс, связанный с лямбда-выражением, может быть обобщенным. В этом случае целевой тип лямбда-выражения отчасти определяется аргументом типа или теми аргументами, которые указываются при объявлении ссылки на функциональный интерфейс.

Чтобы понять и оценить значение обобщенных функциональных интерфейсов, вернемся к двум примерам из предыдущего раздела. В них применялись два разных функциональных интерфейса: `NumericFunc` и `StringFunc`. Но в обоих этих интерфейсах был определен метод `func()`, возвращавший результат. В первом случае этот метод принимал параметр и возвращал значение типа `int`, а во втором случае — значение типа `String`. Следовательно, единственное отличие обоих вариантов этого метода состояло в типе требовавшихся данных. Вместо того чтобы объявлять два функциональных интерфейса, методы которых отличаются только типом данных, можно объявить один обобщенный интерфейс, который можно использовать в обоих случаях. Именно такой подход и принят в следующем примере программы:

```
// Применить обобщенный функциональный интерфейс
// с разнотипными лямбда-выражениями

// Обобщенный функциональный интерфейс
interface SomeFunc<T> {
    T func(T t);
}

class GenericFunctionalInterfaceDemo {
    public static void main(String args[])
    {
        // использовать строковый вариант интерфейса SomeFunc
        SomeFunc<String> reverse = (str) -> {
            String result = "";
            int i;

            for(i = str.length()-1; i >= 0; i--)
                result += str.charAt(i);

            return result;
        };
    }
}
```

```
System.out.println("Лямбда обращается на " +  
                    reverse.func("Лямбда"));  
System.out.println("Выражение обращается на " +  
                    reverse.func("Выражение"));  
  
// а теперь использовать целочисленный вариант интерфейса SomeFunc  
SomeFunc<Integer> factorial = (n) -> {  
    int result = 1;  
  
    for(int i=1; i <= n; i++)  
        result = i * result;  
  
    return result;  
};  
  
System.out.println("Факториал числа 3 равен " + factorial.func(3));  
System.out.println("Факториал числа 5 равен " + factorial.func(5));  
}
```

Ниже приведен результат выполнения данной программы.

```
Лямбда обращается на адбмял  
Выражение обращается на еинежарыв  
факториал числа 3 равен 6  
факториал числа 5 равен 120
```

В данном примере программы обобщенный функциональный интерфейс `SomeFunc` объявляется следующим образом:

```
interface SomeFunc<T> {  
    T func(T t);  
}
```

где `T` обозначает как возвращаемый тип, так и тип параметра метода `func()`. Это означает, что он совместим с любым лямбда-выражением, принимающим один параметр и возвращающим значение того же самого типа.

Обобщенный функциональный интерфейс `SomeFunc` служит для предоставления ссылки на два разных типа лямбда-выражений. В первом из них используется тип `String`, а во втором — тип `Integer`. Таким образом, один и тот же интерфейс может быть использован для обращения к обоим лямбда-выражениям — `reverse` и `factorial`. Отличается лишь аргумент типа, передаваемый обобщенному функциональному интерфейсу `SomeFunc`.

Передача лямбда-выражений в качестве аргументов

Как пояснялось ранее, лямбда-выражение может быть использовано в любом контексте, предоставляющем его целевой тип. Один из таких контекстов возникает при передаче лямбда-выражения в качестве аргумента. В действительности передача лямбда-выражений в качестве аргументов является весьма распространенным примером их применения. Более того, это весьма эффективное их применение, поскольку оно дает возможность передать исполняемый код методу в ка-

честве его аргумента. Благодаря этому значительно повышется выразительная сила языка Java.

Для передачи лямбда-выражения в качестве аргумента параметр, получающий это выражение в качестве аргумента, должен иметь тип функционального интерфейса, совместимого с этим лямбда-выражением. Несмотря на всю простоту применения лямбда-выражений в качестве передаваемых аргументов, полезно все же показать, как это происходит на практике. В следующем примере программы демонстрируется весь этот процесс:

```
// Передать лямбда-выражение в качестве аргумента методу

interface StringFunc {
    String func(String n);
}

class LambdasAsArgumentsDemo {

    // Первый параметр этого метода имеет тип функционального
    // интерфейса. Следовательно, ему можно передать ссылку на
    // любой экземпляр этого интерфейса, включая экземпляр,
    // создаваемый в лямбда-выражении. А второй параметр
    // обозначает обрабатываемую символьную строку
    static String stringOp(StringFunc sf, String s) {
        return sf.func(s);
    }

    public static void main(String args[])
    {
        String inStr = "Лямбда-выражения повышают эффективность Java";
        String outStr;

        System.out.println("Это исходная строка: " + inStr);

        // Ниже приведено простое лямбда-выражение, преобразующее
        // в верхний регистр букв все символы исходной строки,
        // передаваемой методу stringOp()
        outStr = stringOp((str) -> str.toUpperCase(), inStr);
        System.out.println("Это строка в верхнем регистре: " + outStr);

        // А здесь передается блочное лямбда-выражение, удаляющее
        // пробелы из исходной символьной строки
        outStr = stringOp((str) -> {
            String result = "";
            int i;

            for(i = 0; i < str.length(); i++)
                if(str.charAt(i) != ' ')
                    result += str.charAt(i);

            return result;
        }, inStr);

        System.out.println("Это строка с удаленными пробелами: " + outStr);

        // Конечно, можно передать и экземпляр интерфейса StringFunc,
        // созданный в предыдущем лямбда-выражении. Например, после
        // следующего объявления ссылка reverse делается на экземпляр
        // интерфейса StringFunc
        StringFunc reverse = (str) -> {
```

```

String result = "";
int i;

for(i = str.length()-1; i >= 0; i--)
    result += str.charAt(i);

return result;
};

// А теперь ссылку reverse можно передать в качестве первого
// параметра методу stringOp()
// since it refers to a StringFunc object.
System.out.println("Это обращенная строка: " +
    stringOp(reverse, inStr));
}
}

```

Ниже приведен результат выполнения данной программы.

Это исходная строка: Лямбда-выражения повышают эффективность Java
 Это строка в верхнем регистре: ЛЯМБДА-ВЫРАЖЕНИЯ ПОВЫШАЮТ ЭФФЕКТИВНОСТЬ JAVA
 Это строка с удаленными пробелами: Лямбда-выраженияповышаютэффективностьJava
 Это обращенная строка: яинежарыв-адбмяЛ тьюшывоп ьтсонвиткеффэ avaJ

Прежде всего обратите внимание в данном примере программы на метод `stringOp()`, у которого имеются два параметра. Первый параметр относится к типу `StringFunc`, т.е. к функциональному интерфейсу. Следовательно, этот параметр может получать ссылку на любой экземпляр функционального интерфейса `StringFunc`, в том числе и создаваемый в лямбда-выражении. А второй параметр метода, `stringOp()`, относится к типу `String` и обозначает обрабатываемую символьную строку.

Затем обратите внимание на первый вызов метода `stringOp()`:

```
outStr = stringOp((str) -> str.toUpperCase(), inStr);
```

где в качестве аргумента данному методу передается простое лямбда-выражение. При этом создается экземпляр функционального интерфейса `StringFunc` и ссылка на данный объект передается первому параметру метода `stringOp()`. Таким образом, код лямбда-выражения, встраиваемый в экземпляр класса, передается данному методу. Контекст целевого типа лямбда-выражения определяется типом его параметра. А поскольку лямбда-выражение совместимо с этим типом, то рассматриваемый здесь вызов достоверен. Встраивать в метод такие простые лямбда-выражения, как упомянутое выше, нередко оказывается очень удобно, особенно когда лямбда-выражение предназначается для однократного употребления.

Далее в рассматриваемом здесь примере программы методу `stringOp()` передается блочное лямбда-выражение. Оно удаляет пробелы из исходной символьной строки и еще раз показано ниже.

```

outStr = stringOp((str) -> {
    String result = "";
    int i;

    for(i = 0; i < str.length(); i++)
        if(str.charAt(i) != ' ')
            result += str.charAt(i);

```



```
        return result;
    }, inStr);
```

И хотя здесь указывается блочное лямбда-выражение, описанный выше процесс передачи лямбда-выражения остается тем же самым и для простого одиночного лямбда-выражения. Но в данном случае некоторым программистам синтаксис может показаться несколько неуклюжим.

Если блочное выражение кажется слишком длинным для встраивания в вызов метода, то его можно просто присвоить переменной ссылки на функциональный интерфейс, как это делалось в предыдущих примерах. И тогда остается только передать эту ссылку вызываемому методу. Такой прием показан в конце рассматриваемого здесь примера программы, где определяется блочное лямбда-выражение, изменяющее порядок следования символов в строке на обратный. Это лямбда-выражение присваивается переменной `reverse`, ссылающейся на функциональный интерфейс `StringFunc`. Следовательно, переменную `reverse` можно передать в качестве аргумента первому параметру метода `stringOp()`. Именно так и делается в конце данной программы, где методу `stringOp()` передаются переменная `reverse` и обрабатываемая символьная строка. Экземпляр, получаемый в результате вычисления каждого лямбда-выражения, является реализацией функционального интерфейса `StringFunc`, поэтому каждое из этих выражений может быть передано в качестве первого аргумента вызываемому методу `stringOp()`.

И последнее замечание: помимо инициализации переменных, присваивания и передачи аргументов, следующие операции образуют контекст целевого типа лямбда-выражений: приведение типов, тернарная операция `?`, инициализация массивов, операторы `return`, а также сами лямбда-выражения.

Лямбда-выражения и исключения

Лямбда-выражение может генерировать исключение. Но если оно генерирует проверяемое исключение, то последнее должно быть совместимо с исключениями, перечисленными в выражении `throws` из объявления абстрактного метода в функциональном интерфейсе. Эта особенность демонстрируется в приведенном ниже примере, где вычисляется среднее числовых значений типа `double` в массиве. А если лямбда-выражению передается массив нулевой длины, то генерируется исключение типа `EmptyArrayException`. Как следует из данного примера, это исключение перечислено в выражении `throws` из объявления метода `func()` в функциональном интерфейсе `DoubleNumericArrayFunc`.

```
// Сгенерировать исключение из лямбда-выражения

interface DoubleNumericArrayFunc {
    double func(double[] n) throws EmptyArrayException;
}

class EmptyArrayException extends Exception {
    EmptyArrayException() {
        super("Массив пуст");
    }
}
```

```
class LambdaExceptionDemo {  
  
    public static void main(String args[]) throws EmptyArrayException  
    {  
        double[] values = { 1.0, 2.0, 3.0, 4.0 };  
  
        // В этом лямбда-выражении вычисляется среднее числовых  
        // значений типа double в массиве  
        DoubleNumericArrayFunc average = (n) -> {  
            double sum = 0;  
  
            if(n.length == 0)  
                throw new EmptyArrayException();  
  
            for(int i=0; i < n.length; i++)  
                sum += n[i];  
  
            return sum / n.length;  
        };  
  
        System.out.println("Среднее равно " + average.func(values));  
  
        // Эта строка кода приводит к генерированию исключения  
        System.out.println("Среднее равно " + average.func(new double[0]));  
    }  
}
```

В результате первого вызова метода `average.func()` возвращается среднее значение **2,5**. А при втором вызове этому методу передается массив нулевой длины, что приводит к генерированию исключения типа `EmptyArrayException`. Напомним, что наличие выражения `throws` в объявлении метода `func()` обязательно. Без этого программа не будет скомпилирована, поскольку лямбда-выражение перестанет быть совместимым с методом `func()`.

Данный пример демонстрирует еще одну важную особенность лямбда-выражений. Обратите внимание на то, что параметр, указываемый при объявлении метода `func()` в функциональном интерфейсе `DoubleNumericArrayFunc`, обозначает массив, тогда как параметр лямбда-выражения просто указан как `n`, а не `n[]`. Напомним, что тип параметра лямбда-выражения выводится из целевого контекста. В данном случае целевым контекстом является массив типа `double[]`, поэтому и параметр `n` относится к типу `double[]`. Следовательно, указывать этот параметр как `n[]` совсем не обязательно и даже недопустимо. И хотя его можно было бы явно указать как `double[] n`, это не дало бы в данном случае никаких преимуществ.

Лямбда-выражения и захват переменных

Переменные, определяемые в объемлющей области действия лямбда-выражения, доступны в этом выражении. Например, в лямбда-выражении можно использовать переменную экземпляра или статическую переменную, определяемую в объемлющем его классе. В лямбда-выражении доступен также по ссылке `this` (явно или неявно) вызывающий экземпляр объемлющего его класса. Таким образом, в лямбда-

выражении можно получить или установить значение переменной экземпляра или статической переменной и вызвать метод из объемлющего его класса.

Но если в лямбда-выражении используется локальная переменная из объемлющей его области действия, то возникает особый случай, называемый *захватом переменной*. В этом случае в лямбда-выражении можно использовать только те локальные переменные, которые *действительно* являются *завершенными*. Действительно завершенной считается такая переменная, значение которой не изменяется после ее первого присваивания. Такую переменную совсем не обязательно объявлять как `final`, хотя это и не считается ошибкой. (Параметр `this` в объемлющей области действия автоматически оказывается действительно завершенным, а у лямбда-выражений собственный параметр `this` отсутствует.)

Следует, однако, иметь в виду, что локальная переменная из объемлющей области действия не может быть видоизменена в лямбда-выражении. Ведь это нарушило бы ее действительно завершенное состояние, а следовательно, привело бы к недопустимому ее захвату.

В следующем примере программы демонстрируется отличие действительно конечных переменных от изменяемых локальных переменных:

// Пример захвата локальной переменной из объемлющей области действия

```
interface MyFunc {
    int func(int n);
}

class VarCapture {
    public static void main(String args[])
    {
        // Локальная переменная, которая может быть захвачена
        int num = 10;

        MyFunc myLambda = (n) -> {
            // Такое применение переменной num допустимо, поскольку
            // она не видоизменяется
            int v = num + n;

            // Но следующая строка кода недопустима, поскольку в ней
            // предпринимается попытка видоизменить значение переменной num
            // num++;
            return v;
        };

        // И следующая строка кода приведет к ошибке, поскольку в ней
        // нарушается действительно завершенное состояние переменной num
        // num = 9;
    }
}
```

Как следует из комментариев к данному примеру программы, переменная `num` является действительно завершенной, и поэтому ее можно использовать в лямбда-выражении `myLambda`. Но если попытаться видоизменить переменную `num` как в самом лямбда-выражении, так и за его пределами, то она утратит свое действительно завершенное состояние. Это привело бы к ошибке, а программа не поддалась бы компиляции.

Следует особо подчеркнуть, что в лямбда-выражении можно использовать и видоизменять переменную экземпляра из вызывающего его класса. Но нельзя использовать локальную переменную из объемлющей его области действия, если только эта переменная не является действительно завершенной.

Ссылки на методы

С лямбда-выражениями связано еще одно очень важное средство, называемое *ссылкой на метод*. Такая ссылка позволяет обращаться к методу, не вызывая его. Она связана с лямбда-выражениями потому, что ей также требуется контекст целевого типа, состоящий из совместимого функционального интерфейса. Имеются разные виды ссылок на методы. Рассмотрим сначала ссылки на статические методы.

Ссылки на статические методы

Для создания ссылки на статический метод служит следующая общая форма:

имя_класса::имя_метода

Обратите внимание на то, что имя класса в этой форме отделяется от имени метода двумя двоеточиями (::). Этот новый разделитель внедрен в версии JDK 8 специально для данной цели. Такой ссылкой на метод можно пользоваться везде, где она совместима со своим целевым типом.

В следующем примере программы демонстрируется применение ссылки на статический метод:

```
// Продемонстрировать ссылку на статический метод

// Функциональный интерфейс для операций с символьными строками
interface StringFunc {
    String func(String n);
}

// В этом интерфейсе определяется статический метод strReverse()
class MyStringOps {
    // Статический метод, изменяющий порядок
    // следования символов в строке
    static String strReverse(String str) {
        String result = "";
        int i;

        for(i = str.length()-1; i >= 0; i--)
            result += str.charAt(i);

        return result;
    }
}

class MethodRefDemo {

    // В этом методе функциональный интерфейс указывается в качестве
    // типа первого его параметра. Следовательно, ему может быть передан
    // любой экземпляр этого интерфейса, включая и ссылку на метод
    static String stringOp(StringFunc sf, String s) {
```

```

    return sf.func(s);
}

public static void main(String args[])
{
    String inStr = "Лямбда-выражения повышают эффективность Java";
    String outStr;

    // Здесь ссылка на метод strReverse() передается методу stringOp()
    outStr = stringOp(MyStringOps::strReverse, inStr);

    System.out.println("Исходная строка: " + inStr);
    System.out.println("Обращенная строка: " + outStr);
}
}

```

Ниже приведен пример выполнения данной программы.

Исходная строка: Лямбда-выражения повышают эффективность Java
 Обращенная строка: яинежарыв-адбмяЛ тшашвоп ьтсонвиткеффэ аvаJ

В данной программе особое внимание обратите на следующую строку кода:

```
outStr = stringOp(MyStringOps::strReverse, inStr);
```

В этой строке кода ссылка на статический метод `strReverse()`, объявляемый в классе `MyStringOps`, передается первому аргументу метода `stringOp()`. И это вполне допустимо, поскольку метод `strReverse()` совместим с функциональным интерфейсом `StringFunc`. Следовательно, в выражении `MyStringOps::strReverse` вычисляется ссылка на объект того класса, в котором метод `strReverse()` предоставляет реализацию метода `func()` из функционального интерфейса `StringFunc`.

Ссылки на методы экземпляра

Для передачи ссылки на метод экземпляра для конкретного объекта служит следующая общая форма:

ссылка_на_объект::имя_метода

Как видите, синтаксис этой формы ссылки на метод экземпляра похож на тот, что используется для ссылки на статический метод, за исключением того, что вместо имени класса в данном случае используется ссылка на объект. Ниже приведен переделанный вариант программы из предыдущего примера, чтобы продемонстрировать применение ссылки на метод экземпляра.

```

// Продемонстрировать применение ссылки на метод экземпляра

// Функциональный интерфейс для операций с символьными строками
interface StringFunc {
    String func(String n);
}

// Теперь в этом классе определяется метод экземпляра strReverse()
class MyStringOps {
    String strReverse(String str) {
        String result = "";
        int i;
    }
}

```

```

        for(i = str.length()-1; i >= 0; i--)
            result += str.charAt(i);

        return result;
    }
}

class MethodRefDemo2 {

    // В этом методе функциональный интерфейс указывается в качестве
    // типа первого его параметра. Следовательно, ему может быть передан
    // любой экземпляр этого интерфейса, включая и ссылку на метод
    static String stringOp(StringFunc sf, String s) {
        return sf.func(s);
    }

    public static void main(String args[])
    {
        String inStr = "Лямбда-выражения повышают эффективность Java";
        String outStr;

        // создать объект типа MyStringOps
        MyStringOps strOps = new MyStringOps();

        // А теперь ссылка на метод экземпляра strReverse()
        // передается методу stringOp()
        outStr = stringOp(strOps::strReverse, inStr);

        System.out.println("Исходная строка: " + inStr);
        System.out.println("Обращенная строка: " + outStr);
    }
}

```

Эта версия программы выводит такой же результат, как и предыдущая ее версия. В данном примере программы обратите внимание на то, что метод `strReverse()` теперь объявляется в классе `MyStringOps` как метод экземпляра. А в теле метода `main()` создается экземпляр `strOps` класса `MyStringOps`. Этот экземпляр служит для создания ссылки на свой метод `strReverse()` при вызове метода `stringOp()`, как еще раз показано ниже. В данном примере метод экземпляра `strReverse()` вызывается для объекта `strOps`.

```
outStr = stringOp(strOps::strReverse, inStr);
```

Возможны и такие случаи, когда требуется указать метод экземпляра, который будет использоваться вместе с любым объектом данного класса, а не только с указанным объектом. В подобных случаях можно создать ссылку на метод экземпляра в следующей общей форме:

имя_класса::имя_метода_экземпляра

В этой форме имя класса указывается вместо имени конкретного объекта, несмотря на то, что в ней указывается и метод экземпляра. В соответствии с этой формой первый параметр метода из функционального интерфейса совпадает с вызывающим объектом, а второй параметр — с параметром, указанным в методе экземпляра. Рассмотрим пример программы, в которой определяется метод `counter()`, подсчитывающий количество объектов в массиве, удовлетворяющих усло-

вию, определяемому в методе `func()` из функционального интерфейса `MyFunc`.
В данном случае подсчитываются экземпляры класса `HighTemp`.

// Использовать ссылку на метод экземпляра вместе с разными объектами

// Функциональный интерфейс с методом, принимающим два ссылочных
// аргумента и возвращающим логическое значение

```
interface MyFunc<T> {
    boolean func(T v1, T v2);
}
```

// Класс для хранения максимальной температуры за день

```
class HighTemp {
    private int hTemp;
```

```
    HighTemp(int ht) { hTemp = ht; }
```

// вернуть логическое значение **true**, если вызывающий объект
// типа **HighTemp** имеет такую же температуру, как и у объекта **ht2**

```
    boolean sameTemp(HighTemp ht2) {
        return hTemp == ht2.hTemp;
    }
```

// вернуть логическое значение **true**, если вызывающий объект
// типа **HighTemp** имеет температуру ниже, чем у объекта **ht2**

```
    boolean lessThanTemp(HighTemp ht2) {
        return hTemp < ht2.hTemp;
    }
```

```
}
```

```
class InstanceMethWithObjectRefDemo {
```

// Метод, возвращающий количество экземпляров объекта,
// найденных по критериям, задаваемым параметром

// функционального интерфейса **MyFunc**

```
    static <T> int counter(T[] vals, MyFunc<T> f, T v) {
```

```
        int count = 0;
```

```
        for(int i=0; i < vals.length; i++)
            if(f.func(vals[i], v)) count++;
```

```
        return count;
```

```
    }
```

```
    public static void main(String args[])
```

```
    {
```

```
        int count;
```

// создать массив объектов типа **HighTemp**

```
        HighTemp[] weekDayHighs = { new HighTemp(89), new HighTemp(82),
                                     new HighTemp(90), new HighTemp(89),
                                     new HighTemp(89), new HighTemp(91),
                                     new HighTemp(84), new HighTemp(83) };
```

// Использовать метод **counter()** вместе с массивами объектов

// типа **HighTemp**. Обратите внимание на то, что ссылка на метод

// экземпляра **sameTemp()** передается в качестве второго параметра

```
        count = counter(weekDayHighs, HighTemp::sameTemp,
                        new HighTemp(89));
```

```
        System.out.println(
```

```
            "Дней, когда максимальная температура была 89: " + count);
```

```
// А теперь создать и использовать вместе с данным
// методом еще один массив объектов типа HighTemp
HighTemp[] weekDayHighs2 = { new HighTemp(32), new HighTemp(12),
                             new HighTemp(24), new HighTemp(19),
                             new HighTemp(18), new HighTemp(12),
                             new HighTemp(-1), new HighTemp(13) };

count = counter(weekDayHighs2, HighTemp::sameTemp,
                new HighTemp(12));
System.out.println(
    "Дней, когда максимальная температура была 12: " + count);

// А теперь воспользоваться методом lessThanTemp(), чтобы
// выяснить, сколько дней температура была меньше заданной
count = counter(weekDayHighs, HighTemp::lessThanTemp,
                new HighTemp(89));
System.out.println(
    "Дней, когда максимальная температура была меньше 89: " + count);

count = counter(weekDayHighs2, HighTemp::lessThanTemp,
                new HighTemp(19));
System.out.println(
    "Дней, когда максимальная температура была меньше 19: " + count);
}
```

Ниже приведен результат выполнения данной программы.

```
Дней, когда максимальная температура была 89: 3
Дней, когда максимальная температура была 12: 2
Дней, когда максимальная температура была меньше 89: 3
Дней, когда максимальная температура была меньше 19: 5
```

В данном примере программы обратите внимание на то, что в классе `HighTemp` объявлены два метода экземпляра: `sameTemp()` и `lessThanTemp()`. Первый метод возвращает логическое значение `true`, если оба объекта типа `HighTemp` содержат одинаковую температуру. А второй метод возвращает логическое значение `true`, если температура в вызывающем объекте меньше, чем в передаваемом. Каждый из этих методов принимает параметр типа `HighTemp` и возвращает логическое значение. Следовательно, каждый из них совместим с функциональным интерфейсом `MyFunc`, поскольку тип вызывающего объекта может быть приведен к типу первого параметра метода `func()`, а тип его аргумента — к типу второго параметра этого метода. Таким образом, когда следующее выражение:

```
HighTemp::sameTemp
```

передается методу `counter()`, то создается экземпляр функционального интерфейса `MyFunc`, где тип первого параметра метода `func()` соответствует типу объекта, вызывающего метод экземпляра, т.е. типу `HighTemp`. А тип второго параметра метода `func()` также соответствует типу `HighTemp`, поскольку это тип параметра метода экземпляра `sameTemp()`. Это же справедливо и для метода экземпляра `lessThanTemp()`.

И последнее замечание: используя оператор `super`, можно обращаться к варианту метода из суперкласса, как показано ниже, где *имя* обозначает имя вызываемого метода.

```
super::имя
```

Ссылки на обобщенные методы

Ссылками на методы можно также пользоваться для обращения к обобщенным классам и/или методам. В качестве примера рассмотрим следующую программу:

```
// Продемонстрировать применение ссылки на обобщенный метод,
// объявленный в необобщенном классе

// Функциональный интерфейс для обработки массива значений
// и возврата целочисленного результата
interface MyFunc<T> {
    int func(T[] vals, T v);
}

// В этом классе определяется метод countMatching(), возвращающий
// количество элементов в массиве, равных указанному значению.
// Обратите внимание на то, что метод countMatching() является
// обобщенным, тогда как класс MyArrayOps – необобщенным
class MyArrayOps {
    static <T> int countMatching(T[] vals, T v) {
        int count = 0;

        for(int i=0; i < vals.length; i++)
            if(vals[i] == v) count++;

        return count;
    }
}

class GenericMethodRefDemo {

    // В качестве первого параметра этого метода указывается
    // функциональный интерфейс MyFunc, а в качестве двух других
    // параметров – массив и значение, причем оба типа T
    static <T> int myOp(MyFunc<T> f, T[] vals, T v) {
        return f.func(vals, v);
    }

    public static void main(String args[])
    {
        Integer[] vals = { 1, 2, 3, 4, 2, 3, 4, 4, 5 };
        String[] strs = { "Один", "Два", "Три", "Два" };
        int count;

        count = myOp(MyArrayOps.<Integer>countMatching, vals, 4);
        System.out.println("Массив vals содержит " + count + " числа 4");

        count = myOp(MyArrayOps.<String>countMatching, strs, "Two");
        System.out.println("Массив strs содержит " + count + " числа два");
    }
}
```

Ниже приведен результат выполнения данной программы.

```
Массив vals содержит 3 числа 4
Массив strs содержит 2 числа два
```

В данном примере программы необобщенный класс `MyArrayOps` содержит обобщенный метод `countMatching()`. Этот метод возвращает количество элементов в массиве, совпадающих с указанным значением. Обратите внимание на порядок указания аргумента обобщенного типа. Например, при первом вызове из метода `main()` этому методу передается аргумент типа `Integer` следующим образом:

```
count = myOp(MyArrayOps.<Integer>countMatching, vals, 4);
```

Обратите также внимание на то, что это происходит после разделителя `::`. Этот синтаксис можно обобщить. Когда обобщенный метод указывается как метод экземпляра, его аргумент типа указывается после разделителя `::` и перед именем этого метода. Следует, однако, заметить, что явно указывать аргумент типа в данном случае (и во многих других) совсем не обязательно, поскольку тип этого аргумента выводится автоматически. А в тех случаях, когда указывается обобщенный класс, аргумент типа следует после имени этого класса и перед разделителем `::`.

Несмотря на то что в предыдущих примерах был продемонстрирован механизм применения ссылок на методы, эти примеры все же не раскрывают в полной мере их преимуществ. Ссылки на методы могут, в частности, оказаться очень полезными в сочетании с каркасом коллекций `Collections Framework`, описываемым в главе 18. И ради полноты изложения ниже приведен краткий, но наглядный пример применения ссылки на метод, чтобы определить наибольший элемент в коллекции. (Если вы еще незнакомы с каркасом коллекций `Collections Framework`, вернитесь к этому примеру после того, как проработаете материал главы 18.)

Обнаружить в коллекции наибольший элемент можно, в частности, вызвав метод `max()`, определенный в классе `Collections`. При вызове варианта метода `max()`, применяемого в рассматриваемом здесь примере, нужно передать ссылку на коллекцию и экземпляр объекта, реализующего интерфейс `Comparator<T>`. В этом интерфейсе определяется порядок сравнения двух объектов. В нем объявляется единственный абстрактный метод `compare()`, принимающий два аргумента, имеющих типы сравниваемых объектов. Этот метод должен вернуть числовое значение больше нуля, если первый аргумент больше второго; нулевое значение, если оба аргумента равны; и числовое значение меньше нуля, если первый объект меньше второго.

Прежде для вызова метода `max()` с двумя определяемыми пользователем объектами экземпляр интерфейса `Comparator<T>` приходилось получать, реализовав сначала этот интерфейс явным образом в отдельном классе, а затем создав экземпляр данного класса. Далее этот экземпляр передавался в качестве компаратора методу `max()`. В версии `JDK 8` появилась возможность просто передать методу `max()` ссылку на сравнение, поскольку в этом случае компаратор реализуется автоматически. Этот процесс демонстрируется ниже на простом примере создания коллекции типа `ArrayList` объектов типа `MyClass` и поиска в ней наибольшего значения, определяемого в методе сравнения.

```
// Использовать ссылку на метод, чтобы найти
// максимальное значение в коллекции
import java.util.*;
```

```
class MyClass {
    private int val;
```

```

MyClass(int v) { val = v; }

int getVal() { return val; }
}

class UseMethodRef {
    // Метод compare(), совместимый с аналогичным методом,
    // определенным в интерфейсе Comparator<T>
    static int compareMC(MyClass a, MyClass b) {
        return a.getVal() - b.getVal();
    }

    public static void main(String args[])
    {
        ArrayList<MyClass> al = new ArrayList<MyClass>();

        al.add(new MyClass(1));
        al.add(new MyClass(4));
        al.add(new MyClass(2));
        al.add(new MyClass(9));
        al.add(new MyClass(3));
        al.add(new MyClass(7));

        // найти максимальное значение, используя метод compareMC()
        MyClass maxValObj = Collections.max(al, UseMethodRef::compareMC);

        System.out.println("Максимальное значение равно: " + maxValObj.getVal());
    }
}

```

Ниже приведен результат выполнения данной программы.

Максимальное значение равно: 9

В данном примере программы обратите внимание на то, что в самом классе `MyClass` не определяется метод сравнения и не реализуется интерфейс `Comparator`. Тем не менее максимальное значение в списке объектов типа `MyClass` может быть получено в результате вызова метода `max()`, поскольку в классе `UseMethodRef` определяется статический метод `compareMC()`, совместимый с методом `compare()`, определенным в интерфейсе `Comparator`. Таким образом, отпадает необходимость явным образом реализовывать и создавать экземпляр интерфейса `Comparator`.

Ссылки на конструкторы

Ссылки на конструкторы можно создавать таким же образом, как и ссылки на методы. Ниже приведена общая форма синтаксиса, которую можно употреблять для создания ссылок на конструкторы.

имя_класса::new

Эта ссылка может быть присвоена любой ссылке на функциональный интерфейс, в котором определяется метод, совместимый с конструктором. Ниже приведен простой пример применения ссылки на конструктор.

```
// Продемонстрировать применение ссылки на конструктор

// В функциональном интерфейсе MyFunc определяется метод,
// возвращающий ссылку на класс MyClass
interface MyFunc {
    MyClass func(int n);
}

class MyClass {
    private int val;

    // Этот конструктор принимает один аргумент
    MyClass(int v) { val = v; }

    // А это конструктор по умолчанию
    MyClass() { val = 0; }

    // ...

    int getVal() { return val; };
}

class ConstructorRefDemo {
    public static void main(String args[])
    {
        // Создать ссылку на конструктор класса MyClass.
        // Метод func() из интерфейса MyFunc принимает аргумент,
        // поэтому оператор new обращается к параметризованному
        // конструктору класса MyClass, а не к его конструктору по умолчанию
        MyFunc myClassCons = MyClass::new;

        // создать экземпляр класса MyClass по ссылке на его конструктор
        MyClass mc = myClassCons.func(100);

        // использовать только что созданный экземпляр класса MyClass
        System.out.println("Значение val в объекте mc равно " + mc.getVal());
    }
}
```

Ниже приведен результат, выводимый данной программой.

Значение val в объекте mc равно 100

В данном примере программы обратите внимание на то, что метод `func()` из интерфейса `MyFunc` возвращает ссылку на тип `MyClass` и принимает параметр типа `int`. Обратите также внимание на то, что в классе `MyClass` определяются два конструктора. В первом конструкторе указывается параметр типа `int`, а второй является конструктором по умолчанию и поэтому не имеет параметров. А теперь проанализируем следующую строку кода:

```
MyFunc myClassCons = MyClass::new;
```

В этой строке кода создается ссылка на конструктор класса `MyClass` в выражении `MyClass::new`. В данном случае ссылка делается на конструктор `MyClass(int v)`, поскольку метод `func()` из интерфейса `MyFunc` принимает параметр типа `int`, а с ним совпадает именно этот конструктор. Обратите также внимание на то, что ссылка на этот конструктор присваивается переменной `myClassCons` ссылки на функциональный интерфейс `MyFunc`. После выполнения данной строки кода

переменную `myClassCons` можно использовать для создания экземпляра класса `MyClass`, как показано ниже. По существу, переменная `myClassCons` предоставляет еще один способ вызвать конструктор `MyClass(int v)`.

```
MyClass mc = myClassCons.func(100);
```

Аналогичным образом создаются ссылки на конструкторы обобщенных классов. Единственное отличие состоит в том, что в данном случае может быть указан аргумент типа. И делается это после имени класса, как и при создании ссылки на обобщенный метод. Создание и применение ссылки на конструктор обобщенного класса демонстрируется на приведенном ниже примере, где функциональный интерфейс `MyFunc` и класс `MyClass` объявляются как обобщенные.

```
// Продемонстрировать применение ссылки на
// конструктор обобщенного класса

// Теперь функциональный интерфейс MyFunc обобщенный
interface MyFunc<T> {
    MyClass<T> func(T n);
}

class MyClass<T> {
    private T val;

    // Этот конструктор принимает один аргумент
    MyClass(T v) { val = v; }

    // А это конструктор по умолчанию
    MyClass() { val = null; }

    // ..
    T getVal() { return val; };
}

class ConstructorRefDemo2 {

    public static void main(String args[])
    {
        // создать ссылку на конструктор обобщенного класса MyClass<T>
        MyFunc<Integer> myClassCons = MyClass<Integer>::new;

        // создать экземпляр класса MyClass<T>
        // по данной ссылке на конструктор
        MyClass<Integer> mc = myClassCons.func(100);

        // воспользоваться только что созданным
        // экземпляром класса MyClass<T>
        System.out.println(
            "Значение val в объекте mc равно " + mc.getVal() );
    }
}
```

Эта версия программы выводит такой же результат, как и предыдущая ее версия, только теперь функциональный интерфейс `MyFunc` и класс `MyClass` являются обобщенными. Следовательно, в последовательность кода, создающего ссылку на конструктор, можно включить аргумент типа, как показано ниже, хотя это требуется далеко не всегда.

```
MyFunc<Integer> myClassCons = MyClass<Integer>::new;
```

Аргумент типа `Integer` уже указан при создании переменной `myClassCons`, поэтому его можно использовать для создания объекта типа `MyClass<Integer>`, как показано в следующей строке кода:

```
MyClass<Integer> mc = myClassCons.func(100);
```

В представленных выше примерах был продемонстрирован механизм применения ссылки на конструктор, но на практике они подобным образом не используются, поскольку это не приносит никаких выгод. Более того, наличие двух обозначений одного и того же конструктора приводит, по меньшей мере, к конфликтной ситуации. Поэтому с целью продемонстрировать более практический пример применения ссылок на конструкторы в приведенной ниже программе применяется статический метод `myClassFactory()`, который является фабричным для объектов класса любого типа, реализующего интерфейс `MyFunc`. С помощью этого метода можно создать объект любого типа, имеющего конструктор, совместимый с его первым параметром.

// Реализовать простую фабрику классов, используя ссылку на конструктор

```
interface MyFunc<R, T> {  
    R func(T n);  
}
```

```
// Простой обобщенный класс  
class MyClass<T> {  
    private T val;
```

```
    // Конструктор, принимающий один параметр  
    MyClass(T v) { val = v; }
```

```
    // Конструктор по умолчанию. Этот конструктор в  
    // данной программе НЕ используется  
    MyClass() { val = null; }  
    // ...
```

```
    T getVal() { return val; };  
}
```

```
// Простой необобщенный класс  
class MyClass2 {  
    String str;
```

```
    // Конструктор, принимающий один аргумент  
    MyClass2(String s) { str = s; }
```

```
    // Конструктор по умолчанию. Этот конструктор в  
    // данной программе НЕ используется  
    MyClass2() { str = ""; }
```

```
    // ...
```

```
    String getVal() { return str; };  
}
```

```
class ConstructorRefDemo3 {
```

```
    // Фабричный метод для объектов разных классов.  
    // У каждого класса должен быть свой конструктор,  
    // принимающий один параметр типа T. А параметр R  
    // обозначает тип создаваемого объекта
```

```

static <R,T> R myClassFactory(MyFunc<R, T> cons, T v) {
    return cons.func(v);
}

public static void main(String args[])
{
    // Создать ссылку на конструктор класса MyClass.
    // В данном случае оператор new обращается к конструктору,
    // принимающему аргумент
    MyFunc<MyClass<Double>, Double> myClassCons = MyClass<Double>::new;

    // создать экземпляр типа класса MyClass, используя фабричный метод
    MyClass<Double> mc = myClassFactory(myClassCons, 100.1);

    // использовать только что созданный экземпляр класса MyClass
    System.out.println(
        "Значение val в объекте mc равно " + mc.getVal());

    // А теперь создать экземпляр другого класса,
    // используя метод myClassFactory()
    MyFunc<MyClass2, String> myClassCons2 = MyClass2::new;

    // создать экземпляр класса MyClass2, используя фабричный метод
    MyClass2 mc2 = myClassFactory(myClassCons2, "Лямбда");

    // использовать только что созданный экземпляр класса MyClass
    System.out.println(
        "Значение str в объекте mc2 равно " + mc2.getVal());
}
}

```

Ниже приведен результат выполнения данной программы.

Значение val в объекте mc равно 100.1
 Значение str в объекте mc2 равно Лямбда

Как видите, метод `myClassFactory()` используется для создания объектов типа `MyClass<Double>` и `MyClass2`. Несмотря на отличия в обоих классах, в частности, класс `MyClass` является обобщенным, а класс `MyClass2` — необобщенным, объекты обоих классов могут быть созданы с помощью фабричного метода `myClassFactory()`, поскольку оба они содержат конструкторы, совместимые с методом `func()` из функционального интерфейса `MyFunc`, а методу `myClassFactory()` передается конструктор того класса, объект которого требуется создать. Можете поэкспериментировать немного с данной программой, попробовав создать объекты разных классов, а также экземпляры разнотипных объектов класса `MyClass`. При этом вы непременно обнаружите, что с помощью метода `myClassFactory()` можно создать объект любого типа, в классе которого имеется конструктор, совместимый с методом `func()` из функционального интерфейса `MyFunc`. Несмотря на всю простоту данного примера, он все же раскрывает истинный потенциал ссылок на конструкторы в Java.

Прежде чем продолжить дальше, следует упомянуть о второй форме синтаксиса ссылок на конструкторы, в которой применяются массивы. В частности, для создания ссылки на конструктор массива служит следующая форма:

```
тип[]::new
```

где *тип* обозначает создаваемый объект. Так, если обратиться к форме класса `MyClass`, представленной в первом примере применения ссылки на конструктор

(ConstructorRefDemo), а также объявить интерфейс `MyArrayCreator` следующим образом:

```
interface MyArrayCreator<T> {
    T func(int n);
}
```

то в приведенном ниже фрагменте кода создается двухэлементный массив объектов типа `MyClass` и каждому из них присваивается начальное значение.

```
MyArrayCreator<MyClass[]> mcArrayCons = MyClass[]::new;
MyClass[] a = mcArrayCons.func(2);
a[0] = new MyClass(1);
a[1] = new MyClass(2)
```

В этом фрагменте кода вызов метода `func(2)` приводит к созданию двухэлементного массива. Как правило, функциональный интерфейс должен содержать метод, принимающий единственный параметр типа `int`, если он служит для обращения к конструктору массива.

Предопределенные функциональные интерфейсы

В приведенных до сих пор примерах определялись собственные функциональные интерфейсы для целей наглядной демонстрации основных принципов действия лямбда-выражений и функциональных интерфейсов. Но зачастую определять собственный функциональный интерфейс не нужно, поскольку в версии JDK 8 внедрен новый пакет `java.util.function`, предоставляющий несколько предопределенных функциональных интерфейсов. Более подробно они будут рассматриваться в части II данной книги, а в табл. 15.1 приведены некоторые избранные из них.

Таблица 15.1. Предопределенные функциональные интерфейсы

Функциональный интерфейс	Назначение
<code>UnaryOperator<T></code>	Выполняет унарную операцию над объектом типа T и возвращает результат того же типа. Содержит метод apply()
<code>BinaryOperator<T></code>	Выполняет логическую операцию над двумя объектами типа T и возвращает результат того же типа. Содержит метод apply()
<code>Consumer<T></code>	Выполняет операцию над объектом типа T . Содержит метод accept()
<code>Supplier<T></code>	Возвращает объект типа T . Содержит метод get()
<code>Function<T, R></code>	Выполняет операцию над объектом типа T и возвращает в результате объект типа R . Содержит метод apply()
<code>Predicate<T></code>	Определяет, удовлетворяет ли объект типа T некоторому ограничительному условию. Возвращает логическое значение, обозначающее результат. Содержит метод test()

В приведенном ниже переделанном примере представленной ранее программы BlockLambdaDemo демонстрируется применение преопределенного функционального интерфейса `Function`. Если в предыдущей версии данной программы для демонстрации блочных лямбда-выражений на примере вычисления факториала заданного числа был создан собственный функциональный интерфейс `NumericFunc`, то в новой ее версии для этой цели применяется встроенный функциональный интерфейс `Function`, как показано ниже.

```
// Использовать встроенный функциональный интерфейс Function

// импортировать функциональный интерфейс Function
import java.util.function.Function;

class UseFunctionInterfaceDemo {
    public static void main(String args[])
    {
        // Это блочное лямбда-выражение вычисляет факториал
        // целочисленного значения. Для этой цели на сей раз
        // используется функциональный интерфейс Function
        Function<Integer, Integer> factorial = (n) -> {
            int result = 1;
            for(int i=1; i <= n; i++)
                result = i * result;
            return result;
        };

        System.out.println("Факториал числа 3 равен " + factorial.apply(3));
        System.out.println("Факториал числа 5 равен " + factorial.apply(5));
    }
}
```

Эта версия программы выводит такой же результат, как и предыдущая ее версия.

ЧАСТЬ

II

Библиотека Java

ГЛАВА 16

Обработка
символьных строк

ГЛАВА 17

Пакет `java.lang`

ГЛАВА 18

Пакет `java.util`, часть I.
Collections Framework

ГЛАВА 19

Пакет `java.util`, часть II.
Прочие служебные классы

ГЛАВА 20

Пакет `java.io`
для ввода-вывода

ГЛАВА 21

Система ввода-
вывода NIO

ГЛАВА 22

Работа в сети

ГЛАВА 23

Наследование

ГЛАВА 24

Пакеты и интерфейсы

ГЛАВА 25

Обработка исключений

ГЛАВА 26

Многопоточное
программирование

ГЛАВА 27

Перечисления,
автоупаковка и аннотации
(метаданные)

ГЛАВА 28

Ввод-вывод, апплеты
и прочие вопросы

ГЛАВА 29

Обобщения

ГЛАВА 30

Лямбда-выражения

Краткий обзор обработки символьных строк в Java был сделан в главе 7. А в этой главе данная тема рассматривается подробнее. Как и в других языках программирования, в Java *символьная строка* является последовательностью символов. Но в отличие от некоторых других языков, где символьные строки реализованы в виде массивов символов, в Java они являются объектами класса `String`.

Реализация символьных строк в виде встроенных объектов даст возможность предоставить в Java полный набор средств для удобной обработки символьных строк. Например, в Java предоставляются методы для сравнения и объединения двух символьных строк, поиска в них подстрок и изменения регистра символов. Кроме того, объекты класса `String` могут быть созданы самыми разными способами, что позволяет легко получать символьные строки по мере надобности в них.

Как ни странно, в результате создания объекта типа `String` получается символьная строка, которая не может быть изменена. Иными словами, после того как объект типа `String` будет создан, изменить символы, составляющие новую строку, уже нельзя. На первый взгляд это может показаться серьезным ограничением, но на самом деле оно не так и важно. Над символьными строками можно выполнять любые операции, но всякий раз, когда требуется измененная версия существующей символьной строки, создается новый объект типа `String`, содержащий все внесенные изменения. А исходная символьная строка остается без изменения. Такой подход принят потому, что фиксированная неизменяемая символьная строка может быть реализована более эффективно, чем изменяемая. Если требуются видоизменяемые символьные строки, то в Java предлагаются на выбор два класса: `StringBuffer` и `StringBuilder`. Объекты обоих классов содержат символьные строки, которые могут быть изменены после их создания.

Классы `String`, `StringBuffer` и `StringBuilder` определены в пакете `java.lang` и поэтому доступны во всех программах автоматически. Все эти классы объявлены с модификатором доступа `final`, а следовательно, ни от одного из них нельзя произвести подклассы. Это допускает некоторую оптимизацию, повышающую производительность общих операций с символьными строками. Все три класса реализуют интерфейс `CharSequence`.

И последнее замечание: неизменяемость символьных строк в объектах типа `String` означает, что содержимое экземпляра класса `String` не может быть изменено после его создания. Но переменная, объявленная как ссылка на объект типа `String`, может быть в любой момент изменена таким образом, чтобы указывать на другой объект типа `String`.

Конструкторы символьных строк

В классе `String` поддерживается несколько конструкторов. Для создания пустого объекта типа `String` вызывается стандартный конструктор. Например, в следующей строке кода создается экземпляр класса `String`, не содержащий символы:

```
String s = new String();
```

Зачастую символьные строки требуется создавать с начальными значениями. Для этой цели в классе `String` предоставляются разнообразные конструкторы. В частности, для создания символьной строки, инициализируемой массивом символов, служит следующий конструктор:

```
String(char символы[])
```

Ниже приведен пример применения этого конструктора. Он инициализирует строку `s` символами "abc".

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
```

Используя следующий конструктор, можно задать поддиапазон, т.е. определенную часть массива символов для инициализации ними строки:

```
String(char символы[], int начальный_индекс, int количество_символов)
```

где параметр *начальный_индекс* обозначает начало поддиапазона, а параметр *количество_символов* — те символы, которые нужно использовать для инициализации строки. В следующем примере строка `s` инициализируется символами "cde":

```
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
String s = new String(chars, 2, 3);
```

Используя следующий конструктор, можно создать объект типа `String`, содержащий ту же последовательность символов, что и другой объект типа `String`:

```
String(String строковый_объект)
```

где параметр *строковый_объект* обозначает объект типа `String`. Рассмотрим следующий пример программы:

```
// Создать один объект типа String из другого
class MakeString {
    public static void main(String args[]) {
        char c[] = {'J', 'a', 'v', 'a'};
        String s1 = new String(c);
        String s2 = new String(s1);
        System.out.println(s1);
        System.out.println(s2);
    }
}
```

Ниже приведен результат, выводимый данной программой. Как видите, символьные строки `s1` и `s2` содержат одинаковые значения.

```
Java
Java
```

Несмотря на то что в примитивном типе `char` языка Java для представления основного набора символов в Юникоде используется 16 бит, в типичном формате символьных строк, пересылаемых через Интернет, используются массивы 8-разрядных байтов, создаваемых из набора символов в коде ASCII. Чаще всего употребляются 8-разрядные строки в коде ASCII, и поэтому в классе `String` предоставляются конструкторы, инициализирующие символьную строку массивом типа `byte`. Ниже приведена общая форма этих конструкторов.

```
String(byte символы[])  
String(byte символы[], int начальный_индекс, int количество_символов)
```

Здесь параметр *символы* обозначает массив байтов. Вторая форма конструктора позволяет указать требуемый поддиапазон. В каждом из этих конструкторов преобразование байтов в символы выполняется в соответствии с кодировкой, выбираемой на конкретной платформе по умолчанию. Применение этих конструкторов демонстрируется в следующем примере программы:

```
// Создать символьную строку из подмножества массива символов  
class SubStringCons {  
    public static void main(String args[]) {  
        byte ascii[] = {65, 66, 67, 68, 69, 70 };  
        String s1 = new String(ascii);  
        System.out.println(s1);  
        String s2 = new String(ascii, 2, 3);  
        System.out.println(s2);  
    }  
}
```

Ниже приведен результат, выводимый данной программой.

```
ABCDEF  
CDE
```

Имеются также расширенные версии конструкторов, где байты преобразуются в символьные строки и можно указать кодировку символов, определяющую порядок преобразования байтов в символы. Но, как правило, требуется кодировка, выбираемая на конкретной платформе по умолчанию.

На заметку! Содержимое массива копируется всякий раз, когда объект типа `String` создается из массива. Даже если содержимое массива изменится после создания символьной строки в виде объекта типа `String`, последний останется без изменения.

Объект типа `String` можно создать из объекта типа `StringBuffer`, используя следующий конструктор:

```
String(StringBuffer объект_буфера_строк)
```

А создать символьную строку из объекта типа `StringBuilder` можно с помощью такого конструктора:

```
String(StringBuilder объект_построения_строки)
```

В следующем конструкторе поддерживается расширенный набор символов в Юникоде:

```
String(int кодовые_точки[], int начальный_индекс, int количество_символов)
```

где параметр *кодовые_точки* обозначает массив, содержащий символы в Юникоде. Результирующая строка создается из поддиапазона символов, начало которого обозначает *начальный_индекс*, а длина — *количество_символов*. Имеются также конструкторы, позволяющие указывать набор символов.

На заметку! Обсуждение основных понятий и составляющих Юникода, в том числе кодовых точек, а также обращение с ними в Java приведено в главе 17.

Длина символьной строки

Количество символов, из которых состоит строка, определяет ее длину. Чтобы получить это значение, достаточно вызвать метод `length()`:

```
int length()
```

В следующем фрагменте кода выводится символьная строка "З", поскольку именно три символа содержит исходная строка `s`:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());
```

Специальные строковые операции

Символьные строки являются очень важной составляющей программирования, и поэтому в Java обеспечивается специальная поддержка некоторых строковых операций в рамках синтаксиса этого языка. Эти операции включают автоматическое создание новых экземпляров класса `String` из строковых литералов, сцепление многих объектов типа `String` с помощью операции `+`, а также преобразование других типов данных в их строковое представление. Для реализации всех этих операций имеются специальные явно указываемые методы, но для удобства и большей ясности программирования на Java они выполняются автоматически.

Строковые литералы

В предыдущих примерах было показано, как явным образом создавать объекты класса `String` из массива символов с помощью оператора `new`. Но есть и более простой способ сделать то же самое с помощью строковых литералов. Для каждого строкового литерала в прикладной программе на Java автоматически создается объект типа `String`. Таким образом, строковый литерал можно использовать для инициализации объекта типа `String`. Например, в следующем фрагменте кода создаются две одинаковые символьные строки:

```
char chars[] = { 'a', 'b', 'c' };
String s1 = new String(chars);

String s2 = "abc"; // использовать строковый литерал
```

Объект типа `String` создается для каждого строкового литерала, и поэтому строковый литерал можно использовать в любом месте, где допускается применение объекта типа `String`. Например, методы можно вызывать непосредственно для символьных строк в кавычках как по ссылкам на объекты. Так, в приведенном ниже примере метод `length()` вызывается для символьной строки `"abc"`. Как и следовало ожидать, на экран выводится строка `"3"`.

```
System.out.println("abc".length());
```

Сцепление строк

Как правило, в Java не разрешается выполнять арифметические операции над объектами типа `String`. Но из этого правила имеется исключение: с помощью операции `+` можно сцеплять, т.е. соединять две символьные строки, порождая в итоге объект типа `String`. Операции сцепления символьных строк можно объединять в цепочку. Например, в приведенном ниже фрагменте кода выполняется сцепление трех символьных строк. В итоге выводится символьная строка `"Ему 9 лет"`.

```
String age = "9";
String s = "Ему " + age + " лет.";
System.out.println(s);
```

Сцепление символьных строк находит практическое применение, в частности, при создании очень длинных строк. Вместо того чтобы вводить длинные символьные строки в исходном коде неразрывно и допускать их автоматический перенос на новую строку, такие строки можно разбить на мелкие части, сцепляемые с помощью операции `+`, как показано ниже.

```
// Использовать сцепление во избежание длинных строк
class ConCat {
    public static void main(String args[]) {
        String longStr = "Это может быть очень длинная строка, " +
                        "которую следовало бы перенести на " +
                        "новую строку. Но благодаря сцеплению " +
                        "этого удастся избежать.";

        System.out.println(longStr);
    }
}
```

Сцепление символьных строк с другими типами данных

Символьные строки можно сцеплять с данными других типов. Рассмотрим в качестве примера следующую немного измененную версию приведенного ранее примера:

```
int age = 9;
String s = "Ему " + age + " лет.";
System.out.println(s);
```

В данном примере переменная `age` относится к типу `int`, а не `String`, но результат получается прежним. Так происходит потому, что значение типа `int` автоматически преобразуется в свое строковое представление в объекте типа `String`.

После этого символьные строки сцепляются, как и прежде. Компилятор преобразует операнд `age` в его строковый эквивалент, тогда как остальные операнды рассматриваемой здесь операции `+` являются экземплярами класса `String`.

Сцепляя операнды других типов данных с символьными строками в операциях сцепления, следует быть внимательным, иначе можно получить совершенно неожиданные результаты. Рассмотрим в качестве примера следующий фрагмент кода:

```
String s = "четыре: " + 2 + 2;  
System.out.println(s);
```

В данном примере выводится следующий результат сцепления:

четыре: 22

вместо ожидаемого:

четыре: 4

Дело в том, что благодаря предшествованию операций сначала выполняется сцепление символьной строки "четыре" со строковым представлением первого числа 2, а полученный результат сцепляется затем со строковым представлением второго числа 2. Для того чтобы выполнить сначала целочисленное сложение, следует заключить эту операцию в круглые скобки, как показано ниже. В итоге строка `s` будет содержать символы "четыре: 4".

```
String s = " четыре: " + (2 + 2);
```

Преобразование символьных строк и метод `toString()`

При сцеплении символьных строк в Java данные других типов преобразуются в их строковое представление путем вызова одного из перегружаемых вариантов метода преобразования `valueOf()`, определенного в классе `String`. Метод `valueOf()` перегружается для всех примитивных типов данных, а также для типа `Object`. В частности, для элементарных типов данных метод `valueOf()` возвращает символьную строку, содержащую удобочитаемый эквивалент того значения, с которым он был вызван. А для объектов метод `valueOf()` вызывает метод `toString()` вызываемого объекта. Подробнее о методе `valueOf()` речь пойдет далее в этой главе, а до тех пор рассмотрим метод `toString()` как удобное средство для строкового представления объектов создаваемых классов.

Метод `toString()` реализуется в каждом классе, поскольку он определен в классе `Object`. Но реализация метода `toString()` по умолчанию редко оказывается полезной. Поэтому во всех наиболее важных из создаваемых классов метод `toString()`, скорее всего, придется переопределить, чтобы обеспечить в каждом из них свое строковое представление. Правда, сделать это совсем не трудно, используя следующую общую форму метода `toString()`:

```
String toString()
```

Чтобы реализовать этот метод в своем классе, достаточно вернуть объект типа `String`, содержащий удобочитаемую символьную строку, надлежащим образом описывающую объект данного класса.

Переопределение метода `toString()` в создаваемых классах позволяет полностью интегрировать их в среду программирования на Java. Например, переопределенные варианты метода `toString()` можно применять в операторах `print()` и `println()`, а также в операциях сцепления символьных строк с данными других типов. В следующей программе эта особенность демонстрируется на примере переопределения метода `toString()` в классе `Box`:

```
// Переопределить метод toString() в классе Box
class Box {
    double width;
    double height;
    double depth;
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
    public String toString() {
        return "Размеры " + width + " на " +
            depth + " на " + height + ".";
    }
}
class toStringDemo {
    public static void main(String args[]) {
        Box b = new Box(10, 12, 14);
        String s = "Объект b типа Box: " + b; // выполнить сцепление
        // символьной строки с объектом типа Box
        System.out.println(b); // преобразовать объект типа Box
        // в символьную строку при выводе
        System.out.println(s);
    }
}
```

Эта программа выводит следующий результат:

```
Размеры 10.0 на 14.0 на 12.0
Объект Box типа b: Размеры 10.0 на 14.0 на 12.0
```

Извлечение символов

В классе `String` предоставляется немало способов извлечь символы из объекта типа `String`. Рассмотрим некоторые из них. И хотя символы, составляющие строку, нельзя индексировать таким же образом, как и в символьных массивах, тем не менее, для выполнения операций во многих методах из класса `String` применяется индекс (или смещение) в символьной строке. Как и массивы, символьные строки индексируются начиная с нуля.

Метод `charAt()`

Чтобы извлечь из строки единственный символ, достаточно обратиться к нему непосредственно, вызвав метод `charAt()`. Ниже приведена общая форма этого метода.

```
char charAt(int где)
```

Здесь параметр *где* обозначает индекс символа, который требуется получить. Значение параметра *где* должно быть неотрицательным и указывать положение извлекаемого символа в строке. Метод `charAt()` возвращает символ, находящийся в указанном положении. В следующем примере строковое значение "b" присваивается переменной *ch*.

```
char ch;
ch = "abc".charAt(1);
```

Метод `getChars()`

Если требуется извлечь несколько символов сразу, то для этой цели можно вызвать метод `getChars()`. Ниже приведена общая форма этого метода.

```
void getChars(int начало_источника, int конец_источника,
              char адресат[], int начало_адресата)
```

Здесь параметр *начало_источника* обозначает индекс начала подстроки, а параметр *конец_источника* — индекс символа, следующего после конца извлекаемой подстроки. Таким образом, извлекается подстрока, содержащая символы в пределах от *начало_источника* до *конец_источника*-1. Массив, принимающий извлекаемые символы, задается в качестве параметра *адресат*, а индекс, начиная с которого извлекаемая подстрока будет копироваться в указанный массив *адресат*, — в качестве параметра *начало_адресата*. Следует, однако, принять меры, чтобы указанный массив *адресат* оказался достаточного размера и в нем разместились все символы из заданной подстроки.

В следующем примере программы демонстрируется применение метода `getChars()`:

```
class getCharsDemo {
    public static void main(String args[]) {
        String s = "Это демонстрация метода getChars().";
        int start = 4;
        int end = 8;
        char buf[] = new char[end - start];
        s.getChars(start, end, buf, 0);
        System.out.println(buf);
    }
}
```

Ниже приведен результат, выводимый данной программой.

демо

Метод `getBytes()`

В качестве альтернативы методу `getChars()` предоставляется метод `getBytes()`, сохраняющий символы в массиве байтов. В этом методе выполняется преобразование символов в байты, выбираемое на конкретной платформе по умолчанию. Ниже приведена простейшая форма этого метода.

```
byte[] getBytes()
```

Имеются и другие формы метода `getBytes()`. Этот метод применяется, главным образом, в тех случаях, когда значения типа `String` экспортируются в те среды, где не поддерживаются 16-разрядные символы в Юникоде. Например, в большинстве сетевых протоколов Интернета и форматов текстовых файлов применяется 8-разрядный код ASCII для всех операций обмена текстовыми данными.

Метод `toCharArray()`

Если требуется преобразовать все символы из объекта типа `String` в символьный массив, то сделать это проще всего, вызвав метод `toCharArray()`. Этот метод возвращает массив символов из всей строки. Ниже приведена его общая форма.

```
char[] toCharArray()
```

Этот метод предоставляется в качестве дополнения, поскольку тот же самый результат можно получить, вызвав метод `getChars()`.

Сравнение символьных строк

В классе `String` имеется немало методов, предназначенных для сравнения символьных строк или подстрок в них. Рассмотрим некоторые из этих методов.

Методы `equals()` и `equalsIgnoreCase()`

Для сравнения двух символьных строк на равенство достаточно вызвать метод `equals()`, который имеет следующую общую форму:

```
boolean equals(Object строка)
```

где параметр *строка* обозначает объект типа `String`, который сравнивается с вызывающим объектом типа `String`. Этот метод возвращает логическое значение `true`, если сравниваемые строки содержат те же самые символы и в том же порядке, а иначе — логическое значение `false`. Сравнение выполняется с учетом регистра.

Для сравнения символьных строк без учета регистра символов достаточно вызвать метод `equalsIgnoreCase()`. При сравнении двух символьных строк наборы символов `A-Z` и `a-z` в этом методе считаются одинаковыми. Он имеет следующую общую форму:

```
boolean equalsIgnoreCase(Object строка)
```

где параметр *строка* обозначает объект типа `String`, который сравнивается с вызывающим объектом типа `String`. Этот метод возвращает логическое значение `true`, если сравниваемые строки содержат одинаковые символы в том же самом порядке, а иначе — логическое значение `false`.

Ниже приведен пример программы, в котором демонстрируется применение методов `equals()` и `equalsIgnoreCase()`.

```
// Продемонстрировать применение методов equals()
// и equalsIgnoreCase()
```

```

class equalsDemo {
    public static void main(String args[]) {
        String s1 = "Привет";
        String s2 = "Привет";
        String s3 = "Прощай";
        String s4 = "ПРИВЕТ";
        System.out.println(s1 + " равно " + s2 + " -> " +
                           s1.equals(s2));
        System.out.println(s1 + " равно " + s3 + " -> " +
                           s1.equals(s3));
        System.out.println(s1 + " равно " + s4 + " -> " +
                           s1.equals(s4));
        System.out.println(s1 + " равно без учета регистра " +
                           s4 + " -> " + s1.equalsIgnoreCase(s4));
    }
}

```

Эта программа выводит следующий результат:

```

Привет равно Привет -> true
Привет равно Прощай -> false
Привет равно ПРИВЕТ -> false
Привет равно без учета регистра ПРИВЕТ -> true

```

Метод `regionMatches()`

Этот метод сравнивает одну заданную часть символьной строки с другой ее частью. Имеется также перегружаемая форма метода `regionMatches()` для сравнения частей символьной строки без учета регистра. Ниже приведены общие формы этого метода.

```

boolean regionMatches(int начальный_индекс, String строка2,
                     int индекс_начала_строки2, int количество_символов)

```

```

boolean regionMatches(boolean игнорировать_регистр, int начальный_индекс,
                     String строка2, int индекс_начала_строки2,
                     int количество_символов)

```

В обеих формах данного метода параметр *начальный_индекс* обозначает индекс начала той части символьной строки из вызывающего объекта типа `String`, с которой сравнивается символьная строка, задаваемая в качестве параметра *строка2*. Индекс символа, начиная с которого должна сравниваться заданная строка 2, задается в качестве параметра *индекс_начала_строки2*, а длина сравниваемой подстроки — в качестве параметра *количество_символов*. Если во второй форме данного метода параметр *игнорировать_регистр* принимает логическое значение `true`, то сравнение подстрок выполняется без учета регистра. В противном случае регистр учитывается.

Методы `startsWith()` и `endsWith()`

В классе `String` определены два метода, являющиеся в большей или меньшей степени специализированными формами метода `regionMatches()`. Так, в методе `startsWith()` определяется, начинается ли заданный объект типа `String` с ука-

занной символьной строки, а в методе `endsWith()` — завершается ли объект типа `String` заданной подстрокой. Ниже приведены общие формы этих методов.

```
boolean startsWith(String строка)
boolean endsWith(String строка)
```

Здесь параметр *строка* обозначает подстроку, наличие которой проверяется в начале или в конце вызывающего объекта типа `String` соответственно. Если эта подстрока присутствует в данном месте вызывающего объекта типа `String`, то возвращается логическое значение `true`, а иначе — логическое значение `false`. Например, в результате следующих вызовов:

```
"Foobar".endsWith("bar")
```

и

```
"Foobar".startsWith("Foo")
```

возвращается логическое значение `true`.

Во второй, приведенной ниже форме метода `startsWith()` можно задать начальную точку для поиска заданной подстроки.

```
boolean startsWith(String строка, int начальный_индекс)
```

где *начальный_индекс* обозначает индекс символа, с которого начинается поиск заданной подстроки в исходной строке. Например, в результате следующего вызова:

```
"Foobar".startsWith("bar", 3)
```

возвращается логическое значение `true`.

Метод `equals()` в сравнении с операцией `==`

Следует иметь в виду, что метод `equals()` и операция `==` выполняют разные действия. Как пояснялось ранее, метод `equals()` сравнивает символы из объекта типа `String`, тогда как операция `==` — две ссылки на объекты и определяет, ссылаются ли они на один и тот же экземпляр. В следующем примере программы показано, что два разных объекта типа `String` могут содержать одинаковые символы, но ссылки на эти объекты при сравнении не будут равнозначными:

```
// Метод equals() в сравнении с операцией ==
class EqualsNotEqualTo {
    public static void main(String args[]) {
        String s1 = "Привет";
        String s2 = new String(s1);

        System.out.println(s1 + " равно " + s2 + " -> " +
                           s1.equals(s2));

        System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
    }
}
```

Переменная `s1` ссылается на экземпляр класса `String`, созданный присваиванием ей строкового литерала "Привет". А объект, на который ссылается переменная `s2`, создается с использованием переменной `s1` в качестве инициализатора. Таким образом, содержимое обоих объектов типа `String` одинаково, но это раз-

ные объекты. Следовательно, переменные `s1` и `s2` ссылаются *не* на один и тот же объект, и поэтому они не равны (при сравнении в операции `==`), как доказывает приведенный ниже результат, выводимый данной программой.

```
Привет равно Привет -> true
Привет == Привет -> false
```

Метод `compareTo()`

Зачастую недостаточно знать, что символьные строки одинаковы. В прикладных программах, выполняющих сортировку, обычно требуется выяснить, оказывается ли текущая символьная строка *меньше*, *больше* или *равной* следующей строке. Одна символьная строка меньше другой, если она следует *перед* ней в лексикографическом порядке, и больше другой, если она следует *после* нее. Для этой цели служит метод `compareTo()`, определенный в интерфейсе `Comparable<T>`, реализуемом в классе `String`. Этот метод имеет следующую общую форму:

```
int compareTo(String строка)
```

Здесь параметр *строка* обозначает объект типа `String`, сравниваемый с вызывающим объектом типа `String`. Возвращаемый результат сравнения символьных строк интерпретируется так, как показано в табл. 16.1.

Таблица 16.1. Результат сравнения символьных строк в методе `compareTo()`

Значение	Описание
Меньше нуля	Вызывающая символьная строка меньше строки str
Больше нуля	Вызывающая символьная строка больше строки str
Нуль	Символьные строки равны

В приведенном ниже примере программы сортируется массив символьных строк. Для определения порядка пузырьковой сортировки в этой программе используется метод `compareTo()`.

```
// Пузырьковая сортировка объектов типа String
class SortString {
    static String arr[] = {
        "Now", "is", "the", "time", "for", "all", "good", "men",
        "to", "come", "to", "the", "aid", "of", "their", "country"
    };
    public static void main(String args[]) {
        for(int j = 0; j < arr.length; j++) {
            for(int i = j + 1; i < arr.length; i++) {
                if(arr[i].compareTo(arr[j]) < 0) {
                    String t = arr[j];
                    arr[j] = arr[i];
                    arr[i] = t;
                }
            }
            System.out.println(arr[j]);
        }
    }
}
```

Эта программа выводит отсортированные слова в следующем порядке:

```
Now  
aid  
all  
come  
country  
for  
good  
is  
men  
of  
the  
the  
their  
time  
to  
to
```

Как следует из результата выполнения данной программы, сравнение символьных строк в методе `compareTo()` происходит с учетом регистра. В частности, слово "Now" следует прежде всех остальных слов, поскольку оно начинается с прописной буквы, а прописная буква имеет меньшее значение в наборе символов в коде ASCII.

Если символьные строки требуется сравнивать без учета регистра, то для этой цели служит метод `compareToIgnoreCase()`. Ниже приведена его общая форма.

```
int compareToIgnoreCase(String строка)
```

Этот метод возвращает такой же результат, как и метод `compareTo()`, но только без учета регистра символов. Попробуйте ввести этот метод в приведенный выше пример программы. В итоге слово "Now" уже не появится первым в списке отсортированных слов.

Поиск в символьных строках

В классе `String` предоставляются два метода для поиска в символьной строке определенного символа или подстроки.

- Метод `indexOf()` — находит первое вхождение символа или подстроки.
- Метод `lastIndexOf()` — находит последнее вхождение символа или подстроки.

Оба эти метода перестраиваются несколькими способами и по-разному. Но в любом случае они возвращают позицию в строке (индекс), где найден символ или подстрока, а при неудачном исходе поиска — значение `-1`.

Для поиска первого вхождения символа в строке служит следующая форма:

```
int indexOf(char символ)
```

А для поиска последнего вхождения символа в строке служит такая форма:

```
int lastIndexOf(char символ)
```

где параметр `символ` обозначает искомый символ в строке.

И наконец, для поиска первого или последнего вхождения подстроки служит приведенная ниже форма, где параметр *строка* обозначает искомую подстроку.

```
int indexOf(String строка)
int lastIndexOf(String строка)
```

Воспользовавшись следующими формами рассматриваемых здесь методов, можно указать начальную позицию для поиска символа или подстроки в исходной строке:

```
int indexOf(int символ, int начальный_индекс)
int lastIndexOf(int символ, int начальный_индекс)

int indexOf(String строка, int начальный_индекс)
int lastIndexOf(String строка, int начальный_индекс)
```

где параметр *начальный_индекс* задает начальную позицию для поиска в строке. В методе `indexOf()` поиск начинается от позиции *начальный_индекс* и до конца строки, а в методе `lastIndexOf()` — от позиции *начальный_индекс* и до нуля.

В следующем примере программы демонстрируется применение различных форм методов индексирования для поиска символов и подстрок в исходной строке:

```
// Продемонстрировать применение разных форм
// методов indexOf() и lastIndexOf()
class indexDemo {
    public static void main(String args[]) {
        String s = "Now is the time for all good men " +
            "to come to the aid of their country.";
        System.out.println(s);
        System.out.println("indexOf(t) = " +
            s.indexOf('t'));
        System.out.println("lastIndexOf(t) = " +
            s.lastIndexOf('t'));
        System.out.println("indexOf(the) = " +
            s.indexOf("the"));
        System.out.println("lastIndexOf(the) = " +
            s.lastIndexOf("the"));
        System.out.println("indexOf(t, 10) = " +
            s.indexOf('t', 10));
        System.out.println("lastIndexOf(t, 60) = " +
            s.lastIndexOf('t', 60));
        System.out.println("indexOf(the, 10) = " +
            s.indexOf("the", 10));
        System.out.println("lastIndexOf(the, 60) = " +
            s.lastIndexOf("the", 60));
    }
}
```

Ниже приведен результат, выводимый данной программой.

```
Now is the time for all good men to come to the aid of their country.
indexOf(t) = 7
lastIndexOf(t) = 65
indexOf(the) = 7
lastIndexOf(the) = 55
indexOf(t, 10) = 11
lastIndexOf(t, 60) = 55
indexOf(the, 10) = 44
lastIndexOf(the, 60) = 55
```

Видоизменение символьных строк

Объекты типа `String` неизменяемы, и поэтому всякий раз, когда требуется их видоизменить, их содержимое следует скопировать в объект типа `StringBuffer`

или `StringBuilder` или же воспользоваться одним из методов из класса `String`, создающих новые копии символьных строк с внесенными изменениями. В этом разделе рассматриваются простейшие из этих методов.

Метод `substring()`

Чтобы извлечь подстроку из символьной строки, достаточно вызвать метод `substring()`, у которого имеются две формы. Первая его форма такова:

`String substring(int начальный_индекс)`

где параметр *начальный_индекс* обозначает позицию, с которой должна начинаться подстрока. Эта форма возвращает копию подстроки, которая начинается с позиции *начальный_индекс* и продолжается до завершения вызывающей строки.

Вторая форма метода `substring()` позволяет указать как начальный, так и конечный индекс подстроки следующим образом:

`String substring(int начальный_индекс, int конечный_индекс)`

где параметр *начальный_индекс* обозначает позицию, с которой должна начинаться извлекаемая подстрока, а *конечный_индекс* — позицию, на которой должна оканчиваться извлекаемая подстрока. Возвращаемая подстрока содержит все символы, от первой позиции и до последней, но исключая последнюю.

В следующем примере программы метод `substring()` используется для замены в исходной символьной строке всех экземпляров одной подстроки другой подстрокой.

```
// Замена подстроки
class StringReplace {
    public static void main(String args[]) {
        String org = "This is a test. This is, too.";
        String search = "is";
        String sub = "was";
        String result = "";
        int i;
        do { // заменить все совпадающие подстроки
            System.out.println(org);
            i = org.indexOf(search);
            if(i != -1) {
                result = org.substring(0, i);
                result = result + sub;
                result = result + org.substring(i + search.length());
                org = result;
            }
        } while(i != -1);
    }
}
```

Ниже приведен результат, выводимый данной программой.

```
This is a test. This is, too.
Thwas is a test. This is, too.
Thwas was a test. This is, too.
Thwas was a test. Thwas is, too.
Thwas was a test. Thwas was, too.
```

Метод `concat()`

Чтобы соединить две подстроки, достаточно вызвать метод `concat()`. Ниже приведена его общая форма:

```
String concat(String строка)
```

Этот метод создает новый строковый объект, содержащий вызываемую строку, в конце которой добавляется содержимое параметра *строка*. Метод `concat()` выполняет то же действие, что и операция `+`, т.е. сцепление символьных строк. Например, в следующем фрагменте кода соединенная строка "onetwo" присваивается переменной `s2`:

```
String s1 = "one";
String s2 = s1.concat("two");
```

Этот фрагмент кода дает такой же результат, как и приведенный ниже фрагмент кода.

```
String s1 = "one";
String s2 = s1 + "two";
```

Метод `replace()`

У этого метода имеются две формы. В первой форме все вхождения одного символа в исходной строке заменяются другим символом:

```
String replace(char исходный, char заменяемый)
```

где параметр *исходный* обозначает заменяемый символ, а параметр *заменяемый* — заменяющий. В результате замены возвращается видоизмененная символьная строка. Например, в следующей строке кода:

```
String s = "Hello".replace('l', 'w');
```

переменной `s` присваивается символьная строка "Hewwo", получившее в результате замены символа 'l' на 'w' в исходной строке. А во второй форме метода `replace()` одна последовательность символов заменяется другой следующим образом:

```
String replace(CharSequence исходная, CharSequence заменяемая)
```

Метод `trim()`

Этот метод возвращает копию вызывающей символьной строки, из которой удалены все начальные и конечные пробелы. Он имеет следующую общую форму:

```
String trim()
```

Ниже приведен пример применения метода `trim()`. В итоге переменной `s` присваивается символьная строка "Здравствуй, мир".

```
String s = "    Здравствуй, мир    ".trim();
```

Метод `trim()` очень удобно вызывать для обработки команд, вводимых пользователем. Так, в приведенном ниже примере программы пользователю сначала предлагается ввести название штата, а затем выводится название города — столи-

цы штата. Метод `trim()` используется в этой программе для удаления всех начальных и конечных пробелов, которые могут быть непреднамеренно введены пользователем.

```
// Использовать метод trim() для обработки команд,
// вводимых пользователем
import java.io.*;
class UseTrim {
    public static void main(String args[])
        throws IOException
    {
        // создать буферизированный поток чтения типа BufferedReader,
        // используя стандартный поток ввода System.in
        BufferedReader br = new
            BufferedReader(new InputStreamReader(System.in));
        String str;
        System.out.println("Введите 'стоп' для завершения.");
        System.out.println("Введите название штата: ");
        do {
            str = br.readLine();
            str = str.trim(); // удалить пробелы
            if (str.equals("Иллинойс"))
                System.out.println("Столица - Спрингфилд.");
            else if (str.equals("Миссури"))
                System.out.println("Столица - Джефферсон-сити.");
            else if (str.equals("Калифорния"))
                System.out.println("Столица - Сакраменто.");
            else if (str.equals("Вашингтон"))
                System.out.println("Столица - Олимпия.");
            // ...
        } while (!str.equals("стоп"));
    }
}
```

Преобразование данных методом `valueOf()`

Метод `valueOf()` преобразует данные из внутреннего представления в удобочитаемую форму. Этот статический метод перегружается в классе `String` для всех встроенных в Java типов данных таким образом, чтобы каждый тип был правильно преобразован в символьную строку. Метод `valueOf()` перегружается и для типа `Object`, поэтому объект типа любого создаваемого класса также может использоваться в качестве аргумента. (Напомним, что класс `Object` является суперклассом для всех остальных классов.) Ниже приведены некоторые формы метода `valueOf()`.

```
static String valueOf(double число)
static String valueOf(long число)
static String valueOf(Object объект)
static String valueOf(char символы[])
```

Как упоминалось ранее, метод `valueOf()` вызывается в том случае, если требуется строковое представление некоторого другого типа данных, например, в операциях сцепления символьных строк. Этот метод можно вызывать и непосредственно с любым типом данных, чтобы получить подходящее строковое представление этого типа данных. Все примитивные типы данных преобразуются

в их общее строковое представление. Для любого объекта, передаваемого методу `valueOf()`, возвращается результат вызова метода `toString()`. На самом деле тот же самый результат можно получить, просто вызвав метод `toString()`.

Для большинства массивов метод `valueOf()` возвращает зашифрованную символьную строку, которая обозначает, что это массив определенного типа. Но для массивов типа `char` создается объект типа `String`, содержащий все символы из массива типа `char`. Для этой цели служит следующая форма метода `valueOf()`:

```
static String valueOf(char символы[], int начальный_индекс,
                     int количество_символов)
```

где параметр *символы* обозначает массив, содержащий символы, параметр *начальный_индекс* — позицию в массиве, с которой начинается подстрока, а параметр *количество_символов* — длину подстроки.

Изменение регистра символов в строке

Метод `toLowerCase()` преобразует все символы строки из верхнего регистра в нижний, а метод `toUpperCase()` — из нижнего регистра в верхний. Небуквенные символы, например десятичные цифры, остаются без изменения. Ниже приведены простейшие формы этих методов.

```
String toLowerCase()
String toUpperCase()
```

Оба метода возвращают объект типа `String`, содержащий эквивалент вызываемой строки в нижнем или верхнем регистре символов соответственно. В обоих случаях преобразование выполняется с учетом региональных настроек по умолчанию.

Ниже приведен пример программы, демонстрирующий применение методов `toLowerCase()` и `toUpperCase()`.

```
// Продемонстрировать применение методов toUpperCase() и toLowerCase()
class ChangeCase {
    public static void main(String args[])
    {
        String s = "Это тест.";
        System.out.println("Исходная строка: " + s);
        String upper = s.toUpperCase();
        String lower = s.toLowerCase();
        System.out.println("Верхний регистр: " + upper);
        System.out.println("Нижний регистр: " + lower);
    }
}
```

Эта программа выводит следующий результат:

```
Исходная строка: Это тест
Верхний регистр: ЭТО ТЕСТ
Нижний регистр: это тест
```

Следует также иметь в виду, что имеются и перегружаемые варианты методов `toLowerCase()` и `toUpperCase()`, позволяющие определять объект типа `Locale` для управления преобразованием. В некоторых случаях определение региональ-

ных настроек может иметь особое значение и способствовать интернационализации прикладной программы.

Соединение символьных строк

В версию JDK 8 в класс `String` был внедрен новый метод `join()`, предназначенный для соединения двух и более символьных строк, разграничиваемых указанным разделителем, например, пробелом или запятой. У этого метода имеются две формы. Ниже приведена первая из них.

`static String join(CharSequence разделитель, CharSequence . . . строки)`

Здесь параметр *разделитель* обозначает знак, используемый для разделения последовательностей символов, задаваемых в качестве параметра *строки*. Благодаря тому что в классе `String` реализуется интерфейс `CharSequence`, в качестве параметра *строки* может быть указан список символьных строк. (Подробнее об интерфейсе `CharSequence` речь пойдет в главе 17.). В следующем примере программы демонстрируется применение первой формы метода `join()`:

```
// Продемонстрировать применение метода join(),
// определенного в классе String
class StringJoinDemo {
    public static void main(String args[]) {

        String result = String.join(" ", "Alpha", "Beta", "Gamma");
        System.out.println(result);

        result = String.join(", ", "John", "ID#: 569",
                                "E-mail: John@HerbSchildt.com");
        System.out.println(result);
    }
}
```

Эта программа выводит следующий результат:

```
Alpha Beta Gamma
John, ID#: 569, E-mail: John@HerbSchildt.com
```

При первом вызове метода `join()` вводится пробел между каждой из символьных строк, а при втором его вызове — запятая с пробелом. Данный пример наглядно показывает, что разделитель совсем не обязательно должен быть одиночным знаком.

Вторая форма метода `join()` позволяет соединить список символьных строк, получаемых из объекта класса, реализующего интерфейс `Iterable`. Среди прочего, интерфейс `Iterable` реализуется в классах из каркаса коллекций `Collections Framework`, рассматриваемого в главе 18. Подробнее об интерфейсе `Iterable` речь пойдет в главе 17.

Дополнительные методы из класса `String`

Помимо представленных выше методов, в классе `String` имеется также целый ряд других методов, включая и перечисленные в табл. 16.2.

Таблица 16.2. Дополнительные методы из класса `String`

Метод	Описание
<code>int codePointAt(int i)</code>	Возвращает кодовую точку в Юникоде на позиции <i>i</i>
<code>int codePointBefore(int i)</code>	Возвращает кодовую точку в Юникоде на позиции, предшествующей <i>i</i>
<code>int codePointCount(int начало, int конец)</code>	Возвращает количество кодовых точек в части вызывающей символьной строки от позиции <i>начало</i> и до позиции <i>конец-1</i>
<code>boolean contains(CharSequence строка)</code>	Возвращает логическое значение true , если вызывающий объект содержит указанную <i>строку</i> , а иначе — логическое значение false
<code>boolean contentEquals(CharSequence строка)</code>	Возвращает логическое значение true , если вызывающий объект содержит указанную <i>строку</i> , а иначе — логическое значение false
<code>boolean contentEquals(StringBuffer строка)</code>	Возвращает логическое значение true , если вызывающий объект содержит указанную <i>строку</i> , а иначе — логическое значение false
<code>static String format(String форматизирующая_строка, Object ... аргументы)</code>	Возвращает символьную строку, отформатированную так, как определяет заданная <i>форматизирующая_строка</i> . (Подробнее о форматировании — в главе 19.)
<code>static String format(Locale регион, String форматизирующая_строка, Object ... аргументы)</code>	Возвращает символьную строку, отформатированную так, как определяет заданная <i>форматизирующая_строка</i> . (Подробнее о форматировании см. в главе 19.)
<code>boolean isEmpty()</code>	Возвращает логическое значение true , если вызывающая строка не содержит символы и имеет нулевую длину
<code>boolean matches(строка регулярное_выражение)</code>	Возвращает логическое значение true , если вызывающая строка совпадает с заданным <i>регулярным_выражением</i> , а иначе — логическое значение false
<code>int offsetByCodePoints(int начало, int число)</code>	Возвращает индекс позиции в вызывающей строке, которая отстоит на заданное <i>число</i> кодовых точек от начальной позиции, задаваемой по индексу <i>начало</i>
<code>String replaceFirst(строка регулярное_выражение, String новая_строка)</code>	Возвращает символьную строку, в которой первая подстрока, совпадающая с заданным <i>регулярным_выражением</i> , заменяется <i>новой_строкой</i>
<code>String replaceAll(строка регулярное_выражение, String новая_строка)</code>	Возвращает символьную строку, в которой все подстроки, совпадающие с заданным <i>регулярным_выражением</i> , заменяются <i>новой_строкой</i>
<code>String[] split(строка регулярное_выражение)</code>	Разбивает вызывающую строку на части и возвращает массив, содержащий результат. Каждая часть разделяется заданным <i>регулярным_выражением</i>

Окончание табл. 16.2

Метод	Описание
String[] split(String <i>регулярное_выражение</i>, int <i>максимум</i>)	Разбивает вызывающую строку на части и возвращает массив, содержащий результат. Каждая часть разделяется заданным <i>регулярным выражением</i> . Количество частей определяется параметром <i>максимум</i> . Если параметр <i>максимум</i> принимает отрицательное значение, вызывающая строка разбивается полностью. Если же параметр <i>максимум</i> принимает неотрицательное значение, то последний элемент возвращаемого массива содержит остаток вызывающей строки. А если значение параметра <i>максимум</i> равно нулю, то и в этом случае вызывающая строка разбивается полностью
CharSequence subSequence(int <i>начальный_индекс</i>, int <i>конечный_индекс</i>)	Возвращает подстроку из вызывающей строки, начиная с позиции <i>начальный_индекс</i> и кончая позицией <i>конечный_индекс</i> . Этот метод требуется для интерфейса CharSequence , реализуемого в классе String

Обратите внимание на то, что некоторые из перечисленных выше методов оперируют регулярными выражениями, которые мы рассмотрим в главе 30.

Класс StringBuffer

Этот класс подобен классу String, в котором предоставляется большая часть функциональных возможностей для обработки символьных строк. Как вам должно быть уже известно, класс String представляет неизменяемые последовательности символов постоянной длины, тогда как класс StringBuffer — расширяемые и доступные для изменений последовательности символов. Он позволяет вставлять символы и подстроки в середину исходной строки или добавлять их в ее конце. Объект типа StringBuffer автоматически наращивается, чтобы предоставить место для подобных расширений, и зачастую для возможности такого наращивания он содержит больше предварительно определенных символов, чем требуется на самом деле.

Конструкторы класса StringBuffer

В классе StringBuffer определены следующие четыре конструктора:

```
StringBuffer()
StringBuffer(int размер)
StringBuffer(String строка)
StringBuffer(CharSequence символы)
```

Первый конструктор по умолчанию (без параметров) резервирует место для 16 символов, не перераспределяя память. Второй конструктор принимает целочисленный аргумент, явно задающий размер буфера. Третий конструктор принимает аргумент типа String, задающий начальное содержимое объекта типа StringBuffer и резервирующий место для 16 символов, не перераспределяя память. Класс StringBuffer выделяет место для 16 дополнительных символов, если

не указывается конкретный размер буфера, чтобы сэкономить время, затрачиваемое на перераспределение памяти. Кроме того, частое перераспределение памяти может привести к ее фрагментации. Выделяя место под несколько дополнительных символов, класс `StringBuffer` снижает количество требующихся повторных перераспределений памяти. Четвертый конструктор создает объект, содержащий последовательность символов, задаваемых в качестве параметра *символы*, а также резервирует место для 16 дополнительных символов.

Методы `length()` и `capacity()`

Текущую длину объекта типа `StringBuffer` можно получить, вызвав метод `length()`, а текущий объем выделенной памяти — вызвав метод `capacity()`. Эти методы имеют следующие общие формы:

```
int length()
int capacity()
```

В приведенном ниже примере демонстрируется применение методов `length()` и `capacity()`.

```
// Сравнить методы length() и capacity() из класса StringBuffer
class StringBufferDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Hello");
        System.out.println("буфер = " + sb);
        System.out.println("длина = " + sb.length());
        System.out.println("емкость = " + sb.capacity());
    }
}
```

Ниже приведен результат, выводимый этой программой. Он показывает, каким образом класс `StringBuffer` резервирует свободное пространство для дополнительных манипуляций с символьными строками.

```
буфер = Hello
длина = 5
емкость = 21
```

Переменная `sb` инициализируется строковым значением "Hello" при ее создании, поэтому длина буфера для хранения этого значения равна 5. А объем выделяемой памяти (емкость буфера) составляет 21, поскольку 16 дополнительных символов добавляются автоматически.

Метод `ensureCapacity()`

Если требуется предварительно выделить место для определенного количества символов после создания объекта типа `StringBuffer`, то можно воспользоваться методом `ensureCapacity()`, чтобы установить емкость буфера. Это удобно, если заранее известно, что к объекту типа `StringBuffer` предполагается присоединить большое количество мелких символьных строк. Метод `ensureCapacity()` имеет следующую общую форму:

```
void ensureCapacity(int минимальная_емкость)
```

где параметр *минимальная_емкость* обозначает минимальный размер буфера. (Буфера, размер которых превышает заданную *минимальную_емкость*, также могут быть выделены из соображений эффективности.)

Метод `setLength()`

Для задания длины символьной строки в объекте типа `StringBuffer` служит метод `setLength()`, общая форма которого выглядит следующим образом:

```
void setLength(int длина)
```

где параметр *длина* обозначает конкретную длину символьной строки. Ее значение должно быть неотрицательным.

Когда увеличивается длина символьной строки, в конце существующей строки добавляются пустые символы. Если метод `setLength()` вызывается со значением меньше текущего значения, возвращаемого методом `length()`, то символы, оказавшиеся за пределами вновь заданной длины строки, будут удалены. В примере программы, представленном в следующем разделе, метод `setLength()` применяется для сокращения объекта типа `StringBuffer`.

Методы `charAt()` и `setCharAt()`

Значение отдельного символа можно извлечь из объекта типа `StringBuffer`, вызвав метод `charAt()`. А значение символа в объекте типа `StringBuffer` можно установить с помощью метода `setCharAt()`. Ниже приведены общие формы этих методов.

```
char charAt(int где)
void setCharAt(int где, char символ)
```

В форме метода `charAt()` параметр *где* обозначает индекс извлекаемого символа, а в форме метода `setCharAt()` — индекс задаваемого символа, тогда как параметр *символ* — значение этого символа. Значение параметра *где* для обоих методов должно быть неотрицательным и не должно указывать место за пределами символьной строки.

В следующем примере программы демонстрируется применение методов `charAt()` и `setCharAt()`:

```
// Продемонстрировать применение методов charAt() и setCharAt()
class setCharAtDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Hello");
        System.out.println("буфер до = " + sb);
        System.out.println("до вызова charAt(1) = " + sb.charAt(1));

        sb.setCharAt(1, 'i');
        sb.setLength(2);
        System.out.println("буфер после = " + sb);
        System.out.println("после вызова charAt(1) = " + sb.charAt(1));
    }
}
```

Эта программа выводит следующий результат:

```

буфер до = Hello
до вызова charAt(1) = e
буфер после = Hi
после вызова charAt(1) = i

```

Метод `getChars()`

Для копирования подстроки из объекта типа `StringBuffer` в массив служит метод `getChars()`, имеющий следующую общую форму:

```

void getChars(int начало_источника, int конец_источника,
               char адресат[], int начало_адресата)

```

где параметр *начало_источника* обозначает индекс начала подстроки, а параметр *конец_источника* — индекс символа, следующего после конца требуемой подстроки. Это означает, что подстрока содержит символы от позиции *начало_источника* до позиции *конец_источника*-1. Массив, принимающий символы, передается в качестве параметра *адресат*, а индекс массива, куда копируется подстрока, — в качестве параметра *начало_адресата*. Следует принять меры к тому, чтобы массив *адресат* имел достаточный размер, позволяющий вместить количество символов из указанной подстроки.

Метод `append()`

Метод `append()` присоединяет строковое представление любого другого типа данных в конце вызывающего объекта типа `StringBuffer`. У него имеется несколько перегружаемых вариантов. Ниже приведены некоторые из них.

```

StringBuffer append(String строка)
StringBuffer append(int число)
StringBuffer append(Object объект)

```

Строковое представление каждого параметра зачастую получается в результате вызова метода `String.valueOf()`. Полученный результат присоединяется к текущему объекту типа `StringBuffer`. Сам буфер возвращается каждым вариантом метода `append()`. Это позволяет соединить в цепочку несколько последовательных вызовов, как показано в следующем примере программы:

```

// Продемонстрировать применение метода append()
class appendDemo {
    public static void main(String args[]) {
        String s;
        int a = 42;
        StringBuffer sb = new StringBuffer(40);

        s = sb.append("a = ").append(a).append("!\n").toString();
        System.out.println(s);
    }
}

```

Ниже приведен результат, выводимый данной программой.

```

a = 42;

```

Метод `insert()`

Этот метод вставляет одну символьную строку в другую. Он перегружается таким образом, чтобы принимать в качестве параметра значения всех примитивных типов плюс объекты типа `String`, `Object` и `CharSequence`. Подобно методу `append()`, метод `insert()` получает строковое представление значения, с которым он вызывается. Эта строка затем вставляется в вызывающий объект типа `StringBuffer`. Ниже приведены некоторые общие формы метода `insert()`.

```
StringBuffer insert(int индекс, String строка)  
StringBuffer insert(int индекс, char символ)  
StringBuffer insert(int индекс, Object объект)
```

Здесь параметр *индекс* обозначает индекс позиции, на которой символьная строка будет вставлена в вызывающий объект типа `StringBuffer`. В следующем примере программы демонстрируется вставка слова "нравится" между словами "Мне" и "Java":

```
// Продемонстрировать применение метода insert()  
class insertDemo {  
    public static void main(String args[]) {  
        StringBuffer sb = new StringBuffer("Мне Java!");  
  
        sb.insert(4, "нравится ");  
        System.out.println(sb);  
    }  
}
```

Эта программа выводит следующий результат:

Мне нравится Java!

Метод `reverse()`

Изменить порядок следования символов в объекте типа `StringBuffer` на обратный можно с помощью метода `reverse()`. Общая форма которого приведена ниже.

```
StringBuffer reverse()
```

Этот метод возвращает объект с обратным порядком следования символов по сравнению с вызывающим объектом. В следующем примере программы демонстрируется применение метода `reverse()`:

```
// Изменить порядок следования символов в объекте  
// типа StringBuffer с помощью метода reverse()  
class ReverseDemo {  
    public static void main(String args[]) {  
        StringBuffer s = new StringBuffer("abcdef");  
  
        System.out.println(s);  
        s.reverse();  
        System.out.println(s);  
    }  
}
```

Ниже приведен результат, выводимый данной программой.

abcdef
fedcba

Методы `delete()` и `deleteCharAt()`

Удалить символы из объекта типа `StringBuffer` можно с помощью методов `delete()` и `deleteCharAt()`. Ниже приведены их общие формы.

```
StringBuffer delete(int начальный_индекс, int конечный_индекс)  
StringBuffer deleteCharAt(int позиция)
```

Метод `delete()` удаляет последовательность символов из вызывающего объекта. Его параметр *начальный_индекс* обозначает индекс первого символа, который требуется удалить, а параметр *конечный_индекс* — индекс символа, следующего за последним из удаляемых символов. Таким образом, удаляемая подстрока начинается с позиции *начальный_индекс* и оканчивается на позиции *конечный_индекс*-1. Из этого метода возвращается результирующий объект типа `StringBuffer`.

Метод `deleteCharAt()` удаляет символ на указанной *позиции*. Из этого метода возвращается результирующий объект типа `StringBuffer`.

В следующем примере программы демонстрируется применение методов `delete()` и `deleteCharAt()`:

```
// Продемонстрировать применение методов delete() и deleteCharAt()  
class deleteDemo {  
    public static void main(String args[]) {  
        StringBuffer sb = new StringBuffer("Это простой тест.");  
  
        sb.delete(3, 11);  
        System.out.println("После вызова delete(): " + sb);  
  
        sb.deleteCharAt(0);  
        System.out.println("После вызова deleteCharAt(): " + sb);  
    }  
}
```

Эта программа выводит следующий результат:

```
После вызова delete(): Это тест.  
После вызова deleteCharAt(): то тест.
```

Метод `replace()`

Вызвав метод `replace()`, можно заменить один набор символов другим в объекте типа `StringBuffer`. Ниже приведена общая форма этого метода.

```
StringBuffer replace(int начальный_индекс, int конечный_индекс,  
String строка)
```

Подстрока, которую требуется заменить, задается параметрами *начальный_индекс* и *конечный_индекс*. Таким образом, заменяется подстрока от символа на позиции *начальный_индекс* до символа на позиции *конечный_индекс*-1. А заменяющая строка передается в качестве параметра *строка*. Из этого метода возвращается результирующий объект типа `StringBuffer`.

В следующем примере программы демонстрируется применение метода `replace()`:

```
// Продемонстрировать применения метода replace()  
class replaceDemo {
```

```

public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("Это простой тест.");
    sb.replace(4, 8, "был");
    System.out.println("После замены: " + sb);
}

```

Ниже приведен результат, выводимый данной программой.

После замены: Это был тест.

Метод `substring()`

Вызвав метод `substring()`, можно получить часть содержимого объекта типа `StringBuffer`. У этого метода имеются две следующие формы:

```

String substring(int начальный_индекс)
String substring(int начальный_индекс, int конечный_индекс)

```

В первой форме этот метод возвращает подстроку, которая начинается с позиции *начальный_индекс* и продолжается до конца вызывающего объекта типа `StringBuffer`. А во второй форме он возвращает подстроку от позиции *начальный_индекс* и до позиции *конечный_индекс*-1. Эти формы метода `substring()` действуют таким же образом, как и рассмотренные ранее их аналоги из класса `String`.

Дополнительные методы из класса `StringBuffer`

Помимо описанных выше методов, класс `StringBuffer` содержит ряд других методов, включая и перечисленные в табл. 16.3.

Таблица 16.3. Дополнительные методы из класса `StringBuffer`

Метод	Описание
StringBuffer <code>appendCodePoint(int <i>символ</i>)</code>	Присоединяет кодовую точку в Юникоде в конце вызывающего объекта. Возвращает ссылку на объект
<code>int codePointAt(int <i>i</i>)</code>	Возвращает кодовую точку в Юникоде на позиции <i>i</i>
<code>int codePointBefore(int <i>i</i>)</code>	Возвращает кодовую точку в Юникоде на позиции, предшествующей <i>i</i>
<code>int codePointCount(int <i>начало</i>, int <i>конец</i>)</code>	Возвращает количество кодовых точек в части вызывающей строки от позиции <i>начало</i> и до позиции <i>конец</i> -1
<code>int indexOf(String <i>строка</i>)</code>	Выполняет поиск в вызывающем объекте типа StringBuffer первого вхождения <i>строки</i> . Возвращает индекс позиции при совпадении, а иначе — значение -1
<code>int indexOf(String <i>строка</i>, int <i>начальный_индекс</i>)</code>	Выполняет поиск в вызывающем объекте типа StringBuffer первого вхождения <i>строки</i> , начиная с позиции <i>начальный_индекс</i> . Возвращает индекс позиции при совпадении, а иначе — значение -1

Метод	Описание
<code>int lastIndexOf(String строка)</code>	Выполняет поиск в вызывающем объекте типа StringBuffer последнего вхождения строки . Возвращает индекс позиции при совпадении, а иначе — значение -1
<code>int lastIndexOf(String строка, int начальный_индекс)</code>	Выполняет поиск в вызывающем объекте типа StringBuffer последнего вхождения строки , начиная с позиции начальный_индекс . Возвращает индекс позиции при совпадении, а иначе — значение -1
<code>int offsetByCodePoints(int начало, int число)</code>	Возвращает индекс символа в вызывающей строке, который отстоит на заданное число кодовых точек от начального индекса, определяемого параметром начало
<code>CharSequence subSequence(int начальный_индекс, int конечный_индекс)</code>	Возвращает подстроку из вызывающей строки, начиная с позиции начальный_индекс и оканчивая позицией конечный_индекс . Этот метод требуется для интерфейса CharSequence , реализуемого в классе StringBuffer
<code>void trimToSize()</code>	Требует, чтобы размер символьного буфера вызывающего объекта был уменьшен для большего соответствия текущему содержимому

В следующем примере программы демонстрируется применение методов `indexOf()` и `lastIndexOf()`.

```
class IndexOfDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("one two one");
        int i;

        i = sb.indexOf("one");
        System.out.println("Индекс первого вхождения: " + i);

        i = sb.lastIndexOf("one");
        System.out.println("Индекс последнего вхождения: " + i);
    }
}
```

Эта программа выводит следующий результат:

```
Индекс первого вхождения: 0
Индекс последнего вхождения: 8
```

Класс **StringBuilder**

Класс **StringBuilder**, появившийся в версии JDK 5, — относительно недавнее дополнение функциональных возможностей, существовавших до него в Java для обработки строк. Класс **StringBuilder** ничем не отличается от класса **StringBuffer**, за исключением того, что он не синхронизирован, а следовательно, не является потокобезопасным. Применение класса **StringBuilder** дает выигрыш в производительности. Но в тех случаях, когда обращение к изменяемой строке происходит из нескольких потоков исполнения без внешней синхронизации, следует применять класс **StringBuffer**, а не **StringBuilder**.

Пакет java.lang

Эта глава посвящена классам и интерфейсам, определенным в пакете java.lang. Как вам должно быть уже известно, пакет java.lang автоматически импортируется во все программы. Он содержит классы и интерфейсы, которые составляют основу всех программ на Java. Пакет java.lang наиболее широко используется в Java и включает в себя следующие классы:

Boolean	Enum	Process	String
Byte	Float	ProcessBuilder	StringBuffer
Character	InheritableThreadLocal	ProcessBuilder.Redirect	StringBuilder
Character.Subset	Integer	Runtime	System
Character.UnicodeBlock	Long	RuntimePermission	Thread
Class	Math	SecurityManager	ThreadGroup
ClassLoader	Number	Short	ThreadLocal
ClassValue	Object	StackTraceElement	Throwable
Compiler	Package	StrictMath	Void
Double			

В данном пакете определены также два вложенных класса типа Character: Character.SubSet и Character.UnicodeBlock.

Ниже перечислены интерфейсы, определенные в пакете java.lang.

Appendable	Cloneable	Readable
AutoCloseable	Comparable	Runnable
CharSequence	Iterable	Thread.UncaughtExceptionHandler

Некоторые классы, входящие в пакет java.lang, содержат устаревшие методы, большинство из которых относится еще к версии Java 1.0. Эти методы все еще предоставляются в Java для поддержки постепенно выводимого из эксплуатации унаследованного кода и не рекомендуются для употребления в новом коде. Поэтому не рекомендованные к употреблению методы в этой главе не рассматриваются.

Оболочки примитивных типов

Как упоминалось в части I, примитивные типы данных наподобие `int` и `char` применяются в Java из соображений производительности и не являются частью объектной иерархии. Они передаются методам по значению и не могут быть переданы им по ссылке. Кроме того, из двух методов нельзя сослаться на *один и тот же экземпляр* типа `int`. Но рано или поздно возникает потребность в объектном представлении одного из примитивных типов данных. Например, существуют классы коллекций, предназначенные для обращения только с объектами (они обсуждаются в главе 18). Чтобы сохранить примитивный тип данных в одном из таких классов, нужно заключить в него этот примитивный тип. Для того чтобы удовлетворить потребность в этом для каждого примитивного типа данных, в Java предоставляется отдельный класс, обычно называемый *оболочкой типа*. Оболочки типов были представлены в главе 12, а здесь они рассматриваются более подробно.

Класс Number

Абстрактный класс `Number` является суперклассом, который реализуется в классах оболочек числовых типов `byte`, `short`, `int`, `long`, `float` и `double`. В классе `Number` имеются абстрактные методы, возвращающие значение объекта в разных числовых форматах. Например, метод `doubleValue()` возвращает значение как тип `double`, а метод `floatValue()` — как тип `float` и т.д. Все эти методы перечислены ниже.

```
byte byteValue()
double doubleValue()
float floatValue()
int intValue()
long longValue()
short shortValue()
```

Значения, возвращаемые этими методами, могут быть округлены, усечены или собраны в “мусор” вследствие сужающего преобразования. У класса `Number` имеются конкретные подклассы, содержащие явные значения каждого числового типа: `Double`, `Float`, `Byte`, `Short`, `Integer` и `Long`.

Классы Double и Float

Эти классы служат оболочками для числовых значений с плавающей точкой типа `double` и `float` соответственно. Ниже приведены конструкторы класса `Float`.

```
Float(double число)
Float(float число)
Float(String строка) throws NumberFormatException
```

Как видите, объекты типа `Float` должны быть созданы со значениями типа `float` или `double`. Они могут также быть созданы из строкового представления числа с плавающей точкой.

Ниже приведены конструкторы класса `Double`.

`Double(double число)`
`Double(String строка)` throws `NumberFormatException`

Объекты типа `Double` могут быть созданы из значения типа `double` или символической строки, содержащей значение с плавающей точкой.

Методы, определенные в классе `Float`, перечислены в табл. 17.1, а методы, определенные в классе `Double`, — в табл. 17.2. В классах `Float` и `Double` определяются следующие константы.

BYTES	Длина типа <code>float</code> или <code>double</code> в байтах (внедрена в версии JDK 8)
MAX_EXPONENT	Максимальный показатель степени
MAX_VALUE	Максимальное положительное значение
MIN_EXPONENT	Минимальный показатель степени
MIN_NORMAL	Минимальное положительное нормальное значение
MIN_VALUE	Минимальное положительное значение
NaN	Не число
POSITIVE_INFINITY	Положительная бесконечность
NEGATIVE_INFINITY	Отрицательная бесконечность
SIZE	Размер заключенного в оболочку значения в битах
TYPE	Объект типа <code>Class</code> для типов <code>float</code> и <code>double</code>

Таблица 17.1. Методы из класса `Float`

Метод	Описание
<code>byte byteValue()</code>	Возвращает значение вызывающего объекта как тип <code>byte</code>
<code>static int compare(float число1, float число2)</code>	Сравнивает значения <code>число1</code> и <code>число2</code> . Возвращает нулевое значение, если сравниваемые числовые значения равны; отрицательное значение, если <code>число1</code> меньше, чем <code>число2</code> ; или положительное значение, если <code>число1</code> больше, чем <code>число2</code>
<code>int compareTo(Float f)</code>	Сравнивает числовое значение вызывающего объекта со значением <code>f</code> . Возвращает нулевое значение, если сравниваемые числовые значения равны; отрицательное значение, если вызывающий объект имеет меньшее значение; или положительное значение, если вызывающий объект имеет большее значение
<code>double doubleValue()</code>	Возвращает значение вызывающего объекта как тип <code>double</code>
<code>boolean equals(Object FloatObj)</code>	Возвращает логическое значение <code>true</code> , если вызывающий объект типа <code>Float</code> равен объекту <code>FloatObj</code> ; а иначе — логическое значение <code>false</code>
<code>static int floatToIntBits(float число)</code>	Возвращает совместимую со стандартом IEEE комбинацию двоичных разрядов одинарной точности, соответствующую заданному числу

Метод	Описание
static int floatToRawIntBits(float <i>число</i>)	Возвращает совместимую со стандартом IEEE комбинацию двоичных разрядов одинарной точности, соответствующую заданному <i>числу</i> . Значение NaN не допускается
float floatValue()	Возвращает значение вызывающего объекта как тип float
int hashCode()	Возвращает хеш-код вызывающего объекта
static int hashCode(float <i>число</i>)	Возвращает хеш-код заданного <i>числа</i> (внедрен в версии JDK 8)
static float intBitsToFloat(int <i>число</i>)	Возвращает эквивалент типа float совместимой со стандартом IEEE комбинации двоичных разрядов одинарной точности, определяемой параметром <i>число</i>
int intValue()	Возвращает значение вызывающего объекта как тип int
boolean isInfinite()	Возвращает логическое значение true , если вызывающий объект содержит бесконечное значение, а иначе — логическое значение false
static boolean isInfinite(float <i>число</i>)	Возвращает логическое значение true , если <i>число</i> определяет бесконечное значение, а иначе — логическое значение false
boolean isNaN()	Возвращает логическое значение true , если вызывающий объект содержит нечисловое значение, а иначе — логическое значение false
static boolean isNaN(float <i>число</i>)	Возвращает логическое значение true , если <i>число</i> определяет нечисловое значение, а иначе — логическое значение false
long longValue()	Возвращает значение вызывающего объекта как тип long
static float max(float <i>val</i>, float <i>val2</i>)	Возвращает наибольшее из двух значений <i>val</i> и <i>val2</i> (внедрен в версии JDK 8)
static float min(float <i>val</i>, float <i>val2</i>)	Возвращает наименьшее из двух значений <i>val</i> и <i>val2</i> (внедрен в версии JDK 8)
static float parseFloat(String <i>строка</i>) throws NumberFormatException	Возвращает эквивалент числа типа float , содержащегося в <i>строке</i> , по основанию 10
short shortValue()	Возвращает значение вызывающего объекта как тип short
static float sum(float <i>val</i>, float <i>val2</i>)	Возвращает результат сложения значений <i>val</i> и <i>val2</i> (внедрен в версии JDK 8)
static String toHexString(float <i>число</i>)	Возвращает символьную строку, содержащую <i>число</i> в шестнадцатеричном формате

Окончание табл. 17.1

Метод	Описание
<code>String toString()</code>	Возвращает строковый эквивалент вызывающего объекта
<code>static String toString(float число)</code>	Возвращает строковый эквивалент значения, определяемого параметром <i>число</i>
<code>static Float valueOf(float число)</code>	Возвращает объект типа Float , содержащий значение, передаваемое в качестве параметра <i>число</i>
<code>static Float valueOf(String строка) throws NumberFormatException</code>	Возвращает объект типа Float , содержащий значение, указанное в <i>строке</i>

Таблица 17.2. Методы из класса Double

Метод	Описание
<code>byte byteValue()</code>	Возвращает значение вызывающего объекта как byte
<code>static int compare(double число1, double число2)</code>	Сравнивает значения <i>число1</i> и <i>число2</i> . Возвращает нулевое значение, если сравниваемые числовые значения равны; отрицательное значение, если <i>число1</i> меньше, чем <i>число2</i> ; или положительное значение, если <i>число1</i> больше, чем <i>число2</i>
<code>int compareTo (Double d)</code>	Сравнивает числовое значение вызывающего объекта со значением <i>d</i> . Возвращает нулевое значение, если сравниваемые числовые значения равны; отрицательное значение, если вызывающий объект имеет меньшее значение; или положительное значение, если вызывающий объект имеет большее значение
<code>static long doubleToLongBits (double число)</code>	Возвращает совместимую со стандартом IEEE комбинацию двоичных разрядов двойной точности, соответствующую заданному <i>числу</i>
<code>static long doubleToRawLongBits (double число)</code>	Возвращает совместимую со стандартом IEEE комбинацию двоичных разрядов двойной точности, соответствующую заданному <i>числу</i> . Значение NaN не допускается
<code>double doubleValue()</code>	Возвращает значение вызывающего объекта как тип double
<code>boolean equals (Object DoubleObj)</code>	Возвращает логическое значение true , если вызывающий объект типа Double равен объекту <i>DoubleObj</i> , а иначе — логическое значение false
<code>float floatValue()</code>	Возвращает значение вызывающего объекта как тип float
<code>int hashCode()</code>	Возвращает хеш-код вызывающего объекта
<code>static int hashCode (double число)</code>	Возвращает хеш-код заданного <i>числа</i> (внедрен в версии JDK 8)
<code>int intValue()</code>	Возвращает значение вызывающего объекта как тип int
<code>boolean isInfinite()</code>	Возвращает логическое значение true , если вызывающий объект содержит бесконечное значение, а иначе — логическое значение false

Метод	Описание
static boolean isInfinite(double число)	Возвращает логическое значение true , если <i>число</i> определяет бесконечное значение, а иначе — логическое значение false
boolean isNaN()	Возвращает логическое значение true , если вызывающий объект содержит нечисловое значение, а иначе — логическое значение false
static boolean isNaN(double число)	Возвращает логическое значение true , если <i>число</i> определяет нечисловое значение, а иначе — логическое значение false
static double longBitsToDouble(long число)	Возвращает эквивалент типа double совместимой со стандартом IEEE комбинации двоичных разрядов двойной точности, определяемой параметром <i>число</i>
long longValue()	Возвращает значение вызывающего объекта как тип long
static double max(double val, double val2)	Возвращает наибольшее из двух значений <i>val</i> и <i>val2</i> (внедрен в версии JDK 8)
static double min(double val, double val2)	Возвращает наименьшее из двух значений <i>val</i> и <i>val2</i> (внедрен в версии JDK 8)
static double parseDouble(String строка) throws NumberFormatException	Возвращает эквивалент числа типа double , содержащегося в <i>строке</i> , по основанию 10
short shortValue()	Возвращает значение вызывающего объекта как тип short
static double sum(double val, double val2)	Возвращает результат сложения значений <i>val+val2</i> (внедрен в версии JDK 8)
static String toHexString(double число)	Возвращает символьную строку, содержащую <i>число</i> в шестнадцатеричном формате
String toString()	Возвращает строковый эквивалент вызывающего объекта
static String toString(double число)	Возвращает строковый эквивалент значения, определяемого параметром <i>число</i>
static Double valueOf(double число)	Возвращает объект типа Double , содержащий значение, передаваемое в качестве параметра <i>число</i>
static Double valueOf(String строка) throws NumberFormatException	Возвращает объект типа Double , содержащий значение, указанное в <i>строке</i>

В приведенном ниже примере программы создаются два объекта типа **Double**: один — с помощью значения типа **double**, другой — с помощью символьной строки, которая может быть интерпретирована как тип **double**.

```
class DoubleDemo {
    public static void main(String args[]) {
        Double dl = new Double(3.14159);
```

```

        Double d2 = new Double("314159E-5");

        System.out.println(d1 + " = " + d2 + " -> " + d1.equals(d2));
    }
}

```

Как следует из приведенного ниже результата выполнения данной программы, оба конструктора создают одинаковые экземпляры класса `Double`. Об этом свидетельствует также вызов метода `equals()`, возвращающего логическое значение `true`.

```
3.14159 = 3.14159 -> true
```

Методы `isInfinite()` и `isNaN()`

В классах `Float` и `Double` предоставляются методы `isInfinite()` и `isNaN()`, помогающие манипулировать двумя специальными значениями типа `double` и `float`. Эти методы выполняют проверку на равенство двум однозначным значениям, определенным по стандарту IEEE для чисел с плавающей точкой: бесконечности и NaN (не число). Метод `isInfinite()` возвращает логическое значение `true`, если проверяемое число бесконечно велико или бесконечно мало по величине. А метод `isNaN()` возвращает логическое значение `true`, если проверяемое значение является нечисловым.

В следующем примере программы создаются два объекта типа `Double`: один из них содержит бесконечное, другой — нечисловое значение:

```

// Продемонстрировать применение методов isInfinite() и isNaN()
class InfNaN {
    public static void main(String args[]) {
        Double d1 = new Double(1/0.);
        Double d2 = new Double(0/0.);

        System.out.println(d1 + ": " + d1.isInfinite() + ", " +
                           d1.isNaN());
        System.out.println(d2 + ": " + d2.isInfinite() + ", " +
                           d2.isNaN());
    }
}

```

Эта программа выводит следующий результат:

```

Infinity: true, false
NaN: false, true

```

Классы `Byte`, `Short`, `Integer` и `Long`

Классы `Byte`, `Short`, `Integer` и `Long` служат оболочками для целочисленных типов `byte`, `short`, `int` и `long` соответственно. Ниже приведены их конструкторы.

Byte(byte *число*)

Byte(String *строка*) throws `NumberFormatException`

Short(short *число*)

Short(String *строка*) throws `NumberFormatException`

Integer(int *число*)

Integer(String *строка*) throws `NumberFormatException`

Long(long *число*)

Long(String *строка*) throws **NumberFormatException**

Как видите, объекты этих классов могут быть созданы из числовых значений или символьных строк, содержащих допустимые представления числовых значений.

Методы, определенные в этих классах, перечислены в табл. 17.3–17.6. Как видите, в них определяются методы для синтаксического анализа целых чисел из символьных строк и преобразования символьных строк обратно в целые числа. Варианты этих методов позволяют указывать *основание* системы счисления для преобразования чисел. Чаще всего применяется основание 2 для двоичных чисел, 8 — для восьмеричных, 10 — для десятичных и 16 — для шестнадцатеричных чисел.

В этих классах определены следующие константы:

MIN_VALUE	Минимальное значение
MAX_VALUE	Максимальное значение
SIZE	Длина заключенного в оболочку значения в битах
TYPE	Объект типа Class для типов byte , short , int или long

Таблица 17.3. Методы из класса Byte

Метод	Описание
byte byteValue()	Возвращает значение вызывающего объекта как тип byte
static int compare(byte <i>число1</i> , byte <i>число2</i>)	Сравнивает значения <i>число1</i> и <i>число2</i> . Возвращает нулевое значение, если сравниваемые числовые значения равны; отрицательное значение, если <i>число1</i> меньше, чем <i>число2</i> ; положительное значение, если <i>число1</i> больше, чем <i>число2</i>
int compareTo(Byte <i>b</i>)	Сравнивает числовое значение вызывающего объекта со значением <i>b</i> . Возвращает нулевое значение, если сравниваемые числовые значения равны; отрицательное значение, если вызывающий объект имеет меньшее значение; или положительное значение, если вызывающий объект имеет большее значение
static Byte decode(String <i>строка</i>) throws NumberFormatException	Возвращает объект типа Byte , содержащий значение, указанное в <i>строке</i>
double doubleValue()	Возвращает значение вызывающего объекта как тип double
boolean equals(Object <i>ByteObj</i>)	Возвращает логическое значение true , если вызывающий объект типа Double равен объекту <i>ByteObj</i> , а иначе — логическое значение false
float floatValue()	Возвращает значение вызывающего объекта как значение типа float
int hashCode()	Возвращает хеш-код вызывающего объекта
static int hashCode(byte <i>число</i>)	Возвращает хеш-код заданного <i>числа</i> (внедрен в версии JDK 8)

Окончание табл. 17.3

Метод	Описание
<code>int intValue()</code>	Возвращает значение вызывающего объекта как тип <code>int</code>
<code>long longValue()</code>	Возвращает значение вызывающего объекта как тип <code>long</code>
<code>static byte parseByte(String строка) throws NumberFormatException</code>	Возвращает эквивалент числа типа <code>byte</code> , содержащегося в <i>строке</i> , по основанию 10
<code>static byte parseByte(String строка, int основание) throws NumberFormatException</code>	Возвращает эквивалент числа типа <code>byte</code> , содержащегося в <i>строке</i> , по указанному <i>основанию</i> системы счисления
<code>short shortValue()</code>	Возвращает значение вызывающего объекта как <code>short</code>
<code>String toString()</code>	Возвращает символьную строку, содержащую десятичный эквивалент вызывающего объекта
<code>static String toString(byte число)</code>	Возвращает символьную строку, содержащую десятичный эквивалент <i>числа</i>
<code>static int toUnsignedInt(byte val)</code>	Возвращает значение <i>val</i> в виде целого значения без знака (внедрен в версии JDK 8)
<code>static long toUnsignedLong(byte val)</code>	Возвращает значение <i>val</i> в виде длинного целого значения без знака (внедрен в версии JDK 8)
<code>static Byte valueOf(byte число)</code>	Возвращает объект типа <code>Byte</code> , содержащий значение, передаваемое в качестве параметра <i>число</i>
<code>static Byte valueOf(String строка) throws NumberFormatException</code>	Возвращает объект типа <code>Byte</code> , содержащий значение, указанное в <i>строке</i>
<code>static Byte valueOf(String строка, int основание) throws NumberFormatException</code>	Возвращает объект типа <code>Byte</code> , содержащий значение, указанное в <i>строке</i> с учетом <i>основания</i> системы счисления

Таблица 17.4. Методы из класса Short

Метод	Описание
<code>byte byteValue()</code>	Возвращает значение вызывающего объекта как тип <code>byte</code>
<code>static int compare(short число1, short число2)</code>	Сравнивает значения <i>число1</i> и <i>число2</i> . Возвращает нулевое значение, если сравниваемые числовые значения равны; отрицательное значение, если <i>число1</i> меньше, чем <i>число2</i> ; положительное значение, если <i>число1</i> больше, чем <i>число2</i>
<code>int compareTo(Short s)</code>	Сравнивает числовое значение вызывающего объекта со значением <i>s</i> . Возвращает нулевое значение, если сравниваемые числовые значения равны; отрицательное значение, если вызывающий объект имеет меньшее значение; или положительное значение, если вызывающий объект имеет большее значение

Метод	Описание
static Short decode(String строка) throws NumberFormatException	Возвращает объект типа Short , содержащий значение, указанное в <i>строке</i>
double doubleValue()	Возвращает значение вызывающего объекта как тип double
boolean equals(Object ShortObj)	Возвращает логическое значение true , если вызывающий объект типа Short равен объекту <i>ShortObj</i> , а иначе — логическое значение false
float floatValue()	Возвращает значение вызывающего объекта как тип float
int hashCode()	Возвращает хеш-код вызывающего объекта
static int hashCode(short число)	Возвращает хеш-код заданного <i>числа</i> (внедрен в версии JDK 8)
int intValue()	Возвращает значение вызывающего объекта как тип int
long longValue()	Возвращает значение вызывающего объекта как тип long
static short parseShort(String строка) throws NumberFormatException	Возвращает эквивалент числа типа short , содержащегося в <i>строке</i> , по основанию 10
static short parseShort(String строка, int основание) throws NumberFormatException	Возвращает эквивалент числа типа short , содержащегося в <i>строке</i> , по указанному <i>основанию</i> системы счисления
static short reverseBytes(short число)	Меняет местами старший и младший байты заданного <i>числа</i> и возвращает результат
short shortValue()	Возвращает значение вызывающего объекта как тип short
String toString()	Возвращает символьную строку, содержащую десятичный эквивалент вызывающего объекта
static String toString(short число)	Возвращает символьную строку, содержащую десятичный эквивалент <i>числа</i>
static int toUnsignedInt(short val)	Возвращает значение <i>val</i> в виде целого значения без знака (внедрен в версии JDK 8)
static long toUnsignedLong(short val)	Возвращает значение <i>val</i> в виде длинного целого значения без знака (внедрен в версии JDK 8)
static Short valueOf(short число)	Возвращает объект типа Short , содержащий значение, передаваемое в качестве параметра <i>число</i>
static Short valueOf(String строка) throws NumberFormatException	Возвращает объект типа Short , содержащий значение, указанное в <i>строке</i>
static Short valueOf(String строка, int основание) throws NumberFormatException	Возвращает объект типа Short , содержащий значение, указанное в <i>строке</i> с учетом <i>основания</i> системы счисления

Таблица 17.5. Методы из класса Integer

Метод	Описание
<code>static int bitCount(int число)</code>	Возвращает количество битов в заданном <i>числе</i>
<code>byte byteValue()</code>	Возвращает значение вызывающего объекта как тип byte
<code>static int compare(int число1, int число2)</code>	Сравнивает значения <i>число1</i> и <i>число2</i> . Возвращает нулевое значение, если сравниваемые числовые значения равны; отрицательное значение, если <i>число1</i> меньше, чем <i>число2</i> ; положительное значение, если <i>число1</i> больше, чем <i>число2</i>
<code>int compareTo(Integer i)</code>	Сравнивает числовое значение вызывающего объекта со значением <i>i</i> . Возвращает нулевое значение, если сравниваемые числовые значения равны; отрицательное значение, если вызывающий объект имеет меньшее значение; или положительное значение, если вызывающий объект имеет большее значение
<code>static int compareUnsigned(int число1, int число2)</code>	Сравнивает значения <i>число1</i> и <i>число2</i> без учета знака. Возвращает нулевое значение, если сравниваемые числовые значения равны; отрицательное значение, если <i>число1</i> меньше, чем <i>число2</i> ; положительное значение, если <i>число1</i> больше, чем <i>число2</i> (внедрен в версии JDK 8)
<code>static Integer decode(String строка) throws NumberFormatException</code>	Возвращает объект типа Integer , содержащий значение, указанное в <i>строке</i>
<code>double doubleValue()</code>	Возвращает значение вызывающего объекта как тип double
<code>static int divideUnsigned(int делимое, int делитель)</code>	Возвращает результат деления <i>делимого</i> на <i>делитель</i> без знака (внедрен в версии JDK 8)
<code>boolean equals(Object IntegerObj)</code>	Возвращает логическое значение true , если вызывающий объект типа Integer равен объекту <i>IntegerObj</i> , а иначе — логическое значение false
<code>float floatValue()</code>	Возвращает значение вызывающего объекта как тип float
<code>static Integer getInteger(String имя_свойства)</code>	Возвращает значение, связанное со свойством окружения, определяемым параметром <i>имя_свойства</i> . При неудачном исходе возвращается пустое значение null
<code>static Integer getInteger(String имя_свойства, int по_умолчанию)</code>	Возвращает значение, связанное со свойством окружения, определяемым параметром <i>имя_свойства</i> . При неудачном исходе возвращается значение, заданное <i>по_умолчанию</i>
<code>static Integer getInteger(String имя_свойства, Integer по_умолчанию)</code>	Возвращает значение, связанное со свойством окружения, определяемым параметром <i>имя_свойства</i> . При неудачном исходе возвращается значение, заданное <i>по_умолчанию</i>

Метод	Описание
<code>int hashCode()</code>	Возвращает хеш-код вызывающего объекта
<code>static int hashCode(int число)</code>	Возвращает хеш-код заданного <i>числа</i> (внедрен в версии JDK 8)
<code>static int highestOneBit(int число)</code>	Определяет позицию самого старшего бита в заданном <i>числе</i> . Возвращает значение, в котором установлен только этот бит. Если ни один из битов не установлен, возвращается нулевое значение
<code>int intValue()</code>	Возвращает значение вызывающего объекта как тип <code>int</code>
<code>long longValue()</code>	Возвращает значение вызывающего объекта как тип <code>long</code>
<code>static int lowestOneBit(int число)</code>	Определяет позицию самого младшего бита в заданном <i>числе</i> . Возвращает значение, в котором установлен только этот бит. Если ни один из битов не установлен, возвращается нулевое значение
<code>static int max(int val, int val2)</code>	Возвращает наибольшее из двух значений <i>val</i> и <i>val2</i> (внедрен в версии JDK 8)
<code>static int min(int val, int val2)</code>	Возвращает наименьшее из двух значений <i>val</i> и <i>val2</i> (внедрен в версии JDK 8)
<code>static int numberOfLeadingZeros(int число)</code>	Возвращает количество старших битов, установленных в нуль и предшествующих первому старшему биту, установленному в заданном <i>числе</i> . Если <i>число</i> равно нулю, возвращается числовое значение 32
<code>static int numberOfTrailingZeros(int число)</code>	Возвращает количество младших битов, установленных в нуль и предшествующих первому младшему биту, установленному в заданном <i>числе</i> . Если <i>число</i> равно нулю, возвращается числовое значение 32
<code>static int parseInt(String строка) throws NumberFormatException</code>	Возвращает целочисленный эквивалент числа, содержащегося в <i>строке</i> , по основанию 10
<code>static int parseInt(String строка, int основание) throws NumberFormatException</code>	Возвращает целочисленный эквивалент числа, содержащегося в <i>строке</i> , по указанному <i>основанию</i> системы счисления
<code>static int parseUnsignedInt(String строка) throws NumberFormatException</code>	Возвращает целочисленный эквивалент числа без знака, содержащегося в <i>строке</i> , по основанию 10 (внедрен в версии JDK 8)
<code>static int parseUnsignedInt(String строка, int основание) throws NumberFormatException</code>	Возвращает целочисленный эквивалент числа без знака, содержащегося в <i>строке</i> , по указанному <i>основанию</i> системы счисления (внедрен в версии JDK 8)
<code>static int remainderUnsigned(int делимое, int делитель)</code>	Возвращает остаток от деления <i>делимого</i> на <i>делитель</i> без знака (внедрен в версии JDK 8)

Окончание табл. 17.5

Метод	Описание
<code>static int reverse(int число)</code>	Изменяет на противоположный порядок следования битов в заданном <i>числе</i> и возвращает результат
<code>static int reverseBytes(int число)</code>	Изменяет на противоположный порядок следования байтов в заданном <i>числе</i> и возвращает результат
<code>static int rotateLeft(int число, int n)</code>	Возвращает результат смещения заданного <i>числа</i> на <i>n</i> позиций влево
<code>static int rotateRight(int число, int n)</code>	Возвращает результат смещения заданного <i>числа</i> на <i>n</i> позиций вправо
<code>short shortValue()</code>	Возвращает значение вызывающего объекта как тип short
<code>static int signum(int число)</code>	Возвращает значение -1, если заданное <i>число</i> отрицательное; нулевое значение, если <i>число</i> равно нулю; и значение 1, если <i>число</i> положительное
<code>static int sum(int val, int val2)</code>	Возвращает результат сложения значений <i>val+val2</i> (внедрен в версии JDK 8)
<code>static String toBinaryString(int число)</code>	Возвращает символьную строку, содержащую двоичный эквивалент заданного <i>числа</i>
<code>static String toHexString(int число)</code>	Возвращает строку, содержащую шестнадцатеричный эквивалент заданного <i>числа</i>
<code>static String toOctalString(int число)</code>	Возвращает строку, содержащую восьмеричный эквивалент заданного <i>числа</i>
<code>String toString()</code>	Возвращает символьную строку, содержащую десятичный эквивалент вызывающего объекта
<code>static String toString(int число)</code>	Возвращает символьную строку, содержащую десятичный эквивалент заданного <i>числа</i>
<code>static String toString(int число, int основание)</code>	Возвращает символьную строку, содержащую десятичный эквивалент заданного <i>числа</i> с учетом указанного <i>основания</i>
<code>static long toUnsignedLong(int val)</code>	Возвращает значение <i>val</i> в виде длинного целого значения без знака (внедрен в версии JDK 8)
<code>static String toUnsignedString(int val)</code>	Возвращает символьную строку, содержащую десятичное целочисленное значение <i>val</i> без знака (внедрен в версии JDK 8)
<code>static String toUnsignedString(int val, int основание)</code>	Возвращает символьную строку, содержащую целочисленное значение <i>val</i> без знака с учетом <i>основания</i> системы счисления (внедрен в версии JDK 8)
<code>static Integer valueOf(int число)</code>	Возвращает объект типа Integer , содержащий значение, передаваемое в качестве параметра <i>число</i>
<code>static Integer valueOf(String строка) throws NumberFormatException</code>	Возвращает объект типа Integer , содержащий значение, указанное в <i>строке</i>
<code>static Integer valueOf(String строка, int основание) throws NumberFormatException</code>	Возвращает объект типа Integer , содержащий значение, указанное в <i>строке</i> с учетом <i>основания</i> системы счисления

Таблица 17.6. Методы из класса Long

Метод	Описание
<code>static int bitCount(long <i>число</i>)</code> <code>byte byteValue()</code>	Возвращает количество битов в заданном <i>числе</i> Возвращает значение вызывающего объекта как тип byte
<code>static int compare(long <i>число1</i>, long <i>число2</i>)</code>	Сравнивает значения <i>число1</i> и <i>число2</i> . Возвращает нулевое значение, если сравниваемые числовые значения равны; отрицательное значение, если <i>число2</i> меньше, чем <i>число1</i> ; положительное значение, если <i>число1</i> больше, чем <i>число2</i>
<code>int compareTo(Long <i>l</i>)</code>	Сравнивает числовое значение вызывающего объекта со значением <i>l</i> . Возвращает нулевое значение, если сравниваемые числовые значения равны; отрицательное значение, если вызывающий объект имеет меньшее значение; или положительное значение, если вызывающий объект имеет большее значение
<code>static int compareUnsigned(long <i>число1</i>, long <i>число2</i>)</code>	Сравнивает значения <i>число1</i> и <i>число2</i> без учета знака. Возвращает нулевое значение, если сравниваемые числовые значения равны; отрицательное значение, если <i>число1</i> меньше, чем <i>число2</i> ; положительное значение, если <i>число1</i> больше, чем <i>число2</i> (внедрен в версии JDK 8)
<code>static Long decode(String <i>строка</i>)</code> <code>throws NumberFormatException</code>	Возвращает объект типа Long , содержащий значение, указанное в <i>строке</i>
<code>static long divideUnsigned(long <i>делимое</i>, long <i>делитель</i>)</code> <code>double doubleValue()</code>	Возвращает результат деления <i>делимое</i> на <i>делитель</i> без знака (внедрен в версии JDK 8) Возвращает значение вызывающего объекта как тип double
<code>boolean equals(Object <i>LongObj</i>)</code>	Возвращает логическое значение true , если вызывающий объект типа Long равен объекту <i>LongObj</i> , а иначе — логическое значение false
<code>float floatValue()</code>	Возвращает значение вызывающего объекта как тип float
<code>static Long getLong(String <i>имя_свойства</i>)</code>	Возвращает значение, связанное со свойством окружения, определяемым параметром <i>имя_свойства</i> . При неудачном исходе возвращается пустое значение null
<code>static Long getLong(String <i>имя_свойства</i>, long <i>по_умолчанию</i>)</code>	Возвращает значение, связанное со свойством окружения, определяемым параметром <i>имя_свойства</i> . При неудачном исходе возвращается значение, заданное <i>по_умолчанию</i>
<code>static Long getLong(String <i>имя_свойства</i>, Long <i>по_умолчанию</i>)</code>	Возвращает значение, связанное со свойством окружения, определяемым параметром <i>имя_свойства</i> . При неудачном исходе возвращается значение, заданное <i>по_умолчанию</i>

Продолжение табл. 17.6

Метод	Описание
<code>int hashCode()</code>	Возвращает хеш-код вызывающего объекта
<code>static int hashCode(long число)</code>	Возвращает хеш-код заданного <i>числа</i> (внедрен в версии JDK 8)
<code>static int highestOneBit(long число)</code>	Определяет позицию самого старшего бита в заданном <i>числе</i> . Возвращает значение, в котором установлен только этот бит. Если ни один из битов не установлен, возвращается нулевое значение
<code>int intValue()</code>	Возвращает значение вызывающего объекта как тип <code>int</code>
<code>long longValue()</code>	Возвращает значение вызывающего объекта как тип <code>long</code>
<code>static int lowestOneBit(long число)</code>	Определяет позицию самого младшего бита в заданном <i>числе</i> . Возвращает значение, в котором установлен только этот бит. Если ни один из битов не установлен, возвращается нулевое значение
<code>static long max(long val, long val2)</code>	Возвращает наибольшее из двух значений <i>val</i> и <i>val2</i> (внедрен в версии JDK 8)
<code>static long min(long val, long val2)</code>	Возвращает наименьшее из двух значений <i>val</i> и <i>val2</i> (внедрен в версии JDK 8)
<code>static int numberOfLeadingZeros(long число)</code>	Возвращает количество старших битов, установленных в нуль и предшествующих первому старшему биту, установленному в заданном <i>числе</i> . Если <i>число</i> равно нулю, возвращается числовое значение 64
<code>static int numberOfTrailingZeros(long число)</code>	Возвращает количество младших битов, установленных в нуль и предшествующих первому младшему биту, установленному в заданном <i>числе</i> . Если <i>число</i> равно нулю, возвращается числовое значение 64
<code>static long parseLong(String строка) throws NumberFormatException</code>	Возвращает эквивалент типа <code>long</code> числа, содержащегося в <i>строке</i> , по основанию 10
<code>static long parseInt(String строка, int основание) throws NumberFormatException</code>	Возвращает эквивалент типа <code>long</code> числа, содержащегося в <i>строке</i> , по указанному <i>основанию</i> системы счисления
<code>static long parseUnsignedInt(String строка) throws NumberFormatException</code>	Возвращает целочисленный эквивалент числа без знака, содержащегося в <i>строке</i> , по основанию 10 (внедрен в версии JDK 8)
<code>static long parseUnsignedInt(String строка, int основание) throws NumberFormatException</code>	Возвращает целочисленный эквивалент числа без знака, содержащегося в <i>строке</i> , по указанному <i>основанию</i> системы счисления (внедрен в версии JDK 8)

Метод	Описание
<code>static long remainderUnsigned(int делимое, int делитель)</code>	Возвращает остаток от деления указанного <i>делимого</i> на <i>делитель</i> без знака (внедрен в версии JDK 8)
<code>static long reverse(long число)</code>	Изменяет на противоположный порядок следования битов в заданном <i>числе</i> и возвращает результат
<code>static long reverseBytes(long число)</code>	Изменяет на противоположный порядок следования байтов в заданном <i>числе</i> и возвращает результат
<code>static long rotateLeft(long число, int n)</code>	Возвращает результат смещения заданного <i>числа</i> на <i>n</i> позиций влево
<code>static long rotateRight(long число, int n)</code>	Возвращает результат смещения заданного <i>числа</i> на <i>n</i> позиций вправо
<code>short shortValue()</code>	Возвращает значение вызывающего объекта как тип <code>short</code>
<code>static int signum(int число)</code>	Возвращает значение <code>-1</code> , если заданное <i>число</i> отрицательное; нулевое значение, если <i>число</i> равно нулю; и значение <code>1</code> , если <i>число</i> положительное
<code>static int sum(long val, int val2)</code>	Возвращает результат сложения значений <i>val+val2</i> (внедрен в версии JDK 8)
<code>static String toBinaryString(long число)</code>	Возвращает символьную строку, содержащую двоичный эквивалент заданного <i>числа</i>
<code>static String toHexString(long число)</code>	Возвращает строку, содержащую шестнадцатеричный эквивалент заданного <i>числа</i>
<code>static String toOctalString(long число)</code>	Возвращает строку, содержащую восьмеричный эквивалент заданного <i>числа</i>
<code>String toString()</code>	Возвращает символьную строку, содержащую десятичный эквивалент вызывающего объекта
<code>static String toString(long число)</code>	Возвращает символьную строку, содержащую десятичный эквивалент заданного <i>числа</i>
<code>static String toString(long число, int основание)</code>	Возвращает символьную строку, содержащую десятичный эквивалент заданного <i>числа</i> с учетом указанного <i>основания</i> системы счисления
<code>static String toUnsignedString(long val)</code>	Возвращает символьную строку, содержащую десятичное целочисленное значение <i>val</i> без знака (внедрен в версии JDK 8)
<code>static String toUnsignedString(long val, int основание)</code>	Возвращает символьную строку, содержащую целочисленное значение <i>val</i> без знака с учетом указанного <i>основания</i> системы счисления (внедрен в версии JDK 8)
<code>static Long valueOf(long число)</code>	Возвращает объект типа <code>Long</code> , содержащий значение, передаваемое в качестве параметра <i>число</i>

Окончание табл. 17.6

Метод	Описание
static Long valueOf(String строка) throws NumberFormatException	Возвращает объект типа Long , содержащий значение, указанное в <i>строке</i>
static Long valueOf(String строка, int основание) throws NumberFormatException	Возвращает объект типа Long , содержащий значение, указанное в <i>строке</i> с учетом указанного <i>основания</i> системы счисления

Взаимное преобразование чисел и символьных строк

Одной из наиболее часто выполняемых рутинных операций в программировании является преобразование строкового представления чисел во внутренний двоичный формат. Правда, сделать это в Java совсем не трудно. В классах `Byte`, `Short`, `Integer` и `Long` для этой цели предоставляются методы `parseByte()`, `parseShort()`, `parseInt()` и `parseLong()` соответственно. Эти методы возвращают значения типа `byte`, `short`, `int` или `long`, эквивалентные числовой строке, с которой они были вызваны (аналогичные методы предусмотрены в классах `Float` и `Double`).

В приведенном ниже примере программы демонстрируется применение метода `parseInt()`. В этой программе суммируется ряд целочисленных значений, вводимых пользователем. С этой целью целочисленные значения считываются методом `readLine()` в виде числовых строк, которые затем преобразуются методом `parseInt()` в эквивалентные им числовые значения типа `int`.

/* Эта программа суммирует ряд целых чисел, вводимых пользователем. Она преобразует строковое представление каждого числа в целое значение методом `parseInt()`

```
*/
import java.io.*;
class ParseDemo {
    public static void main(String args[])
        throws IOException
    {
        // создать буферизированный поток чтения типа BufferedReader,
        // используя стандартный поток ввода System.in
        BufferedReader br =
            new BufferedReader(new InputStreamReader(System.in));
        String str;
        int i;
        int sum=0;
        System.out.println("Введите число, 0 - для выхода.");
        do {
            str = br.readLine();
            try {
                i = Integer.parseInt(str);
            } catch (NumberFormatException e) {
                System.out.println("Неверный формат");
                i = 0;
            }
            sum += i;
            System.out.println("Текущая сумма: " + sum);
        } while(i != 0);
    }
}
```


Для преобразования целого числа в десятичную строку служат варианты метода `toString()`, определенные в классе `Byte`, `Short`, `Integer` или `Long`. В классах `Integer` и `Long` предоставляются также методы `toBinaryString()`, `toHexString()` и `toOctalString()`, преобразующие числовое значение в двоичную, шестнадцатеричную и восьмеричную строки соответственно.

В следующем примере программы демонстрируется преобразование целого числа в двоичную, шестнадцатеричную и восьмеричную строковую форму:

```
/* Преобразовать целое число в двоичную, шестнадцатеричную
   и восьмеричную строковую форму
*/
class StringConversions {
    public static void main(String args[]) {
        int num = 19648;
        System.out.println(Число num + " в двоичной форме: " +
                           Integer.toBinaryString(num));
        System.out.println(Число num + " в восьмеричной форме: " +
                           Integer.toOctalString(num));
        System.out.println(Число num + " в шестнадцатеричной форме: " +
                           Integer.toHexString(num));
    }
}
```

Ниже приведен результат, выводимый данной программой.

```
Число 19648 в двоичной форме: 100110011000000
Число 19648 в восьмеричной форме: 46300
Число 19648 в шестнадцатеричной форме: 4cc0
```

Класс `Character`

Класс `Character` служит простой оболочкой для типа `char`. Конструктор этого класса выглядит следующим образом:

```
Character(char символ)
```

где параметр *символ* обозначает тот символ, который заключается в оболочку создаваемого объекта типа `Character`. Чтобы получить значение типа `char`, содержащееся в объекте типа `Character`, достаточно вызвать метод `charValue()`, как показано ниже. Этот метод возвратит символ.

```
char charValue()
```

В классе `Character` определен ряд констант, включая следующие.

BYTES	Длина типа char в байтах (внедрена в версии JDK 8)
MAX_RADIX	Максимальное основание системы счисления
MIN_RADIX	Минимальное основание системы счисления
MAX_VALUE	Максимальное значение
MIN_VALUE	Минимальное значение
TYPE	Объект типа Class для типа char

В состав класса `Character` входит ряд статических методов, распределяющих символы на категории и изменяющих их регистр. Они перечислены далее в табл. 17.7. В следующем примере программы демонстрируется применение некоторых из этих методов:

```
// Продемонстрировать применение некоторых методов типа Is
class IsDemo {
    public static void main(String args[]) {
        char a[] = {'a', 'b', '5', '?', 'A', ' '};
        for(int i=0; i<a.length; i++) {
            if(Character.isDigit(a[i]))
                System.out.println(a[i] + " - цифра.");
            if(Character.isLetter(a[i]))
                System.out.println(a[i] + " - буква.");
            if(Character.isWhitespace(a[i]))
                System.out.println(a[i] + " - пробельный символ.");
            if(Character.isUpperCase(a[i]))
                System.out.println(a[i] +
                                   " - прописная буква.");
            if(Character.isLowerCase(a[i]))
                System.out.println(a[i] + " - строчная буква.");
        }
    }
}
```

Эта программа выводит следующий результат:

```
a - буква.
a - строчная буква.
b - буква.
b - строчная буква.
5 - цифра.
A - буква.
A - прописная буква.
- пробельный символ.
```

В классе `Character` определены еще два метода, `forDigit()` и `digit()`, предназначенные для взаимного преобразования целочисленных значений и цифр, которые их представляют. Ниже приведены их общие формы.

```
static char forDigit(int число, int основание)
static int digit(char цифра, int основание)
```

Метод `forDigit()` возвращает цифровое значения, передаваемого в качестве параметра *число*. Основание системы счисления для преобразования числа определяется параметром *основание*. А метод `digit()` возвращает целочисленное значение, связанное с заданным символом (предположительно цифрой) и с учетом указанного основания системы счисления. (Имеется еще одна форма метода `digit()` для получения кодовой точки в Юникоде. Более подробно кодовые точки обсуждаются в следующем разделе.)

В классе `Character` определен также метод `compareTo()`, имеющий следующую общую форму:

```
int compareTo(Character c)
```

Этот метод возвращает нулевое значение, если вызывающий объект и символ имеют одинаковое значение; отрицательное значение, если значение вызывающего объекта меньше; а иначе — положительное значение.

Класс `Character` содержит метод `getDirectionality()`, позволяющий определить направленность символа. Для описания направленности символов в этот класс добавлено несколько констант, хотя в большинстве программ определять направленность символов не требуется. Кроме того, в классе `Character` переопределяются методы `equals()` и `hashCode()`.

Таблица 17.7. Различные методы из класса Character

Метод	Описание
<code>static boolean isDefined(char <i>символ</i>)</code>	Возвращает логическое значение true , если <i>символ</i> определен в Юникоде, а иначе — логическое значение false
<code>static boolean isDigit(char <i>символ</i>)</code>	Возвращает логическое значение true , если <i>символ</i> является цифрой, а иначе — логическое значение false
<code>static boolean isIdentifierIgnorable(char <i>символ</i>)</code>	Возвращает логическое значение true , если <i>символ</i> должен быть проигнорирован в идентификаторе, а иначе — логическое значение false
<code>static boolean isISOControl(char <i>символ</i>)</code>	Возвращает логическое значение true , если <i>символ</i> является управляющим символом по стандарту ISO, а иначе — логическое значение false
<code>static boolean isJavaIdentifierPart(char <i>символ</i>)</code>	Возвращает логическое значение true , если <i>символ</i> может быть частью идентификатора Java, а иначе — логическое значение false
<code>static boolean isJavaIdentifierStart(char <i>символ</i>)</code>	Возвращает логическое значение true , если <i>символ</i> может быть первым символом идентификатора Java, а иначе — логическое значение false
<code>static boolean isLetter(char <i>символ</i>)</code>	Возвращает логическое значение true , если <i>символ</i> является буквой, а иначе — логическое значение false
<code>static boolean isLetterOrDigit(char <i>символ</i>)</code>	Возвращает логическое значение true , если <i>символ</i> является буквой или цифрой, а иначе — логическое значение false
<code>static boolean isLowerCase(char <i>символ</i>)</code>	Возвращает логическое значение true , если <i>символ</i> является строчной буквой, а иначе — логическое значение false
<code>static boolean isMirrored(char <i>символ</i>)</code>	Возвращает логическое значение true , если <i>символ</i> является зеркально отображаемым символом в Юникоде, а иначе — логическое значение false . <i>Зеркально отображаемым</i> называется символ, предназначенный для текстов, отображаемых справа налево
<code>static boolean isSpaceChar(char <i>символ</i>)</code>	Возвращает логическое значение true , если <i>символ</i> является пробельным символом в Юникоде, а иначе — логическое значение false
<code>static boolean isTitleCase(char <i>символ</i>)</code>	Возвращает логическое значение true , если <i>символ</i> является прописной буквой, а иначе — логическое значение false
<code>static boolean isUnicodeIdentifierPart(char <i>символ</i>)</code>	Возвращает логическое значение true , если <i>символ</i> может быть частью идентификатора в Юникоде (кроме первого символа), а иначе — логическое значение false

Окончание табл. 17.7

Метод	Описание
<code>static boolean isUnicodeIdentifierStart(char символ)</code>	Возвращает логическое значение true , если <i>символ</i> может быть первым символом идентификатора в Юникоде, а иначе — логическое значение false
<code>static boolean isUpperCase(char символ)</code>	Возвращает логическое значение true , если <i>символ</i> является прописной буквой, а иначе — логическое значение false
<code>static boolean isWhitespace(char символ)</code>	Возвращает логическое значение true , если <i>символ</i> является пробельным символом, а иначе — логическое значение false
<code>static char toLowerCase(char символ)</code>	Возвращает эквивалентную <i>символу</i> строчную букву
<code>static char toTitleCase(char символ)</code>	Возвращает эквивалентную <i>символу</i> прописную букву
<code>static char toUpperCase(char символ)</code>	Возвращает эквивалентную <i>символу</i> прописную букву

Для обращения с символами имеются еще два класса. В частности, класс `Character.Subset` служит для описания подмножества символов в Юникоде, а класс `Character.UnicodeBlock` содержит блоки символов в Юникоде.

Дополнения класса Character для поддержки кодовых точек в Юникоде

За последнее время в класс `Character` были внесены существенные дополнения. Начиная с версии JDK 5 класс `Character` обеспечивает поддержку 32-разрядных символов в Юникоде. В прошлом все символы в Юникоде составляли 16 двоичных разрядов, что равно длине типа `char` (и значения, заключаемого в оболочку класса `Character`), поскольку коды этих символов находятся в пределах от 0 до **FFFF**. Но набор символов в Юникоде был расширен, для чего потребовалось еще 16 двоичных разрядов. Теперь коды символов находятся в пределах от 0 до **10FFFF**.

В связи с этим появились три важных термина. *Кодовая точка* — это символ в пределах от 0 до **10FFFF**. А символы, имеющие код свыше **FFFF**, называются *дополнительными*. Основную многоязыковую плоскость составляют символы в пределах от 0 до **FFFF**.

В связи с расширением набора символов в Юникоде возникло серьезное затруднение для программирования на Java. Значение дополнительных символов оказывается больше, чем умещается в типе `char`, поэтому для их поддержки требуются дополнительные средства. В Java это затруднение разрешается двумя способами. Во-первых, для представления дополнительных символов в Java используется суррогатная пара типа `char`. Первая половина этой пары называется *старшим суррогатом*, а вторая — *младшим суррогатом*. Для взаимного преобразования кодовых точек и до-

полнительных символов предусмотрены новые методы вроде `codePointAt()`. И во-вторых, в Java перегружаются некоторые методы, существовавшие ранее в классе `Character`. В перегружаемых формах этих методов используются данные типа `int` вместо `char`. А поскольку тип `int` достаточно велик, чтобы вместить любой символ как одно значение, то его можно использовать для хранения любого символа. Например, у всех методов из табл. 17.7 имеются перегружаемые формы, оперирующие данными типа `int`. Ниже приведены примеры общих форм некоторых из этих методов.

```
static boolean isDigit(int кодовая_точка)
static boolean isLetter(int кодовая_точка)
static int toLowerCase(int кодовая_точка)
```

Помимо методов, перегружаемых для обращения с кодовыми точками, в класс `Character` введены методы, предоставляющие дополнительную поддержку кодовых точек. Избранные методы этой категории приведены в табл. 17.8.

Таблица 17.8. Избранные методы из класса `Character` для поддержки 32-разрядных кодовых точек в Юникоде

Метод	Описание
<code>static int charCount(int кодовая_точка)</code>	Возвращает значение 1, если <i>кодовая_точка</i> может быть представлена одной переменной типа <code>char</code> . А если требуются две такие переменные, то возвращается значение 2
<code>static int codePointAt(CharSequence символы, int позиция)</code>	Возвращает кодовую точку, находящуюся на заданной <i>позиции</i>
<code>static int codePointAt(char символы[], int позиция)</code>	Возвращает кодовую точку, находящуюся на заданной <i>позиции</i>
<code>static int codePointBefore(CharSequence символы, int позиция)</code>	Возвращает кодовую точку, находящуюся на позиции, предшествующей заданной <i>позиции</i>
<code>static int codePointBefore(char символы[], int позиция)</code>	Возвращает кодовую точку, находящуюся на позиции, предшествующей заданной <i>позиции</i>
<code>static boolean isBmpCodePoint(int кодовая_точка)</code>	Возвращает логическое значение <code>true</code> , если <i>кодовая_точка</i> относится к основной многоязыковой плоскости, а иначе — логическое значение <code>false</code>
<code>static boolean isHighSurrogate(char символ)</code>	Возвращает логическое значение <code>true</code> , если <i>символ</i> содержит достоверный символ старшего суррогата
<code>static boolean isLowSurrogate(char символ)</code>	Возвращает логическое значение <code>true</code> , если <i>символ</i> содержит достоверный символ младшего суррогата
<code>static boolean isSupplementaryCodePoint(int символ)</code>	Возвращает логическое значение <code>true</code> , если <i>символ</i> содержит дополнительный символ
<code>static boolean isSurrogatePair(char старший_суррогат, char младший_суррогат)</code>	Возвращает логическое значение <code>true</code> , если <i>старший_суррогат</i> и <i>младший_суррогат</i> образуют достоверную суррогатную пару

Окончание табл. 17.8

Метод	Описание
static boolean isValidCodePoint(int <i>кодovая_точка</i>)	Возвращает логическое значение true , если <i>кодovая_точка</i> содержит достоверную кодovую точку
static char[] toChars(int <i>кодovая_точка</i>)	Преобразует <i>кодovую_точку</i> в ее эквивалент типа char , для чего могут потребоваться две переменные типа char . Возвращает массив, содержащий результат
static int toChars(int <i>кодovая_точка</i>, char <i>адресат</i>[], int <i>позиция</i>)	Преобразует <i>кодovую_точку</i> в ее эквивалент типа char , сохраняя результат в массиве <i>адресат</i> , начиная с заданной <i>позиции</i> . Возвращает значение 1 , если <i>кодovая_точка</i> может быть представлена одной переменной типа char , а иначе — значение 2
static int toCodePoint(char <i>старший_суррогат</i>, char <i>младший_суррогат</i>)	Преобразует <i>старший_суррогат</i> и <i>младший_суррогат</i> в эквивалентную кодovую точку

Класс Boolean

Класс Boolean служит очень тонкой оболочкой для логических значений типа **boolean**, что удобно в тех случаях, когда логические значения требуется передавать по ссылке. Этот класс содержит константы **TRUE** и **FALSE**, определяющие объекты типа **Boolean**, которые соответствуют истинному и ложному значениям. В классе **Boolean** определяется также поле **TYPE**, являющееся объектом типа **Class** для типа **boolean**. В классе **Boolean** определены следующие конструкторы: **Boolean(boolean *логическое_значение*)**, **Boolean(String *логическая_строка*)**

В первом конструкторе параметр *логическое_значение* должен быть равным **true** или **false**. А во втором конструкторе новый объект класса **Boolean** будет содержать логическое значение **true**, если параметр *логическая_строка* содержит символьную строку "true" (в верхнем или нижнем регистре). В противном случае новый объект будет содержать логическое значение **false**.

В классе **Boolean** определены методы, перечисленные в табл. 17.9.

Таблица 17.9. Методы из класса **Boolean**

Метод	Описание
boolean booleanValue()	Возвращает эквивалент типа boolean
static int compare(boolean <i>b1</i>, boolean <i>b2</i>)	Возвращает нулевое значение, если значения <i>b1</i> и <i>b2</i> одинаковы; положительное значение, если значение <i>b1</i> равно true , а значение <i>b2</i> равно false ; и отрицательное значение противном случае
int compareTo(Boolean <i>b</i>)	Возвращает нулевое значение, если значения вызывающего объекта и <i>b</i> одинаковы; положительное значение, если значение вызывающего объекта равно true , а значение <i>b</i> равно false ; и отрицательное значение противном случае

Метод	Описание
<code>boolean equals(Object <i>BoolObj</i>)</code>	Возвращает логическое значение true , если вызывающий объект равнозначен объекту <i>BoolObj</i> , а иначе — логическое значение false
<code>static boolean getBoolean(String <i>имя_свойства</i>)</code>	Возвращает логическое значение true , если системное свойство, определяемое параметром <i>имя_свойства</i> , равно true , а иначе — логическое значение false
<code>int hashCode()</code>	Возвращает хеш-код вызывающего объекта
<code>static int hashCode(boolean <i>логическое_значение</i>)</code>	Возвращает хеш-код указанного <i>логического_значения</i> (добавлен в версии JDK 8)
<code>static boolean logicalAnd(boolean <i>op1</i>, boolean <i>op2</i>)</code>	Выполняет логическую операцию И над операндами <i>op1</i> и <i>op2</i> и возвращает результат (добавлен в версии JDK 8)
<code>static boolean logicalOR(boolean <i>op1</i>, boolean <i>op2</i>)</code>	Выполняет логическую операцию ИЛИ над операндами <i>op1</i> и <i>op2</i> и возвращает результат (добавлен в версии JDK 8)
<code>static boolean logicalXOR(boolean <i>op1</i>, boolean <i>op2</i>)</code>	Выполняет логическую операцию исключающее ИЛИ над операндами <i>op1</i> и <i>op2</i> и возвращает результат (добавлен в версии JDK 8)
<code>static boolean parseBoolean(String <i>строка</i>)</code>	Возвращает логическое значение true , если <i>строка</i> содержит символьную строку " true " без учета регистра, а иначе — логическое значение false
<code>String toString()</code>	Возвращает строковый эквивалент вызывающего объекта
<code>String toString(boolean <i>логическое_значение</i>)</code>	Возвращает строковый эквивалент указанного <i>логического_значения</i>
<code>static boolean valueOf(boolean <i>логическое_значение</i>)</code>	Возвращает логический эквивалент указанного <i>логического_значения</i>
<code>static boolean valueOf(String <i>логическая_строка</i>)</code>	Возвращает логическое значение true , если указанная <i>логическая_строка</i> содержит символьную строку " true " без учета регистра, а иначе — логическое значение false

Класс Void

Этот класс содержит единственное поле `TYPE`, в котором хранится ссылка на объект типа `Class` для типа `void`. Экземпляры этого класса не создаются.

Класс Process

Абстрактный класс `Process` инкапсулирует *процесс*, т.е. выполняющуюся программу. Он используется в основном в качестве суперкласса для типа объектов,

создаваемых методом `exec()` из класса `Runtime` или методом `start()` из класса `ProcessBuilder`. Класс `Process` содержит абстрактные методы, перечисленные в табл. 17.10.

Таблица 17.10. Методы из класса `Process`

Метод	Описание
<code>void destroy()</code>	Прерывает процесс
<code>Process destroyForcibly()</code>	Принудительно завершает вызывающий процесс. Возвращает ссылку на процесс (добавлен в версии JDK 8)
<code>int exitValue()</code>	Возвращает код завершения процесса
<code>InputStream getErrorStream()</code>	Возвращает поток ввода для чтения данных из потока вывода ошибок <code>err</code> вызывающего процесса
<code>InputStream getOutputStream()</code>	Возвращает поток ввода для чтения данных из потока вывода <code>out</code> вызывающего процесса
<code>OutputStream getOutputStream()</code>	Возвращает поток вывода для записи данных в поток ввода <code>in</code> вызывающего процесса
<code>boolean isAlive()</code>	Возвращает логическое значение <code>true</code> , если вызывающий процесс по-прежнему действует, а иначе — логическое значение <code>false</code> (добавлен в версии JDK 8)
<code>Int waitfor() throws InterruptedException</code>	Возвращает код завершения процесса. Не возвращает управление до тех пор, пока процесс, для которого он вызван, не завершится
<code>Int waitfor(long время_ожидания, TimeUnit единица_времени) throws InterruptedException</code>	Ожидает завершения вызывающего процесса. Период ожидания определяется параметром <i>время_ожидания</i> в единицах времени, обозначаемых параметром <i>единица_времени</i> . Возвращает логическое значение <code>true</code> , если процесс завершился, а по истечении заданного <i>времени_ожидания</i> — логическое значение <code>false</code> (добавлен в версии JDK 8)

Класс `Runtime`

Этот класс инкапсулирует исполняющую среду. Создать объект типа `Runtime` нельзя, но можно получить ссылку на текущий объект типа `Runtime`, вызвав статический метод `Runtime.getRuntime()`. Получив ссылку на текущий объект типа `Runtime`, можно вызвать несколько методов, управляющих состоянием и поведением виртуальной машины JVM. Апплеты и другой не заслуживающий доверия код не могут вызывать методы из класса `Runtime`, не генерируя исключение типа `SecurityException`. Наиболее употребительные методы из класса `Runtime` перечислены в табл. 17.11.

Таблица 17.11. Избранные методы из класса **Runtime**

Метод	Описание
<code>void addShutdownHook (Thread <i>поток</i>)</code>	Регистрирует <i>поток</i> , который должен быть запущен на исполнение при остановке виртуальной машины JVM
<code>Process exec (String <i>имя_программы</i>) throws IOException</code>	Выполняет программу, определяемую параметром <i>имя_программы</i> , как отдельный процесс. Возвращает объект типа Process , описывающий новый процесс
<code>Process exec (String <i>имя_программы</i>, String <i>окружение</i>[]) throws IOException</code>	Выполняет программу, определяемую параметром <i>имя_программы</i> , как отдельный процесс в окружении, обозначаемом параметром <i>окружение</i> . Возвращает объект типа Process , описывающий новый процесс
<code>Process exec (String <i>массив_командной_строки</i>[]) throws IOException</code>	Выполняет командную строку, передаваемую в качестве параметра <i>массив_командной_строки</i> , как отдельный процесс. Возвращает объект типа Process , описывающий новый процесс
<code>Process exec (String <i>массив_командной_строки</i>[], String <i>окружение</i>[]) throws IOException</code>	Выполняет командную строку, передаваемую в качестве параметра <i>массив_командной_строки</i> , как отдельный процесс в окружении, обозначаемом параметром <i>окружение</i> . Возвращает объект типа Process , описывающий новый процесс
<code>void exit (int <i>код_завершения</i>)</code>	Прерывает выполнение и возвращает <i>код_завершения</i> родительскому процессу. Условно нулевой код означает нормальное завершение, а все другие коды завершения обозначают различные виды ошибок
<code>long freeMemory ()</code>	Возвращает приблизительное количество байтов свободной памяти, доступной исполняющей системе Java
<code>void gc ()</code>	Иницирует сборку “мусора”
<code>static Runtime getRuntime ()</code>	Возвращает текущий объект типа Runtime
<code>void halt (int <i>код</i>)</code>	Немедленно прерывает работу виртуальной машины JVM. Никакие потоки или методы завершения не выполняются. Вызывающему процессу возвращается заданный <i>код</i>
<code>void load (String <i>имя_файла_библиотеки</i>)</code>	Загружает динамическую библиотеку, файл которой обозначается параметром <i>имя_файла_библиотеки</i> , включая и полный путь к нему
<code>void loadLibrary (String <i>имя_библиотеки</i>)</code>	Загружает динамическую библиотеку, имя которой связывается с параметром <i>имя_библиотеки</i>
<code>boolean removeShutdownHook (Thread <i>поток</i>)</code>	Удаляет <i>поток</i> из списка потоков, запускаемых на исполнение при останове виртуальной машины JVM. Возвращает логическое значение true при удачном исходе, т.е. в том случае, если поток удален

Окончание табл. 17.11

Метод	Описание
<code>void runFinalization()</code>	Иницииирует вызовы метода finalize() для неиспользованных, но еще не возвращенных объектов
<code>long totalMemory()</code>	Возвращает общее количество байтов оперативной памяти, доступной программе
<code>void traceInstructions(boolean Включить_трассировку)</code>	Включает или отключает трассировку инструкций в зависимости от значения параметра Включить_трассировку . Если этот параметр принимает логическое значение true , то трассировка включается, а если он принимает значение false – трассировка отключается
<code>void traceMethodCalls(boolean Включить_трассировку)</code>	Включает и отключает трассировку вызовов методов в зависимости от значения параметра Включить_трассировку . Если этот параметр принимает логическое значение true , то трассировка включается, а если он принимает значение false – трассировка отключается

Рассмотрим два наиболее распространенных примера применения класса `Runtime`: управление памятью и выполнение дополнительных процессов.

Управление памятью

Несмотря на то что в Java организуется автоматическая сборка “мусора”, иногда требуется знать, какая часть выделяемой оперативной памяти занята объектами и какая ее часть еще свободна. Эти сведения можно, например, использовать, чтобы проверить эффективность прикладного кода или выяснить, сколько еще объектов определенного типа может быть инициализировано. Для получения этих сведений служат методы `totalMemory()` и `freeMemory()`.

Как упоминалось в части I, система сборки “мусора” в Java запускается периодически для утилизации неиспользуемых объектов. Но иногда может возникнуть потребность собрать отвергнутые объекты до того, как система сборки “мусора” будет запущена в очередной раз. Ее можно запускать по требованию, вызывая метод `gc()`. Можно также попробовать вызвать сначала метод `gc()`, а после него – метод `freeMemory()`, чтобы получить основные сведения об использовании памяти. Выполняя далее прикладной код, можно снова вызвать метод `freeMemory()`, чтобы выяснить, сколько памяти еще свободно. Такой подход к управлению памятью демонстрируется в следующем примере программы:

```
// Продемонстрировать применение методов totalMemory(),  
// freeMemory() и gc()  
class MemoryDemo {  
    public static void main(String args[]) {  
        Runtime r = Runtime.getRuntime();  
        long mem1, mem2;  
        Integer someints[] = new Integer[1000];  
  
        System.out.println("Всего памяти: " + r.totalMemory());  
        mem1 = r.freeMemory();
```

```

System.out.println("Свободной памяти исходно: " + mem1);
r.gc();
mem1 = r.freeMemory();
System.out.println("Свободной памяти после очистки: " +
    mem1);

for(int i=0; i<1000; i++)
    someints[i] = new Integer(i); // выделить память для
                                // объектов типа Integer
mem2 = r.freeMemory();
System.out.println("Свободной памяти после выделения: " +
    mem2);
System.out.println("Использовано памяти для выделения: " +
    (mem1-mem2));

// отбросить Integers
for(int i=0; i<1000; i++) someints[i] = null;

r.gc(); // запустить сборку "мусора"

mem2 = r.freeMemory();
System.out.println("Свободной памяти после очистки " +
    "отвергнутых объектов типа Integer: " + mem2);
}
}

```

Ниже приведен примерный результат, выводимый данной программой (у вас он может оказаться иным в зависимости от конкретной исполняющей среды).

```

Всего памяти: 1048568
Свободной памяти исходно: 751392
Свободной памяти после очистки: 841424
Свободной памяти после выделения: 824000
Использовано памяти для выделения: 17424
Свободной памяти после очистки отброшенных объектов типа Integer: 842640

```

Выполнение других программ

В безопасных средах рассматриваемые здесь языковые средства Java можно использовать для выполнения других тяжеловесных процессов (т.е. программ) в многозадачной операционной системе. Некоторые формы метода `exec()` позволяют указывать программу, которую требуется выполнить, а также передать ей входные параметры. Метод `exec()` возвращает объект типа `Process`, который затем может быть использован для управления взаимодействием прикладной программы на Java с этим вновь запущенным процессом. Но поскольку языковые средства Java могут функционировать на разных платформах и в среде различных операционных систем, то метод `exec()` сильно зависит от конкретной среды.

В приведенном ниже примере метод `exec()` используется для запуска приложения Notepad — простого текстового редактора в Windows. Очевидно, что код этого примера должен выполняться в среде операционной системы Windows.

```

// Продемонстрировать применение метода exec()
class ExecDemo {
    public static void main(String args[]) {
        Runtime r = Runtime.getRuntime();

```

```

Process p = null;
try {
    p = r.exec("Notepad");
} catch (Exception e) {
    System.out.println("Ошибка запуска Notepad.");
}
}
}

```

Существует несколько альтернативных форм метода `exec()`, но форма, показанная в данном примере, используется чаще всего. Объектом типа `Process`, возвращаемым методом `exec()`, можно манипулировать, используя другие методы из класса `Process` после запуска программы на выполнение. Так, вызвав метод `destroy()`, можно удалить процесс, а с помощью метода `waitFor()` — заставить прикладную программу ожидать завершения процесса. Метод `exitValue()` возвратит значение, которое возвращается процессом по его завершении. Обычно это нулевое значение, если не возникает никаких осложнений. Ниже приведен предыдущий пример, демонстрирующий применение метода `exec()`, но видоизмененный таким образом, чтобы ожидать завершения запущенного процесса.

```

// Ожидать завершения работы текстового редактора Notepad
class ExecDemoFini {
    public static void main(String args[]) {
        Runtime r = Runtime.getRuntime();
        Process p = null;

        try {
            p = r.exec("Notepad");
            p.waitFor();
        } catch (Exception e) {
            System.out.println("Ошибка запуска Motepad.");
        }

        System.out.println("Notepad возвратил " + p.exitValue());
    }
}

```

Во время выполнения процесса можно выполнять операции стандартного ввода и вывода. Методы `getOutputStream()` и `getInputStream()` возвращают дескрипторы стандартных потоков ввода `in` и вывода `out` для данного процесса. (Операции ввода-вывода подробно рассматриваются в главе 20.)

Класс `ProcessBuilder`

Класс `ProcessBuilder` обеспечивает другой способ запуска процессов (т.е. программ) и управления ими. Как пояснялось ранее, все процессы представлены классом `Process`, и каждый процесс может быть запущен методом `Runtime.exec()`. А в классе `ProcessBuilder` предоставляются более развитые средства управления процессами, с помощью которых можно, например, установить текущий рабочий каталог и изменить параметры окружения. В классе `ProcessBuilder` определены следующие конструкторы:

```

ProcessBuilder(List<String> аргументы)
ProcessBuilder(String ... аргументы)

```

где параметр *аргументы* обозначает список аргументов, указывающих имя программы, которую требуется запустить на выполнение со всеми необходимыми аргументами командной строки. В первом конструкторе аргументы передаются в списке типа `List`, а во втором конструкторе они указываются в качестве параметра переменной длины. В табл. 17.12 перечислены методы, определенные в классе `ProcessBuilder`.

Таблица 17.12. Методы из класса `ProcessBuilder`

Метод	Описание
<code>List<String> command()</code>	Возвращает ссылку на список типа <code>List</code> , который содержит имя программы и ее аргументы. Изменения в этом списке относятся к вызывающему объекту
<code>ProcessBuilder command(List<String> аргументы)</code>	Задает имя программы и ее аргументы в качестве параметра <i>аргументы</i> . Изменения в этом списке относятся к вызывающему объекту. Возвращает ссылку на вызывающий объект
<code>ProcessBuilder command(String ... аргументы)</code>	Задает имя программы и ее аргументы в качестве параметра <i>аргументы</i> . Возвращает ссылку на вызывающий объект
<code>File directory()</code>	Возвращает текущий рабочий каталог для вызывающего объекта. Если каталог тот же самый, что и у программы на Java, которая запустила процесс, то возвращается пустое значение <code>null</code>
<code>ProcessBuilder directory(File каталог)</code>	Устанавливает текущий каталог для вызывающего объекта. Возвращает ссылку на вызывающий объект
<code>Map<String, String> environment()</code>	Возвращает переменные окружения, связанные с вызывающим объектом, в виде пар “ключ–значение”
<code>ProcessBuilder inheritIO()</code>	Вынуждает вызываемый процесс использовать те же самые источник и адресат для стандартных потоков ввода-вывода, что и вызывающий процесс
<code>ProcessBuilder.Redirect redirectError()</code>	Возвращает адресат для стандартного потока вывода ошибок в виде объекта типа <code>ProcessBuilder.Redirect</code>
<code>ProcessBuilder redirectError(File f)</code>	Задает адресат для стандартного потока вывода ошибок в указанный файл. Возвращает ссылку на вызывающий объект
<code>ProcessBuilder redirectError(ProcessBuilder.Redirect адресат)</code>	Задает <i>адресат</i> для стандартного потока вывода ошибок. Возвращает ссылку на вызывающий объект
<code>boolean redirectErrorStream()</code>	Возвращает логическое значение <code>true</code> , если стандартный поток вывода ошибок направляется в стандартный поток вывода данных. А если эти потоки вывода используются раздельно, то возвращается логическое значение <code>false</code>

Окончание табл. 17.12

Метод	Описание
ProcessBuilder redirectErrorStream (boolean <i>слияние</i>)	Если параметр <i>слияние</i> принимает логическое значение true , то стандартный поток вывода ошибок направляется в стандартный поток вывода данных. А если параметр <i>слияние</i> принимает логическое значение false , то эти потоки вывода используются раздельно, что делается по умолчанию. Возвращает ссылку на вызывающий объект
ProcessBuilder.Redirect redirectInput ()	Возвращает источник для стандартного потока ввода в виде объекта типа ProcessBuilder.Redirect
ProcessBuilder redirectInput (File <i>f</i>)	Задаёт источник для стандартного потока ввода в указанный файл. Возвращает ссылку на вызывающий объект
ProcessBuilder redirectInput (ProcessBuilder.Redirect <i>источник</i>)	Задаёт <i>источник</i> для стандартного потока ввода. Возвращает ссылку на вызывающий объект
ProcessBuilder.Redirect redirectOutput ()	Возвращает адресат для стандартного потока вывода в виде объекта типа ProcessBuilder.Redirect
ProcessBuilder redirectOutput (File <i>f</i>)	Задаёт адресат для стандартного потока вывода в указанный файл. Возвращает ссылку на вызывающий объект
ProcessBuilder redirectOutput (ProcessBuilder.Redirect <i>адресат</i>)	Устанавливает <i>адресат</i> для стандартного потока вывода. Возвращает ссылку на вызывающий объект
Process start () throws IOException	Запускает процесс, указываемый вызывающим объектом. Иными словами, запускает заданную программу

В табл. 17.12 обратите внимание на те методы, в которых используется класс **ProcessBuilder.Redirect**. Этот абстрактный класс инкапсулирует источник или адресат ввода-вывода, связанный с процессом. Кроме того, эти методы позволяют переадресовать источник или адресат операций ввода-вывода. Например, вызвав метод **to** (), можно переадресовать вывод в файл, вызвав метод **from** () — переадресовать ввод из файла, а вызвав метод **appendTo** () — присоединить вывод к файлу. Объект типа **File**, связанный с файлом, может быть получен при вызове метода **file** (). Ниже приведены общие формы этих методов.

```
static ProcessBuilder.Redirect to(File f)
static ProcessBuilder.Redirect from(File f)
static ProcessBuilder.Redirect appendTo(File f)
File file()
```

В классе **ProcessBuilder.Redirect** поддерживается также метод **type** (), возвращающий значение перечислимого типа **ProcessBuilder.Redirect.Type**. В этом перечислении описываются виды переадресации ввода-вывода, определяемые константами **APPEND**, **INHERIT**, **PIPE**, **READ** и **WRITE**. В классе **ProcessBuilder.Redirect** также определены константы **INHERIT** и **PIPE**.

Чтобы создать процесс, используя класс `ProcessBuilder`, достаточно создать экземпляр этого класса, указав имя программы и все необходимые аргументы, а чтобы начать выполнение программы — вызвать метод `start()` для этого экземпляра. В приведенном ниже примере демонстрируется запуск текстового редактора Notepad в Windows. Обратите внимание на то, что в качестве аргумента передается имя текстового файла, который требуется отредактировать.

```
class PBDemo {
    public static void main(String args[]) {
        try {
            ProcessBuilder proc =
                new ProcessBuilder("notepad.exe", "testfile");
            proc.start();
        } catch (Exception e) {
            System.out.println("Ошибка запуска Notepad.");
        }
    }
}
```

Класс System

Этот класс содержит коллекцию статических методов и переменных. Стандартные потоки ввода, вывода данных и ошибок в исполняющей системе Java хранятся в переменных `in`, `out` и `err` соответственно. Методы, определенные в классе `System`, перечислены в табл. 17.13. Многие из них генерируют исключение типа `SecurityException`, если операция не допускается диспетчером защиты.

Таблица 17.13. Методы из класса System

Метод	Описание
<code>static void arraycopy(Object источник, int начало_источника, Object адресат, int начало_адресата, int размер)</code>	Копирует массив. Копируемый массив передается в качестве параметра <i>источник</i> , а индекс позиции, с которой начинается копирование из <i>источника</i> , обозначается параметром <i>начало_источника</i> . Массив, принимающий копию, передается в качестве параметра <i>адресат</i> , а индекс позиции, с которой следует начинать копирование, обозначается параметром <i>начало_адресата</i> . И наконец, параметр <i>размер</i> задает количество копируемых элементов
<code>static String clearProperty(String которая)</code>	Удаляет переменную окружения, обозначаемую параметром <i>которая</i> . Возвращает предыдущее значение, связанное с переменной окружения, обозначаемой параметром <i>которая</i>
<code>static Console console()</code>	Возвращает консоль, связанную с виртуальной машиной JVM. В отсутствие текущей консоли у виртуальной машины JVM возвращается пустое значение <code>null</code>

Продолжение табл. 17.13

Метод	Описание
<code>static long currentTimeMillis()</code>	Возвращает текущее время в миллисекундах, прошедших с полуночи 1 января 1970 года
<code>static void exit(int код_завершения)</code>	Прерывает выполнение и возвращает <i>код_завершения</i> родительскому процессу (обычно операционной системе). Условно нулевой код означает нормальное завершение, а все другие коды завершения обозначают различные виды ошибок
<code>static void gc()</code>	Иницирует сборку “мусора”
<code>static Map<String, String> getenv()</code>	Возвращает объект типа Map , содержащий текущие переменные окружения и их значения
<code>static String getenv(String которая)</code>	Возвращает значение, связанное с переменной окружения, передаваемой в качестве параметра <i>которая</i>
<code>static Properties getProperties()</code>	Возвращает свойства, связанные с исполняющей системой Java (класс Properties описывается в главе 18)
<code>static String getProperty(String которая)</code>	Возвращает свойство, связанное с переменной окружения, обозначаемой параметром <i>которая</i> . Если нужное свойство не найдено, возвращается пустое значение null
<code>static String getProperty(String которая, String по_умолчанию)</code>	Возвращает свойство, связанное с переменной окружения, обозначаемой параметром <i>которая</i> . Если нужное свойство не найдено, возвращается пустое значение, задаваемое параметром <i>по_умолчанию</i>
<code>static SecurityManager getSecurityManager()</code>	Возвращает текущий диспетчер защиты или пустое значение null , если диспетчер защиты не установлен
<code>static int identityHashCode(Object объект)</code>	Возвращает хеш-код идентичности <i>объекта</i>
<code>static Channel inheritedChannel() throws IOException</code>	Возвращает канал, наследуемый виртуальной машиной JVM, или пустое значение null , если канал не наследуется
<code>static String lineSeparator()</code>	Возвращает строку, содержащую символы разделителей строк
<code>static void load(String имя_файла_библиотеки)</code>	Загружает динамическую библиотеку, файл которой задается параметром <i>имя_файла_библиотеки</i> , включая полный путь доступа к нему
<code>static void loadLibrary(String имя_библиотеки)</code>	Загружает динамическую библиотеку, имя которой связывается с параметром <i>имя_библиотеки</i>

Метод	Описание
<code>static void mapLibraryName(String библиотека)</code>	Возвращает зависящее от платформы имя указанной <i>библиотеки</i>
<code>static long nanoTime()</code>	Получает наиболее точный таймер системы и возвращает его значение в наносекундах, прошедших от некоторого произвольно выбранного момента времени. Точность таймера неизвестна
<code>static void runFinalization()</code>	Иницирует вызовы метода <code>finalize()</code> для неиспользуемых, но еще не утилизированных объектов
<code>static void setErr(PrintStream поток_вывода_ошибок)</code>	Задает <i>поток_вывода_ошибок</i> в качестве стандартного потока вывода ошибок <code>err</code>
<code>static void setIn(InputStream поток_ввода)</code>	Задает <i>поток_ввода</i> в качестве стандартного потока ввода <code>in</code>
<code>static void setOut(PrintStream поток_вывода)</code>	Задает <i>поток_вывода</i> в качестве стандартного потока вывода <code>out</code>
<code>static void setProperties(Properties системные_свойства)</code>	Устанавливает текущие системные свойства, определяемые параметром <i>системные_свойства</i>
<code>static String setProperty(String которое, String v)</code>	Присваивает заданное значение <i>v</i> свойству, определяемому параметром <i>которое</i>
<code>static void setSecurityManager(SecurityManager диспетчер_защиты)</code>	Устанавливает заданный <i>диспетчер_защиты</i>

Рассмотрим некоторые типичные примеры применения класса `System`.

Измерение времени выполнения программы методом `currentTimeMills()`

Одним из особенно любопытных примеров применения класса `System` является измерение времени выполнения различных частей прикладной программы с помощью метода `currentTimeMills()`, возвращающего текущее время в миллисекундах, прошедшее с полуночи 1 января 1970 года. Чтобы хронометрировать отдельную часть проверяемой программы, следует сохранить это значение непосредственно перед началом выполнения этой части и сразу же после ее выполнения вызвать метод `currentTimeMills()` еще раз. Время выполнения определяется как разность между конечным и начальным моментами времени. В следующем примере программы показано, каким образом такое измерение времени выполнения осуществляется на практике:

```
// Измерение времени выполнения программы
class Elapsed {
    public static void main(String args[]) {
        long start, end;

        System.out.println(
```

```

        "Измерение времени перебора от 0 до 100000000");
    // измерить время перебора от 0 до 100000000

    start = System.currentTimeMillis(); // получить начальный
                                     // момент времени
    for(long i=0; i < 100000000L; i++) ;
    end = System.currentTimeMillis(); // получить конечный
                                     // момент времени
    System.out.println("Время выполнения: " + (end-start));
}
}

```

Ниже приведен примерный результат, выводимый данной программой (у вас он может оказаться иным в зависимости от конкретной исполняющей среды).

```

Измерение времени перебора от 0 до 100000000
Время выполнения: 10

```

Если таймер вашей системы работает с точностью до наносекунд, можете переписать эту программу, воспользовавшись методом `nanoTime()` вместо метода `currentTimeMills()`. В качестве примера ниже показана главная часть этой программы, переписанная с целью воспользоваться методом `nanoTime()`.

```

start = System.nanoTime(); // получить начальный момент времени
for(long i=0; i < 1000000000L; i++) ;
end = System.nanoTime(); // получить конечный момент времени

```

Применение метода `arraycopy()`

С помощью метода `arraycopy()` можно быстро скопировать массив любого типа из одного места в другое. Это намного быстрее, чем выполнить эквивалентный цикл, написанный вручную на Java. В качестве примера ниже приведена программа, где два массива копируются методом `arraycopy()`. Сначала массив `a` копируется в массив `b`, а затем все элементы массива `a` сдвигаются на одну позицию в *начало* массива. После этого весь массив `b` смещается на одну позицию в *конец* массива.

```

// Использовать метод arraycopy()
class ACDemo {
    static byte a[] = { 65, 66, 67, 68, 69, 70, 71, 72, 73, 74 };
    static byte b[] = { 77, 77, 77, 77, 77, 77, 77, 77, 77, 77 };
    public static void main(String args[]) {
        System.out.println("a = " + new String(a));
        System.out.println("b = " + new String(b));
        System.arraycopy(a, 0, b, 0, a.length);
        System.out.println("a = " + new String(a));
        System.out.println("b = " + new String(b));
        System.arraycopy(a, 0, a, 1, a.length - 1);
        System.arraycopy(b, 1, b, 0, b.length - 1);
        System.out.println("a = " + new String(a));
        System.out.println("b = " + new String(b));
    }
}

```

Как следует из приведенного ниже результата выполнения данной программы, копировать можно один и тот же источник и получатель в обоих направлениях.

```
a = ABCDEFGHIJ
b = MMMMMMMMMM
a = ABCDEFGHIJ
b = ABCDEFGHIJ
a = AABCDEFGHI
b = BCDEFGHIJJ
```

Свойства окружения

В любом случае доступны следующие свойства окружения:

<code>file.separator</code>	<code>java.specification.version</code>	<code>java.vm.version</code>
<code>java.class.path</code>	<code>java.vendor</code>	<code>line.separator</code>
<code>java.class.version</code>	<code>java.vendor.url</code>	<code>os.arch</code>
<code>java.compiler</code>	<code>java.version</code>	<code>os.name</code>
<code>java.ext.dirs</code>	<code>java.vm.name</code>	<code>os.version</code>
<code>java.home</code>	<code>java.vm.specification.name</code>	<code>path.separator</code>
<code>java.io.tmpdir</code>	<code>java.vm.specification.vendor</code>	<code>user.dir</code>
<code>java.library.path</code>	<code>java.vm.specification.version</code>	<code>user.home</code>
<code>java.specification.name</code>	<code>java.vm.vendor</code>	<code>user.name</code>
<code>java.specification.vendor</code>		

Значения различных переменных окружения можно получить, вызвав метод `System.getProperty()`. Например, в следующей программе выводится путь к текущему пользовательскому каталогу:

```
class ShowUserDir {
    public static void main(String args[]) {
        System.out.println(System.getProperty("user.dir"));
    }
}
```

Класс Object

Как упоминалось в части I, класс `Object` служит суперклассом для всех остальных классов. В классе `Object` определены методы, перечисленные в табл. 17.14 и применяемые к любому объекту.

Таблица 17.14. Методы из класса `Object`

Метод	Описание
<code>Object clone() throws CloneNotSupportedException</code>	Создает новый объект, который оказывается таким же, как и вызывающий объект
<code>boolean equals(Object объект)</code>	Возвращает логическое значение <code>true</code> , если заданный <i>объект</i> равнозначен вызывающему объекту
<code>void finalize() throws Throwable</code>	Вызывается по умолчанию перед удалением неиспользуемого объекта

Окончание табл. 17.14

Метод	Описание
final Class <?> getClass()	Получает объект типа Class , который описывает вызывающий объект
int hashCode()	Возвращает хеш-код, связанный с вызывающим объектом
final void notify()	Прерывает исполнение потока, ожидающего вызывающий объект
final void notifyAll()	Прерывает исполнение всех потоков, ожидающих вызывающий объект
String toString()	Возвращает символьную строку, описывающую объект
final void wait() throws InterruptedException	Ожидает завершения другого потока исполнения
final void wait(long <i>миллисекунд</i>) throws InterruptedException	Ожидает завершения другого потока исполнения до истечения указанного количества <i>миллисекунд</i>
final void wait(long <i>миллисекунд</i>, int <i>наносекунд</i>) throws InterruptedException	Ожидает завершения другого потока исполнения до истечения указанного количества <i>миллисекунд</i> плюс <i>наносекунд</i>

Применение метода clone () и интерфейса Cloneable

Большинство методов, определенных в классе `Object`, описываются в других главах данной книги. Но один из них, метод `clone()`, требует особого рассмотрения. Этот метод создает дубликат того объекта, который его вызывает. Клонировать можно объекты только тех классов, которые реализуют интерфейс `Cloneable`.

В интерфейсе `Cloneable` никакие члены не объявляются. Он служит лишь для указания на то, что в реализующем его классе допускает поразрядное копирование (т.е. *клонирование*) объектов. Если попытаться вызвать метод `clone()` для объекта класса, не реализующего интерфейс `Cloneable`, будет сгенерировано исключение типа `CloneNotSupportedException`. При клонировании конструктор клонируемого объекта *не* вызывается. Получаемый в итоге клон является точной копией своего оригинала.

Клонирование считается потенциально опасным действием, поскольку оно может вызвать нежелательные побочные эффекты. Так, если клонируемый объект содержит ссылочную переменную `objRef`, то в клоне она будет ссылаться на тот же самый объект, что и в оригинале. Если клон вносит изменения в содержимое объекта, на который ссылается переменная `objRef`, то изменится и оригинал. Обратимся к другому примеру. Если объект открывает поток ввода-вывода, а затем клонируется, то оба объекта будут оперировать одним и тем же потоком ввода-вы-

вода. Более того, если один из этих объектов закроет поток ввода-вывода, то второй может по-прежнему пытаться выводить в него данные, что приведет к ошибке. Чтобы разрешить подобные затруднения, иногда требуется переопределить метод `clone()`, определенный в классе `Object`.

В связи с осложнениями, которыми чревато клонирование объектов, метод `clone()` объявляется в классе `Object` как `protected`. Это означает, что он может быть вызван из метода, определенного в классе, реализующем интерфейс `Cloneable`, или же должен быть явно переопределен в классе как `public`. Рассмотрим каждый из этих подходов на конкретных примерах.

В следующем примере программы реализуется интерфейс `Cloneable` и определяется метод `cloneTest()`, вызывающий метод `clone()` из класса `Object`:

```
// Продемонстрировать применение метода clone()
class TestClone implements Cloneable {
    int a;
    double b;
    // В этом методе вызывается метод clone() из класса Object
    TestClone cloneTest() {
        try {
            // вызвать метод clone() из класса Object
            return (TestClone) super.clone();
        } catch (CloneNotSupportedException e) {
            System.out.println("Клонирование невозможно.");
            return this;
        }
    }
}

class CloneDemo {
    public static void main(String args[]) {
        TestClone x1 = new TestClone();
        TestClone x2;
        x1.a = 10;
        x1.b = 20.98;
        x2 = x1.cloneTest(); // клонировать объект x1
        System.out.println("x1: " + x1.a + " " + x1.b);
        System.out.println("x2: " + x2.a + " " + x2.b);
    }
}
```

В данном примере метод `cloneTest()` вызывает метод `clone()` из класса `Object` и возвращает получаемый результат. Обратите внимание на то, что объект, возвращаемый методом `clone()`, должен быть приведен к соответствующему типу (в данном случае — `TestClone`).

В следующем примере программы метод `clone()` перегружается таким образом, чтобы вызываться за пределами своего класса. Для этого его следует объявить с модификатором доступа `public`, как показано ниже.

```
// Переопределить метод clone()
class TestClone implements Cloneable {
    int a;
    double b;
    // метод clone() переопределяется теперь как public
    public Object clone() {
        try {
            // вызвать метод clone() из класса Object
            return super.clone();
        }
    }
}
```

```

    } catch(CloneNotSupportedException e) {
        System.out.println("Клонирование невозможно.");
        return this;
    }
}

class CloneDemo2 {
    public static void main(String args[]) {
        TestClone x1 = new TestClone();
        TestClone x2;
        x1.a = 10;
        x1.b = 20.98;
        // здесь метод clone() вызывается непосредственно
        x2 = (TestClone) x1.clone();
        System.out.println("x1: " + x1.a + " " + x1.b);
        System.out.println("x2: " + x2.a + " " + x2.b);
    }
}

```

Побочные эффекты от клонирования иногда трудно обнаружить поначалу. Ведь очень легко решить, что класс безопасен для клонирования, когда на самом деле это не так. Вообще говоря, реализовывать интерфейс Cloneable в любом классе без серьезной на то причины не стоит.

Класс Class

Этот класс инкапсулирует состояние времени выполнения класса или интерфейса. Объекты типа Class создаются автоматически при загрузке класса, а объявлять объект класса Class явно нельзя. В общем, для получения объекта типа Class вызывается метод getClass(), определенный в классе Object. Класс Class представляет обобщенный тип, объявляемый следующим образом:

```
Class Class<T>
```

где T обозначает тип представляемого класса или интерфейса. Наиболее часто употребляемые методы, определенные в классе Class, приведены в табл. 17.15.

Таблица 17.15. Избранные методы из класса Class

Метод	Описание
static Class<?> forName(String имя) throws ClassNotFoundException	Возвращает объект типа Class по его полному имени
static Class<?> forName(String имя, boolean способ, ClassLoader загрузчик) throws ClassNotFoundException	Возвращает объект типа Class по его полному имени. Объект загружается с помощью указанного <i>загрузчика</i> . Если параметр <i>способ</i> принимает логическое значение true , то объект инициализируется, а иначе — не инициализируется
<A extends Annotation> A getAnnotation(Class<A> тип_аннотации)	Возвращает объект интерфейса Annotation , содержащий аннотацию, связанную с указанным <i>типом_аннотации</i> для вызывающего объекта

Метод	Описание
<code>Annotation[] getAnnotations()</code>	Получает аннотацию, связанную с вызывающим объектом, и сохраняет ее в массиве объектов интерфейса Annotation . Возвращает ссылку на массив
<code><A extends Annotation> A[] getAnnotationsByType(Class<A> тип_аннотации)</code>	Возвращает массив аннотаций, в том числе и повторяющихся аннотаций, имеющих указанный <i>тип_аннотации</i> , связанный с вызывающим объектом (добавлен в версии JDK 8)
<code>Class<?>[] getClasses()</code>	Возвращает объекты типа Class для каждого открытого класса и интерфейса, являющегося членом класса, представленного вызывающим объектом
<code>ClassLoader getClassLoader()</code>	Возвращает объект типа ClassLoader , который загрузил класс или интерфейс
<code>Constructor<T> getConstructor(Class ... типы_параметров) throws NoSuchMethodException, SecurityException</code>	Возвращает объект типа Constructor , обозначающий конструктор класса, предоставленного вызывающим объектом, имеющим указанные <i>типы_параметров</i>
<code>Constructor<?>[] getConstructors() throws SecurityException</code>	Получает объекты типа Constructor для каждого из открытых конструкторов класса, представленного вызывающим объектом, и сохраняет их в массиве. Возвращает ссылку на этот массив
<code>Annotation[] getDeclaredAnnotations()</code>	Получает объекты интерфейса Annotation для всех аннотаций, объявленных в вызывающем объекте, и сохраняет их в массиве. Возвращает ссылку на этот массив (унаследованные аннотации игнорируются)
<code><A extends Annotation> A[] getDeclaredAnnotationsByType(Class<A> тип_аннотации)</code>	Возвращает массив ненаследуемых аннотаций, в том числе и повторяющихся аннотаций, имеющих указанный <i>тип_аннотации</i> , связанный с вызывающим объектом (добавлен в версии JDK 8)
<code>Constructor<?>[] getDeclaredConstructors() throws SecurityException</code>	Получает объекты типа Constructor для каждого из объявленных конструкторов класса, представленных вызывающим объектом, и сохраняет их в массиве. Возвращает ссылку на этот массив (конструкторы суперклассов игнорируются)
<code>Field[] getDeclaredFields() throws SecurityException</code>	Получает объекты типа Field для каждого из полей, объявленных в классе или интерфейсе, представленном вызывающим объектом, и сохраняет их в массиве. Возвращает ссылку на массив. (Унаследованные поля игнорируются.)
<code>Method[] getDeclaredMethods() throws SecurityException</code>	Получает объекты типа Method для каждого из методов, объявленных в классе или интерфейсе, представленном вызывающим объектом, и сохраняет их в массиве. Возвращает ссылку на этот массив. (Унаследованные методы игнорируются.)

Окончание табл. 17.15

Метод	Описание
Field getField(String <i>имя_поля</i>) throws NoSuchMethodException, SecurityException	Возвращает объект типа Field , который представляет открытое поле, заданное в качестве параметра <i>имя_поля</i> для класса или интерфейса, представленного вызывающим объектом
Field[] getFields() throws SecurityException	Получает объекты типа Field для каждого открытого поля, класса или интерфейса, представленного вызывающим объектом, и сохраняет их в массиве. Возвращает ссылку на этот массив
Class<?>[] getInterfaces()	При вызове для объекта, представляющего класс или интерфейс, этот метод возвращает массив интерфейсов, реализуемых данным классом. Когда этот метод вызывается для объекта, представленного интерфейсом, он возвращает массив интерфейсов, расширяемых данным интерфейсом
Method getMethod(String <i>имя_метода</i> , Class<?> ... <i>типы_параметров</i>) throws NoSuchMethodException, SecurityException	Возвращает объект типа Method , который представляет открытый метод, обозначаемый как <i>имя_метода</i> и имеющий заданные <i>типы_параметров</i> в классе или интерфейсе, представленном вызывающим объектом
Method[] getMethods() throws SecurityException	Получает объекты типа Method для каждого открытого метода из класса или интерфейса, представленного вызывающим объектом, и сохраняет их в массиве. Возвращает ссылку на этот массив
String getName()	Возвращает полное имя класса или интерфейса типа, представленного вызывающим объектом
ProtectionDomain getProtectionDomain()	Возвращает домен защиты, связанный с вызывающим объектом
Class<? super T> getSuperclass()	Возвращает суперкласс типа, представленного вызывающим объектом. Возвращает пустое значение null , если представленный тип относится к классу Object или вообще не относится к классу
boolean isInterface()	Возвращает логическое значение true , если тип, представленный вызывающим объектом, является интерфейсом, а иначе — логическое значение false
T newInstance() throws IllegalAccessException, InstantiationException	Создает новый экземпляр (т.е. новый объект), имеющий такой же тип, как и представленный вызывающим объектом. Это равнозначно применению оператора new со стандартным конструктором класса. Возвращает новый объект. Этот метод завершится неудачно, если представленный тип окажется абстрактным, не относящимся к классу или не имеющим стандартный конструктор
String toString()	Возвращает строковое представление типа, представленного вызывающим объектом или интерфейсом

Методы, определенные в классе `Class`, часто применяются в тех случаях, когда требуются сведения о типе объекта во время выполнения. Как следует из табл. 17.15, в этом классе предоставляются методы, позволяющие получать дополнительные сведения об определенном классе, в том числе его открытых конструкторах, полях и методах. Это имеет значение и для обеспечения функциональных возможностей компонентов Java Beans, как поясняется далее.

В следующем примере программы демонстрируется применение метода `getClass()`, наследуемого из класса `Object`, а также метода `getSuperclass()`, наследуемого из класса `Class`.

```
// Продемонстрировать получение сведений о типе
// объекта во время выполнения
class X {
    int a;
    float b;
}

class Y extends X {
    double c;
}

class RTTI {
    public static void main(String args[]) {
        X x = new X();
        Y y = new Y();
        Class<?> clObj;
        clObj = x.getClass(); // получить ссылку на объект типа Class
        System.out.println("x - объект типа: " +
            clObj.getName());
        clObj = y.getClass(); // получить ссылку на объект типа Class
        System.out.println("y - объект типа: " +
            clObj.getName());
        clObj = clObj.getSuperclass();
        System.out.println("Суперкласс объекта y: " +
            clObj.getName());
    }
}
```

Ниже приведен результат, выводимый данной программой.

```
x - объект типа: X
y - объект типа: Y
Суперкласс объекта y: X
```

Класс `ClassLoader`

Абстрактный класс `ClassLoader` определяет порядок загрузки классов. В прикладной программе можно создавать подклассы, расширяющие класс `ClassLoader`, реализуя его методы. Это позволяет загружать классы не таким способом, которым выполняется обычная загрузка в исполняющей системе Java. Но делать этого обычно не нужно.

Класс Math

Этот класс обеспечивает все математические функции в формате с плавающей точкой, применяемые в геометрии и тригонометрии, а также некоторые методы общего назначения. В классе Math определены две константы типа double: E (равная приблизительно 2,72) и PI (равная примерно 3,14).

Тригонометрические функции

Методы, перечисленные в табл. 17.16, принимают параметр типа double, выражающий угол в радианах и возвращающий результат соответствующей тригонометрической функции.

Таблица 17.16. Методы из класса Math, представляющие прямые тригонометрические функции

Метод	Описание
<code>static double sin(double аргумент)</code>	Возвращает синус угла, определяемого параметром <i>аргумент</i> в радианах
<code>static double cos(double аргумент)</code>	Возвращает косинус угла, определяемого параметром <i>аргумент</i> в радианах
<code>static double tan(double аргумент)</code>	Возвращает тангенс угла, определяемого параметром <i>аргумент</i> в радианах

Методы, перечисленные в табл. 17.17, принимают в качестве параметра результат выполнения тригонометрической функции и возвращают в качестве результата угол в радианах. Они представляют тригонометрические функции, обратные приведенным в табл. 17.16.

Таблица 17.17. Методы из класса Math, представляющие обратные тригонометрические функции

Метод	Описание
<code>static double asin(double аргумент)</code>	Возвращает угол, синус которого определяет указанный <i>аргумент</i>
<code>static double acos(double аргумент)</code>	Возвращает угол, косинус которого определяет указанный <i>аргумент</i>
<code>static double atan(double аргумент)</code>	Возвращает угол, тангенс которого определяет указанный <i>аргумент</i>
<code>static double atan2(double x, double y)</code>	Возвращает угол, тангенс которого равен x/y

Методы, перечисленные в табл. 17.18, вычисляют гиперболический синус, косинус и тангенс угла.

Таблица 17.18. Методы из класса Math, представляющие гиперболические функции

Метод	Описание
<code>static double sinh(double аргумент)</code>	Возвращает гиперболический синус угла, определяемого параметром <i>аргумент</i> в радианах
<code>static double cosh(double аргумент)</code>	Возвращает гиперболический косинус угла, определяемого параметром <i>аргумент</i> в радианах
<code>static double tanh(double аргумент)</code>	Возвращает гиперболический тангенс угла, определяемого параметром <i>аргумент</i> в радианах

Экспоненциальные функции

В классе Math определен ряд методов, представляющих экспоненциальные функции (табл. 17.19).

Таблица 17.19. Методы из класса Math, представляющие экспоненциальные функции

Метод	Описание
<code>static double cbrt(double аргумент)</code>	Возвращает кубический корень из указанного <i>аргумента</i>
<code>static double exp(double аргумент)</code>	Возвращает показатель степени указанного <i>аргумента</i>
<code>static double expm1(double аргумент)</code>	Возвращает показатель степени указанного <i>аргумента</i> -1
<code>static double log(double аргумент)</code>	Возвращает натуральный алгоритм указанного <i>аргумента</i>
<code>static double log10(double аргумент)</code>	Возвращает логарифм по основанию 10 от указанного <i>аргумента</i>
<code>static double log1p(double аргумент)</code>	Возвращает натуральный логарифм указанного <i>аргумента</i> +1
<code>static double pow(double y, double x)</code>	Возвращает <i>y</i> в степени <i>x</i> , например, в результате вызова <code>pow(2.0, 3.0)</code> возвращается числовое значение 8.0
<code>static double scalb(double аргумент, int показатель)</code>	Возвращает результат $\text{аргумент} \times 2^{\text{показатель}}$
<code>static float scalb(float аргумент, int показатель)</code>	Возвращает результат $\text{аргумент} \times 2^{\text{показатель}}$
<code>static double sqrt(double аргумент)</code>	Возвращает квадратный корень из указанного <i>аргумента</i>

Функции округления

В классе Math определены также методы, предназначенные для выполнения различных операций округления (табл. 17.20). Обратите внимание на два метода

`ulp()` в конце таблицы. В данном контексте имя метода `ulp` означает *количество единиц в последнем знаке*, т.е. расстояние между текущим значением и ближайшим большим значением. Эту особенность можно использовать для достижения нужной точности результата.

Таблица 17.20. Методы округления из класса `Math`

Метод	Описание
<code>static int abs(int аргумент)</code>	Возвращает абсолютное значение указанного <i>аргумента</i>
<code>static long abs(long аргумент)</code>	Возвращает абсолютное значение указанного <i>аргумента</i>
<code>static float abs(float аргумент)</code>	Возвращает абсолютное значение указанного <i>аргумента</i>
<code>static double abs(double аргумент)</code>	Возвращает абсолютное значение указанного <i>аргумента</i>
<code>static double ceil(double аргумент)</code>	Возвращает наименьшее целое число, которое больше или равно указанному <i>аргументу</i>
<code>static double floor(double аргумент)</code>	Возвращает наибольшее целое число, которое меньше или равно указанному <i>аргументу</i>
<code>static int floorDiv(int делимое, int делитель)</code>	Возвращает результат целочисленного деления <i>делимое/делитель</i> (добавлен в версии JDK 8)
<code>static int floorDiv(long делимое, long делитель)</code>	Возвращает результат целочисленного деления <i>делимое/делитель</i> (добавлен в версии JDK 8)
<code>static int floorMod(int делимое, int делитель)</code>	Возвращает наименьший целый остаток от деления <i>делимое/делитель</i> (добавлен в версии JDK 8)
<code>static int floorMod(long делимое, long делитель)</code>	Возвращает наименьший целый остаток от деления <i>делимое/делитель</i> (добавлен в версии JDK 8)
<code>static int max(int x, int y)</code>	Возвращает большее из двух чисел <i>x</i> и <i>y</i>
<code>static long max(long x, long y)</code>	Возвращает большее из двух чисел <i>x</i> и <i>y</i>
<code>static float max(float x, float y)</code>	Возвращает большее из двух чисел <i>x</i> и <i>y</i>
<code>static double max(double x, double y)</code>	Возвращает большее из двух чисел <i>x</i> и <i>y</i>
<code>static int min(int x, int y)</code>	Возвращает меньшее из двух чисел <i>x</i> и <i>y</i>
<code>static long min(long x, long y)</code>	Возвращает меньшее из двух чисел <i>x</i> и <i>y</i>
<code>static float min(float x, float y)</code>	Возвращает меньшее из двух чисел <i>x</i> и <i>y</i>
<code>static double min(double x, double y)</code>	Возвращает меньшее из двух чисел <i>x</i> и <i>y</i>
<code>static double nextAfter(double аргумент, double направление)</code>	Начиная со значения указанного <i>аргумента</i> , возвращает следующее значение в заданном <i>направлении</i> . Если <i>аргумент=направление</i> , то возвращается заданное <i>направление</i>

Метод	Описание
<code>static float nextAfter(float аргумент, double направление)</code>	Начиная со значения указанного <i>аргумента</i> , возвращает следующее значение в заданном <i>направлении</i> . Если <i>аргумент</i> — <i>направление</i> , то возвращается заданное <i>направление</i>
<code>static double nextDown(double val)</code>	Возвращает следующее значение, меньшее, чем <i>val</i> (добавлен в версии JDK 8)
<code>static float nextDown(float val)</code>	Возвращает следующее значение, меньшее, чем <i>val</i> (добавлен в версии JDK 8)
<code>static double nextUp(double аргумент)</code>	Возвращает следующее значение в положительном направлении от указанного <i>аргумента</i>
<code>static float nextUp(float аргумент)</code>	Возвращает следующее значение в положительном направлении от указанного <i>аргумента</i>
<code>static double rint(double аргумент)</code>	Возвращает ближайшее к указанному <i>аргументу</i> целое значение
<code>static int round(float аргумент)</code>	Возвращает указанный <i>аргумент</i> , округленный до ближайшего значения типа <code>int</code>
<code>static long round(double аргумент)</code>	Возвращает <i>аргумент</i> , округленный до ближайшего значения типа <code>long</code>
<code>static float ulp(float аргумент)</code>	Возвращает количество единиц в последнем знаке для указанного <i>аргумента</i>
<code>static double ulp(double аргумент)</code>	Возвращает количество единиц в последнем знаке для указанного <i>аргумента</i>

Прочие методы из класса Math

Помимо методов, перечисленных в приведенных выше таблицах, в классе `Math` определен ряд других методов, представленных в табл. 17.21. Обратите внимание на суффикс **Exact** в именах некоторых из этих методов. Эти методы были внедрены в версии JDK 8. Они генерируют исключение типа `ArithmeticException`, если происходит переполнение. Следовательно, эти методы предоставляют простой способ следить за переполнением при выполнении различных операций.

Таблица 17.21. Прочие методы из класса `Math`

Метод	Описание
<code>static int addExact(int аргумент1, int аргумент2)</code>	Возвращает результат сложения <i>аргумент1</i> + <i>аргумент2</i> . Генерирует исключение типа <code>ArithmeticException</code> , если происходит переполнение (добавлен в версии JDK 8)
<code>static long addExact(long аргумент1, long аргумент2)</code>	Возвращает результат сложения <i>аргумент1</i> + <i>аргумент2</i> . Генерирует исключение типа <code>ArithmeticException</code> , если происходит переполнение (добавлен в версии JDK 8)
<code>static double copySign(double аргумент, double знак_аргумента)</code>	Возвращает указанный <i>аргумент</i> с таким же знаком, как и у параметра <i>знак_аргумента</i>

Продолжение табл. 17.21

Метод	Описание
<code>static float copySign(float аргумент, float знак_аргумента)</code>	Возвращает <i>аргумент</i> с таким же знаком, как и у параметра <i>знак_аргумента</i>
<code>static int decrementExact(int аргумент)</code>	Возвращает результат отрицательного приращения <i>аргумент</i> -1. Генерирует исключение типа ArithmeticException , если происходит переполнение (добавлен в версии JDK 8)
<code>static long decrementExact(long аргумент)</code>	Возвращает результат отрицательного приращения <i>аргумент</i> -1. Генерирует исключение типа ArithmeticException , если происходит переполнение (добавлен в версии JDK 8)
<code>static int getExponent(double аргумент)</code>	Возвращает показатель степени 2, используемый для двоичного представления указанного <i>аргумента</i>
<code>static int getExponent(float аргумент)</code>	Возвращает показатель степени 2, используемый для двоичного представления указанного <i>аргумента</i>
<code>static hypot(double сторона1, double сторона2)</code>	Возвращает длину гипотенузы прямоугольного треугольника по длине двух противоположных сторон
<code>static double IEEERemainder(double делимое, double делитель)</code>	Возвращает остаток от деления <i>делимое</i> / <i>делитель</i>
<code>static int incrementExact(int аргумент)</code>	Возвращает результат приращения <i>аргумент</i> +1. Генерирует исключение типа ArithmeticException , если происходит переполнение (добавлен в версии JDK 8)
<code>static long incrementExact(long аргумент)</code>	Возвращает результат приращения <i>аргумент</i> +1. Генерирует исключение типа ArithmeticException , если происходит переполнение (добавлен в версии JDK 8)
<code>static int multiplyExact(int аргумент1, int аргумент2)</code>	Возвращает результат умножения <i>аргумент1</i> * <i>аргумент2</i> . Генерирует исключение типа ArithmeticException , если происходит переполнение (добавлен в версии JDK 8)
<code>static long multiplyExact(long аргумент1, long аргумент2)</code>	Возвращает результат умножения <i>аргумент1</i> * <i>аргумент2</i> . Генерирует исключение типа ArithmeticException , если происходит переполнение (добавлен в версии JDK 8)
<code>static int negateExact(int аргумент)</code>	Возвращает результат отрицания - <i>аргумент</i> . Генерирует исключение типа ArithmeticException , если происходит переполнение (добавлен в версии JDK 8)
<code>static long negateExact(long аргумент)</code>	Возвращает результат отрицания - <i>аргумент</i> . Генерирует исключение типа ArithmeticException , если происходит переполнение (добавлен в версии JDK 8)
<code>static double random()</code>	Возвращает псевдослучайное число в пределах от 0 до 1
<code>static float signum(double аргумент)</code>	Определяет знак значения. Возвращает нулевое значение, если указанный <i>аргумент</i> равен нулю; значение 1, если <i>аргумент</i> больше нуля; и значение -1, если <i>аргумент</i> меньше нуля

Метод	Описание
static float signum(float аргумент)	Определяет знак значения. Возвращает нулевое значение, если указанный <i>аргумент</i> равен нулю; значение 1, если <i>аргумент</i> больше нуля; и значение -1, если <i>аргумент</i> меньше нуля
static int subtractExact(int аргумент1, int аргумент2)	Возвращает результат вычитания <i>аргумент1</i> - <i>аргумент2</i> . Генерирует исключение типа ArithmeticException , если происходит переполнение (добавлен в версии JDK 8)
static long subtractExact(int аргумент1, long аргумент2)	Возвращает результат вычитания <i>аргумент1</i> - <i>аргумент2</i> . Генерирует исключение типа ArithmeticException , если происходит переполнение (добавлен в версии JDK 8)
static double toDegrees(double угол)	Преобразует радианы в градусы, причем заданный <i>угол</i> должен быть указан в радианах. Возвращает полученный результат в градусах
static int toIntExact(long аргумент)	Возвращает указанный <i>аргумент</i> в виде значения типа int . Генерирует исключение типа ArithmeticException , если происходит переполнение (добавлен в версии JDK 8)
static double toRadians(double угол)	Преобразует градусы в радианы, причем заданный <i>угол</i> должен быть указан в градусах. Возвращает полученный результат в радианах

В следующем примере программы демонстрируется применение методов `toRadians()` и `toDegrees()`:

```
// Продемонстрировать применение методов toDegrees() и toRadians()
class Angles {
    public static void main(String args[]) {
        double theta = 120.0;

        System.out.println(theta + " градусов равно " +
            Math.toRadians(theta) + " радиан.");

        theta = 1.312;
        System.out.println(theta + " радиан равно " +
            Math.toDegrees(theta) + " градусов.");
    }
}
```

Ниже приведен результат, выводимый данной программой.

```
120.0 градусов равно 2.0943951023931953 радиан.
1.312 радиан равно 75.17206272116401 градусов.
```

Класс StrictMath

В этом классе определяется полный ряд математических методов, аналогичных тем, что имеются в классе `Math`. Но методы из класса `StrictMath` отличаются тем, что они гарантируют практически одинаковую точность во всех реализациях Java, в то время как методы из класса `Math` обеспечивают меньшую точность ради повышения производительности.

Класс `Compiler`

В классе `Compiler` поддерживается создание таких сред Java, где байт-код компилируется в исполняемый код вместо интерпретируемого. В обычном программировании на Java этот класс не применяется.

Классы `Thread`, `ThreadGroup` и интерфейс `Runnable`

В интерфейсе `Runnable` и классах `Thread` и `ThreadGroup` поддерживается многопоточное программирование. Они рассматриваются ниже по отдельности.

На заметку! Краткий обзор методик, применяемых для управления потоками исполнения, реализации интерфейса `Runnable` и создания многопоточных программ, представлен в главе 11.

Интерфейс `Runnable`

Этот интерфейс должен быть реализован в любом классе, иницилирующем отдельный поток исполнения. В интерфейсе `Runnable` определяется только один абстрактный метод `run()`, который служит точкой входа в поток исполнения. Ниже показано, каким образом определяется этот метод. Он должен быть реализован в создаваемых потоках исполнения.

```
void run()
```

Класс `Thread`

Класс `Thread` создаст новый поток выполнения. В нем реализуется интерфейс `Runnable` и определяются следующие наиболее употребительные конструкторы:

```
Thread()  
Thread(Runnable объект_потока)  
Thread(Runnable объект_потока, String имя_потока)  
Thread(String имя_потока)  
Thread(ThreadGroup группа_потоков, Runnable объект_потока)  
Thread(ThreadGroup группа_потоков, Runnable объект_потока, String имя_потока)  
Thread(ThreadGroup группа_потоков, String имя_потока)
```

где параметр *объект_потока* обозначает экземпляр класса, реализующего интерфейс `Runnable` и определяющего место, где начинается исполнение потока. Имя потока исполнения обозначается параметром *имя_потока*. Если это имя исполнения не указано, оно создается виртуальной машиной JVM. Параметр *группа_потоков* обозначает группу потоков, которой будет принадлежать новый поток исполнения. Если такая группа не указана, новый поток исполнения относится к той же самой группе, что и родительский поток.

В классе `Thread` определены следующие константы:


```
MAX_PRIORITY
MIN_PRIORITY
NORM_PRIORITY
```

Как и следовало ожидать, эти константы определяют максимальный, минимальный и нормальный уровни приоритета потоков исполнения. Методы, определенные в классе `Thread`, перечислены в табл. 17.22. В ранних версиях Java в состав класса `Thread` входили также методы `stop()`, `suspend()` и `resume()`. Но как пояснялось в главе 11, эти методы стали не рекомендованными к употреблению, поскольку они действуют неустойчиво. Не рекомендованным к употреблению является и метод `countStackFrames()`, потому что в нем вызываются методы `suspend()` и `destroy()`, что может привести к взаимной блокировке.

Таблица 17.22. Методы из класса `Thread`

Метод	Описание
<code>static int activeCount()</code>	Возвращает примерное количество активных потоков исполнения в той группе, которой принадлежит данный поток
<code>void checkAccess()</code>	Вынуждает диспетчер защиты проверять, способен ли текущий поток исполнения получать доступ и/или изменить тот поток исполнения, в котором вызывается метод <code>checkAccess()</code>
<code>static Thread currentThread()</code>	Возвращает объект типа <code>Thread</code> , который инкапсулирует поток исполнения, вызывающий этот метод
<code>static void dumpStack()</code>	Выводит стек вызовов потока исполнения
<code>static int enumerate(Thread потоки[])</code>	Размещает копию объектов типа <code>Thread</code> из текущей группы потоков исполнения в указанный массив <i>потоки</i> . Возвращает количество потоков исполнения
<code>static Map<Thread, StackTraceElement[]> getAllStackTraces()</code>	Возвращает объект отображения типа <code>Map</code> , который содержит трассировку стека для всех активных потоков исполнения в группе. Каждая запись в данном отображении состоит из ключа в виде объекта типа <code>Thread</code> и его значения в виде массива элементов типа <code>StackTraceElement</code>
<code>ClassLoader getClassLoader()</code>	Возвращает загрузчик контекста классов, используемый для загрузки классов и ресурсов, предназначенных для текущего потока исполнения
<code>static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler()</code>	Возвращает обработчик необрабатываемых исключений по умолчанию

Продолжение табл. 17.22

Метод	Описание
<code>long getID()</code>	Возвращает идентификатор вызывающего потока исполнения
<code>final String getName()</code>	Возвращает имя потока исполнения
<code>final int getPriority()</code>	Возвращает установленный приоритет потока исполнения
<code>StackTraceElement[] getStackTrace()</code>	Возвращает массив, содержащий трассировку стека вызывающего потока исполнения
<code>Thread.State getState()</code>	Возвращает состояние вызывающего потока исполнения
<code>final ThreadGroup getThreadGroup()</code>	Возвращает объект типа ThreadGroup , членом которого является текущий поток исполнения
<code>static boolean holdsLock(Object объект)</code>	Возвращает логическое значение true , если вызывающий поток владеет блокировкой на заданный <i>объект</i> , а иначе — логическое значение false
<code>void interrupt()</code>	Прерывает поток исполнения
<code>static boolean interrupted()</code>	Возвращает логическое значение true , если текущий поток исполнения прерван, а иначе — логическое значение false
<code>final Boolean isAlive()</code>	Возвращает логическое значение true , если вызывающий поток исполнения еще действует, а иначе — логическое значение false
<code>final Boolean isDaemon()</code>	Возвращает логическое значение true , если вызывающий поток исполнения является потоковым демоном (одним из потоков исполняющей системы Java), а иначе — логическое значение false
<code>boolean isInterrupted()</code>	Возвращает логическое значение true , если вызывающий поток исполнения прерван, а иначе — логическое значение false
<code>final void join() throws InterruptedException</code>	Ожидает завершения потока исполнения
<code>final void join(long миллисекунд) throws InterruptedException</code>	Ожидает завершения потока исполнения, в котором вызван этот метод, до истечения заданного количества <i>миллисекунд</i>
<code>final void join(long миллисекунд, int наносекунд) throws InterruptedException</code>	Ожидает завершения потока исполнения, в котором вызван этот метод, до истечения заданного количества <i>миллисекунд</i> плюс <i>наносекунд</i>
<code>void run()</code>	Начинает исполнение потока
<code>void setContextClassLoader(ClassLoader cl)</code>	Устанавливает <i>cl</i> в качестве загрузчика контекста классов, используемого вызывающим объектом

Метод	Описание
<code>final void setDaemon(boolean <i>остороже</i>)</code>	Помечает поток исполнения как потоковый демон
<code>static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler <i>e</i>)</code>	Устанавливает <i>e</i> в качестве обработчика необрабатываемых исключений по умолчанию
<code>final void setName(String <i>имя_потока</i>)</code>	Устанавливает заданное <i>имя_потока</i> для потока исполнения
<code>final void setPriority(int <i>приоритет</i>)</code>	Устанавливает заданный <i>приоритет</i> для потока исполнения
<code>void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler <i>e</i>)</code>	Устанавливает <i>e</i> в качестве обработчика необрабатываемых исключений по умолчанию для вызывающего потока исполнения
<code>static void sleep(long <i>миллисекунд</i>) throws InterruptedException</code>	Прерывает выполнение потока на заданное количество <i>миллисекунд</i>
<code>static void sleep(long <i>миллисекунд</i>, int <i>микросекунд</i>) throws InterruptedException</code>	Прерывает выполнение потока на заданное количество <i>миллисекунд</i> плюс <i>микросекунд</i>
<code>void start()</code>	Запускает поток на исполнение
<code>String toString()</code>	Возвращает строковое представление потока исполнения
<code>static void yield()</code>	Вызывающий поток исполнения, предлагая уступить ресурс ЦП другому потоку исполнения

Класс ThreadGroup

В этом классе создается группа потоков исполнения. В нем определены два конструктора:

```
ThreadGroup(String имя_группы)
ThreadGroup(ThreadGroup родительский_объект, String имя_группы)
```

В обеих формах конструктора параметр *имя_группы* обозначает имя группы потоков исполнения. В первой форме создается новая группа, родителем которой будет текущий поток исполнения. А во второй форме родитель группы определяется параметром *родительский_объект*. Методы, определенные в классе ThreadGroup и рекомендованные к употреблению, перечислены в табл. 17.23.

Таблица 17.23. Методы из класса ThreadGroup

Метод	Описание
<code>int activeCount()</code>	Возвращает приблизительное количество активных потоков исполнения в вызывающей группе, включая и потоки в подгруппах

Продолжение табл. 17.23

Метод	Описание
<code>int activeGroupCount()</code>	Возвращает приблизительное количество активных групп, включая и подгруппы, для которых вызывающий поток исполнения является родителем
<code>final void checkAccess()</code>	Вынуждает диспетчер защиты проверять, может ли вызывающий поток исполнения получить доступ к группе, для которой вызван метод checkAccess() , и/или изменить ее
<code>final void destroy()</code>	Уничтожает группу потоков исполнения (и любые ее подгруппы), для которой вызывается этот метод
<code>int enumerate(Thread группа[])</code>	Размещает в массиве <i>группа</i> активные потоки исполнения, входящие в группу вызывающего потока, а также в ее подгруппы
<code>int enumerate(Thread группа[], boolean все)</code>	Размещает в массиве <i>группа</i> активные потоки исполнения, входящие в группу вызывающего потока. Если параметр <i>все</i> принимает логическое значение true , то в массиве <i>группа</i> размещаются также все потоки исполнения из подгрупп данного потока
<code>int enumerate(ThreadGroup группа[])</code>	Размещает в массиве <i>группа</i> активные подгруппы (включая подгруппы подгрупп и т.д.) из группы вызывающего потока
<code>int enumerate(ThreadGroup группа[], boolean все)</code>	Размещает в массиве <i>группа</i> активные подгруппы из группы вызывающего потока. Если параметр <i>все</i> принимает логическое значение true , то в массиве <i>группа</i> размещаются также все активные подгруппы всех подгрупп
<code>final int getMaxPriority()</code>	Возвращает максимальный приоритет, установленный для группы
<code>final String getName()</code>	Возвращает имя группы
<code>final ThreadGroup getParent()</code>	Возвращает пустое значение null , если у вызывающего объекта типа ThreadGroup отсутствует родитель. В противном случае возвращается родитель вызывающего объекта
<code>final void interrupt()</code>	Вызывает метод interrupt() для всех потоков исполнения в группе и любых ее подгруппах
<code>final boolean isDaemon()</code>	Возвращает логическое значение true , если текущая группа является демоном, а иначе — логическое значение false
<code>boolean isDestroyed()</code>	Возвращает логическое значение true , если текущая группа уничтожена, а иначе — логическое значение false

Метод	Описание
void list()	Выводит сведения о группе
final boolean parentOf (ThreadGroup <i>группа</i>)	Возвращает логическое значение true , если вызывающий поток исполнения является родителем <i>группы</i> (или самой <i>группой</i>), а иначе — логическое значение false
final void setDaemon (Boolean <i>демон</i>)	Если параметр <i>демон</i> принимает логическое значение true , то вызывающая группа помечается как демон
final void setMaxPriority (int <i>приоритет</i>)	Устанавливает максимальный <i>приоритет</i> для вызывающей группы
String toString()	Возвращает строковое представление группы
void uncaughtException (Thread <i>поток</i>, Throwable <i>e</i>)	Этот метод вызывается, когда исключение становится необрабатываемым

Группы потоков исполнения позволяют управлять ими как единым целым. Это оказывается особенно важно в тех случаях, когда требуется приостановить или продолжить выполнение многих взаимосвязанных потоков исполнения. Допустим, имеется программа, в которой один ряд потоков исполнения используется для печати документов, другой — для отображения документа на экране, а третий — для сохранения документа в файле на диске. Если печать прерывается, то придется прервать все потоки исполнения, имеющие отношение к печати. Такое применение групп потоков исполнения демонстрируется в следующем примере программы, где создаются две группы потоков исполнения — по два потока в каждой.

```
// Продемонстрировать применение групп потоков исполнения
class NewThread extends Thread {
    boolean suspendFlag;

    NewThread(String threadname, ThreadGroup tgOb) {
        super(tgOb, threadname);
        System.out.println("Новый поток: " + this);
        suspendFlag = false;
        start(); // запустить поток исполнения
    }

    // Точка входа в поток исполнения
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println(getName() + ": " + i);
                Thread.sleep(1000);
                synchronized(this) {
                    while(suspendFlag) {
                        wait();
                    }
                }
            }
        } catch (Exception e) {
            System.out.println("Исключение в " + getName());
        }
    }
}
```

```
        System.out.println(getName() + " завершается.");
    }
    synchronized void mysuspend() {
        suspendFlag = true;
    }
    synchronized void myresume() {
        suspendFlag = false;
        notify();
    }
}

class ThreadGroupDemo {
    public static void main(String args[]) {
        ThreadGroup groupA = new ThreadGroup("Группа А");
        ThreadGroup groupB = new ThreadGroup("Группа В");

        NewThread ob1 = new NewThread("Один", groupA);
        NewThread ob2 = new NewThread("Два", groupA);
        NewThread ob3 = new NewThread("Три", groupB);
        NewThread ob4 = new NewThread("Четыре", groupB);

        System.out.println("\nВывод из метода list():");
        groupA.list();
        groupB.list();
        System.out.println();

        System.out.println("Прерывается Группа А");
        Thread tga[] = new Thread[groupA.activeCount()];
        groupA.enumerate(tga); // получить потоки исполнения из группы

        for(int i = 0; i < tga.length; i++) {
            ((NewThread)tga[i]).mysuspend(); // приостановить каждый
                                           // поток исполнения
        }

        try {
            Thread.sleep(4000);
        } catch (InterruptedException e) {
            System.out.println("Главный поток исполнения прерван.");
        }

        System.out.println("Возобновление Группы А");

        for(int i = 0; i < tga.length; i++) {
            ((NewThread)tga[i]).myresume(); // возобновить все потоки
                                           // исполнения в группе
        }

        // ожидать завершения потоков исполнения
        try {
            System.out.println("Ожидание завершения потоков исполнения.");
            ob1.join();
            ob2.join();
            ob3.join();
            ob4.join();
        } catch (Exception e) {
            System.out.println("Исключение в главном потоке исполнения");
        }
        System.out.println("Главный поток исполнения завершен.");
    }
}
```

Ниже приведен примерный результат, выводимый данной программой (у вас он может оказаться иным в зависимости от конкретной исполняющей среды).

```

Новый поток: Thread[Один,5,Группа А]
Новый поток: Thread[Два,5,Группа А]
Новый поток: Thread[Три,5,Группа В]
Новый поток: Thread[Четыре,5,Группа В]
Вывод из list():
java.lang.ThreadGroup[name=Группа А,maxpri=10]
  Thread[Один,5,Группа А]
  Thread[Два,5,Группа А]
java.lang.ThreadGroup[name=Группа В,maxpri=10]
  Thread[Три,5,Группа В]
  Thread[Четыре,5,Группа В]
Прерывается Группа А
Три: 5
Четыре: 5
Три: 4
Четыре: 4
Три: 3
Четыре: 3
Три: 2
Четыре: 2
Возобновление Группы А
Ожидание завершения потоков исполнения.
Один: 5
Два: 5
Три: 1
Четыре: 1
Один: 4
Два: 4
Три завершается.
Четыре завершается.
Один: 3
Два: 3
Один: 2
Два: 2
Один: 1
Два: 1
Один завершается.
Два завершается.
Главный поток исполнения завершен.

```

В этой программе обратите внимание на то, что выполнение группы А приостанавливается на четыре секунды. Как подтверждает результат, выводимый данной программой, приостанавливается выполнение потоков “Один” и “Два”, но потоки “Три” и “Четыре” продолжают выполняться. По истечении четырех секунд выполнение потоков “Один” и “Два” возобновляется. Обратите также внимание на то, как останавливается и возобновляется выполнение группы А. Сначала потоки исполнения из группы А извлекаются путем вызова метода `enumerate()` для этой группы. Затем каждый поток исполнения приостанавливается в процессе обхода результирующего массива. Чтобы продолжить исполнение потоков в группе А, снова делается обход потоков по списку, и каждый поток запускается для продолжения своего исполнения. И последнее замечание: в данном примере применяется рекомендованный подход для приостановки и возобновления потоков исполнения. Это не касается методов `suspend()` и `resume()`, не рекомендованных к употреблению.

Классы ThreadLocal и InheritableThreadLocal

В пакете `java.lang` определены еще два класса, имеющих отношение к потокам исполнения.

- Класс `ThreadLocal` служит для создания локальных переменных потоков исполнения. У каждого потока исполнения будет своя копия локальной переменной.
- Класс `InheritableThreadLocal` служит для создания локальных переменных потоков исполнения, которые могут наследоваться.

Класс Package

Этот класс инкапсулирует данные о версии пакета. Сведения о версии пакета приобретают особую ценность в связи с широким распространением пакетов, а также потому, что в программах на Java могут потребоваться сведения в доступной версии пакета. Методы, определенные в классе `Package`, перечислены в табл. 17.24. А в следующем примере программы демонстрируется применение класса `Package` для вывода списка пакетов, сведения о которых требуются программе в данный момент:

```
// Продемонстрировать применение класса Package
class PkgTest {
    public static void main(String args[]) {
        Package pkgs[];

        pkgs = Package.getPackages();

        for(int i=0; i < pkgs.length; i++)
            System.out.println(
                pkgs[i].getName() + " " +
                pkgs[i].getImplementationTitle() + " " +
                pkgs[i].getImplementationVendor() + " " +
                pkgs[i].getImplementationVersion()
            );
    }
}
```

Таблица 17.24. Методы из класса `Package`

Метод	Описание
<code><A extends Annotation> A getAnnotation(Class<A> тип_аннотации)</code>	Возвращает объект типа <code>Annotation</code> , содержащий аннотацию, связанную с указанным <i>типом_аннотации</i> , для вызывающего объекта
<code>Annotation[] getAnnotations()</code>	Возвращает все аннотации, связанные с вызывающим объектом, в массиве объектов типа <code>Annotation</code> . Возвращает ссылку на этот массив

Метод	Описание
<code><A extends Annotation> A[] getAnnotationsByType(Class<A> <i>тип_аннотации</i>)</code>	Возвращает массив аннотаций, в том числе и повторяющихся аннотаций, имеющих указанный тип_аннотации и связанных с вызывающим объектом (добавлен в версии JDK 8)
<code><A extends Annotation> A[] getDeclaredAnnotation(Class<A> <i>тип_аннотации</i>)</code>	Возвращает объект типа Annotation , содержащий ненаследуемые аннотации, связанные с указанным типом_аннотации (добавлен в версии JDK 8)
<code>Annotation[] getDeclaredAnnotations()</code>	Возвращает объект типа Annotation для всех аннотаций, объявленных в вызывающем объекте (наследуемые аннотации игнорируются)
<code><A extends Annotation> A[] getDeclaredAnnotationsByType(Class<A> <i>тип_аннотации</i>)</code>	Возвращает массив ненаследуемых аннотаций, в том числе и повторяющихся аннотаций, имеющих указанный тип_аннотации и связанных с вызывающим объектом (добавлен в версии JDK 8)
<code>String getImplementationTitle()</code>	Возвращает заголовок вызывающего пакета
<code>String getImplementationVendor()</code>	Возвращает наименование реализатора вызывающего пакета
<code>String getImplementationVersion()</code>	Возвращает номер версии вызывающего пакета
<code>String getName()</code>	Возвращает имя вызывающего пакета
<code>static Package getPackage(String <i>имя_пакета</i>)</code>	Возвращает объект типа Package по указанному имени_пакета
<code>static Package[] getPackages()</code>	Возвращает все пакеты, о которых осведомляется программа, выполняющаяся в данный момент
<code>String getSpecificationTitle()</code>	Возвращает заглавие спецификации вызывающего пакета
<code>String getSpecificationVendor()</code>	Возвращает имя владельца вызывающего пакета из его спецификации
<code>String getSpecificationVersion()</code>	Возвращает номер версии спецификации вызывающего пакета
<code>int hashCode()</code>	Возвращает хеш-код вызывающего пакета
<code>boolean isAnnotationPresent(Class<? extends Annotation> <i>аннотация</i>)</code>	Возвращает логическое значение true , если описываемая аннотация связана с вызывающим объектом, а иначе — логическое значение false
<code>boolean isCompatibleWith(String <i>номер_версии</i>) throws NumberFormatException</code>	Возвращает логическое значение true , если указанный номер_версии меньше или равен номеру версии вызывающего пакета
<code>boolean isSealed()</code>	Возвращает логическое значение true , если вызывающий пакет герметизирован, а иначе — логическое значение false

Окончание табл. 17.24

Метод	Описание
<code>boolean isSealed(URL url)</code>	Возвращает логическое значение true , если вызывающий пакет герметизирован относительно заданного <i>url</i> , а иначе — логическое значение false
<code>String toString()</code>	Возвращает строковый эквивалент вызывающего пакета

Класс RuntimePermission

Класс `RuntimePermission` относится к механизму защиты Java и подробно здесь не рассматривается.

Класс Throwable

В классе `Throwable` поддерживается система обработки исключений в Java. От него происходят все классы исключений. Этот класс подробно рассматривался в главе 10.

Класс SecurityManager

От этого класса можно наследовать подклассы для создания диспетчера защиты. Ссылку на текущий диспетчер защиты можно получить, вызвав метод `getSecurityManager()`, определенный в классе `System`.

Класс StackTraceElement

В классе `StackTraceElement` описывается единственный *стековый фрейм* — отдельный элемент трассировки стека при возникновении исключения. Каждый стековый фрейм представляет *точку выполнения*, которая включает имя класса, метода и файла, а также номер строки исходного кода. Массив элементов типа `StackTraceElement` возвращается при вызове метода `getStackTrace()` из класса `Throwable`.

У класса `StackTraceElement` имеется следующий единственный конструктор:

```
StackTraceElement(String имя_класса, String имя_метода,  
                  String имя_файла, int строка)
```

где параметр *имя_класса* обозначает имя конкретного класса; параметр *имя_метода* — имя конкретного метода; параметр *имя_файла* — имя конкретного файла; параметр *строка* — номер строки кода. В отсутствие достоверного номера строки кода в качестве параметра *строка* следует указать отрицательное значение. Более

того, значение `-2` параметра *строка* означает, что данный стековый фрейм ссылается на платформенно-ориентированный метод.

Методы, поддерживаемые в классе `StackTraceElement`, перечислены в табл. 17.25. Они предоставляют программе доступ к трассировке ее стека.

Таблица 17.25. Методы из класса `StackTraceElement`

Метод	Описание
<code>boolean equals(Object объект)</code>	Возвращает логическое значение true , если вызывающий объект типа StackTraceElement оказывается таким же, как и заданный <i>объект</i> , а иначе — логическое значение false
<code>String getClassName()</code>	Возвращает имя класса, в котором точка выполнения описывается вызывающим объектом типа StackTraceElement
<code>String getFileName()</code>	Возвращает имя файла, где исходный код точки выполнения описывается хранимым вызывающим объектом типа StackTraceElement
<code>int getLineNumber()</code>	Возвращает номер строки исходного кода, где точка выполнения описывается вызывающим объектом типа StackTraceElement . Иногда номер строки кода недоступен, и тогда возвращается отрицательное значение
<code>String getMethodName()</code>	Возвращает имя метода, в котором точка выполнения описывается вызывающим объектом типа StackTraceElement
<code>int hashCode()</code>	Возвращает хеш-код вызывающего объекта типа StackTraceElement
<code>boolean isNativeMethod()</code>	Возвращает логическое значение true , если точка выполнения описывается вызывающим объектом типа StackTraceElement , оказывающимся в платформенно-ориентированном методе, а иначе — логическое значение false
<code>String toString()</code>	Возвращает строковое представление вызывающей последовательности

Класс Enum

Как пояснялось в главе 12, *перечисление* — это список именованных констант. (Напомним, что перечисление создается с помощью ключевого слова `enum`.) Все перечисления автоматически наследуются от класса `Enum`. Класс `Enum` является обобщенным и объявляется следующим образом:

```
class Enum<E> extends Enum<E>
```

где **E** обозначает перечислимый тип. У класса `Enum` отсутствуют открытые конструкторы.

В классе Enum определен ряд методов, доступных для использования во всех перечислениях (табл. 17.26).

Таблица 17.26. Методы из класса Enum

Метод	Описание
protected final Object clone() throws CloneNotSupportedException	Вызов этого метода инициирует генерирование исключения типа CloneNotSupportedException . Этим предотвращается клонирование перечислений
final int compareTo(E e)	Сравнивает порядковое значение двух констант одного того же перечисления. Возвращает отрицательное значение, если порядковое значение вызывающей константы меньше <i>e</i> , нулевое значение, если порядковые значения обеих констант совпадают; и положительное значение, если порядковое значение вызывающей константы больше <i>e</i>
final boolean equals(Object объект)	Возвращает логическое значение true , если заданный <i>объект</i> и вызывающий объект ссылаются на одну и ту же константу
final Class<E> getDeclaringClass()	Возвращает тип перечисления, членом которого является вызывающая константа
final int hashCode()	Возвращает хеш-код вызывающего объекта
final String name()	Возвращает неизменяемое имя вызывающей константы
final int ordinal()	Возвращает значение, обозначающее позицию константы перечислимого типа в списке констант
String toString()	Возвращает имя вызывающей константы. Это имя может отличаться от использовавшегося при объявлении перечисления
static <T extends Enum<T>> T valueOf(Class<T> тип_перечисления, String имя)	Возвращает константу, связанную с указанным именем в заданном <i>типе_перечисления</i>

Класс ClassValue

Применяется для связи значения с типом. Это класс обобщенного типа, определяемый следующим образом:

```
Class ClassValue<T>
```

Он предназначен для узкоспециализированного применения, а не для обычного программирования.

Интерфейс CharSequence

В интерфейсе `CharSequence` определяются методы, предоставляющие доступ только для чтения к последовательности символов. Все его методы перечислены в табл. 17.27. Этот интерфейс реализуется в классах `String`, `StringBuffer`, `StringBuilder` и пр.

Таблица 17.27. Методы из интерфейса `CharSequence`

Метод	Описание
<code>char charAt(int индекс)</code>	Возвращает символ, находящийся на позиции, обозначаемой заданным <i>индексом</i>
<code>default InputStream chars()</code>	Возвращает поток ввода-вывода (в форме интерфейса <code>InputStream</code>) для символов в вызывающем объекте (добавлен в версии JDK 8)
<code>default InputStream codePoints()</code>	Возвращает поток ввода-вывода (в форме интерфейса <code>InputStream</code>) для кодовых точек в вызывающем объекте (добавлен в версии JDK 8)
<code>int length()</code>	Возвращает количество символов в вызывающей последовательности
<code>CharSequence subSequence(int начальный_индекс, int конечный_индекс)</code>	Возвращает подмножество вызывающей последовательности от позиции <i>начальный_индекс</i> и до позиции <i>конечный_индекс-1</i>
<code>String toString()</code>	Возвращает строковое представление вызывающей последовательности

Интерфейс Comparable

Объекты классов, реализующих интерфейс `Comparable`, могут быть упорядочены. Иными словами, классы, реализующие интерфейс `Comparable`, содержат объекты, которые можно сравнивать некоторым целенаправленным образом. Интерфейс `Comparable` является обобщенным и объявляется следующим образом:

```
interface Comparable<T>
```

где **T** обозначает тип сравниваемых объектов. В интерфейсе `Comparable` объявляется один метод, который используется для определения того, что в языке Java называется *естественным упорядочением* экземпляров класса. Ниже приведена общая форма этого метода.

```
int compareTo(T объект)
```

Этот метод сравнивает вызывающий объект с заданным *объектом*. Он возвращает нулевое значение, если значения обоих объектов равны; отрицательное значение, если вызывающий объект имеет меньшее значение, а иначе — положительное значение.

Этот интерфейс реализуется в нескольких рассмотренных ранее классах. В частности, метод `compareTo()` определяется в классах `Byte`, `Character`, `Double`, `Float`, `Long`, `Short`, `String`, `Integer` и `Enum`. Кроме того, объекты классов, реализующих этот интерфейс, могут быть использованы в разных коллекциях, как поясняется в следующей главе.

Интерфейс Appendable

Объекты классов, реализующих интерфейс `Appendable`, позволяют добавлять к ним символы или символьные последовательности. В интерфейсе `Appendable` определяются три формы методов:

```
Appendable append(char символ) throws IOException
Appendable append(CharSequence символы) throws IOException
Appendable append(CharSequence символы, int начало, int конец)
    throws IOException
```

В первой форме заданный *символ* добавляется к вызывающему объекту. Во второй форме к вызывающему объекту добавляется последовательность символов, обозначаемая параметром *символы*. А третья форма позволяет указать часть последовательности символов (от позиции *начало* до позиции *конец-1*), обозначаемой параметром *символы*. Но в любом случае возвращается ссылка на вызывающий объект.

Интерфейс Iterable

Интерфейс `Iterable` должен быть реализован всеми классами, объекты которых предполагается использовать в цикле `for` в стиле `for each`. Иными словами, чтобы использовать объект в цикле `for` в стиле `for each`, его класс должен реализовывать интерфейс `Iterable`. Этот интерфейс является обобщенным и объявляется следующим образом:

```
interface Iterable<T>
```

где *T* обозначает тип объектов, итерируемых (т.е. перебираемых) в цикле. В интерфейсе `Iterable` определяется метод `iterator()`, который возвращает итератор элементов, содержащихся в вызывающем объекте:

```
Iterator<T> iterator()
```

Начиная с версии JDK 8 в интерфейсе `Iterable` определяются также два метода по умолчанию. Первый из них — `forEach()` — объявляется следующим образом:

```
default void forEach(Consumer<? super T> действие)
```

Для каждого перебираемого в цикле элемента метод `forEach()` выполняет код, определяемый параметром *действие*. Здесь `Consumer` обозначает функциональный интерфейс, внедренный в версии JDK 8 и определяемый в пакете `java.util.function` (он будет рассмотрен в главе 19).

Второй метод по умолчанию — `splititerator()` — объявляется следующим образом:

```
default Splititerator<T> splititerator()
```

Этот метод возвращает объект типа `Splititerator` в виде итератора-разделителя для перебираемой в цикле последовательности. (Подробнее об итераторах-разделителях речь пойдет в главах 18 и 29.)

На заметку! Итераторы рассматриваются в главе 18.

Интерфейс `Readable`

Интерфейс `Readable` указывает на то, что объект может быть использован в качестве источника для чтения символов. В этом интерфейсе определен единственный метод `read()`, как показано ниже.

```
int read(CharBuffer буфер) throws IOException
```

Этот метод читает символы в заданный буфер. Он возвращает количество прочитанных символов или значение `-1`, если достигнут знак конца файла (EOF).

Интерфейс `AutoCloseable`

Интерфейс `AutoCloseable` обеспечивает поддержку оператора `try` с ресурсами, реализующего механизм, который иногда называют *автоматическим управлением ресурсами* (ARM). Оператор `try` с ресурсами автоматизирует процесс освобождения ресурса вроде потока ввода-вывода, когда он больше не нужен. (Подробнее об этом см. в главе 13.) Только объекты классов, реализующих интерфейс `AutoCloseable`, могут применяться вместе с оператором `try` с ресурсами. В интерфейсе `AutoCloseable` определяется единственный метод `close()`, как показано ниже.

```
void close() throws Exception
```

Этот метод закрывает вызывающий объект, освобождая любые ресурсы, которые он мог удерживать. Он автоматически вызывается в конце оператора `try` с ресурсами, избавляя таким образом от необходимости явно вызывать метод `close()`. Интерфейс `AutoCloseable` реализуется несколькими классами, включая все классы, открывающие поток ввода-вывода, который может быть закрыт.

Интерфейс `Thread.UncaughtExceptionHandler`

Статический интерфейс `Thread.UncaughtExceptionHandler` реализуется классами, в которых требуется обрабатывать необработываемые исключения. Он реализуется, в частности, в классе `ThreadGroup`. В этом интерфейсе объявлен следующий единственный метод:

```
void uncaughtException(Thread поток, Throwable исключение)
```

где параметр *поток* обозначает ссылку на поток исполнения, сгенерировавший исключение, а *исключение* — ссылку на это исключение.

Подпакеты из пакета `java.lang`

В Java определен ряд следующих подпакетов, входящих в состав пакета `java.lang`:

- `java.lang.annotation`
- `java.lang.instrument`
- `java.lang.invoke`
- `java.lang.management`
- `java.lang.ref`
- `java.lang.reflect`

Далее каждый из этих подпакетов описывается вкратце.

Пакет `java.lang.annotation`

Средства аннотирования в Java поддерживаются с помощью пакета `java.lang.annotation`. В этом пакете определяются интерфейс `Annotation`, а также перечисления `ElementType` и `RetentionPolicy`. (Интерфейс `Annotation` описывался в главе 12.)

Пакет `java.lang.instrument`

В этом пакете предоставляются средства, которые могут быть использованы для дополнения необходимым инструментарием различных аспектов выполнения программ. В нем определяются интерфейсы `Instrumentation` и `ClassFileTransformer`, а также класс `ClassDefinition`.

Пакет `java.lang.invoke`

В этом пакете поддерживаются динамические языки. Он содержит такие классы, как `CallSite`, `MethodHandle` и `MethodType`.

Пакет `java.lang.management`

Обеспечивает поддержку управления виртуальной машиной JVM и исполняющей средой. Используя средства из пакета `java.lang.management`, можно просматривать различные аспекты выполнения программы и управлять ими.

Пакет `java.lang.ref`

Как упоминалось ранее, средства сборки “мусора” в Java автоматически определяют момент, когда не остается ссылок на объект. И тогда предполагается, что этот объект больше не требуется, а занятую им память можно освободить. Классы из пакета `java.lang.ref` предоставляют возможности более гибкого управления процессом сборки “мусора”.

Пакет `java.lang.reflect`

Рефлексия — это способность программы анализировать код во время выполнения. Пакет `java.lang.reflect` позволяет получать сведения о полях, конструкторах, методах и модификаторах класса. Помимо прочего, эти сведения могут понадобиться для создания программных инструментов, которые позволяют работать с компонентами JavaBeans. В таких инструментах рефлексия используется для динамического определения характеристик компонента. Рефлексия была представлена в главе 12 и также рассматривается в главе 30.

В пакете `java.lang.reflect` определяется несколько классов, в том числе `Method`, `Field` и `Constructor`, а также несколько интерфейсов, включая `AnnotatedElement`, `Member` и `Type`. Помимо этого, в состав пакета `java.lang.reflect` входит класс `Array`, позволяющий динамически создавать массивы и оперировать ими.

С этой главы начинается рассмотрение классов и интерфейсов, определенных в пакете java.util. Этот важный пакет предлагает большой выбор классов и интерфейсов, поддерживающих обширный ряд функциональных возможностей. В частности, в пакет java.util входят классы, создающие псевдослучайные числа, управляющие датой и временем, просмотром событий, манипулирующие наборами битов, выполняющие синтаксический анализ символьных строк и обрабатывающие форматированные данные. В пакет java.util входит также одна из самых эффективных подсистем Java – каркас коллекций Collections Framework. Этот каркас представляет собой сложную иерархию интерфейсов и классов, реализующих современную технологию управления группами объектов. Он заслуживает пристального внимания всех программирующих на Java.

Пакет java.util предоставляет обширный ряд функциональных возможностей, поэтому он достаточно объемный. Ниже приведен перечень его основных классов.

AbstractCollection	FormattableFlags	Properties
AbstractList	Formatter	PropertyPermission
AbstractMap	GregorianCalendar	PropertyResourceBundle
AbstractQueue	HashMap	Random
AbstractSequentialList	HashSet	ResourceBundle
AbstractSet	Hashtable	Scanner
ArrayDeque	IdentityHashMap	ServiceLoader
ArrayList	IntSummaryStatistics (добавлен в версии JDK 8)	SimpleTimeZone
Arrays	LinkedHashMap	Spliterators (добавлен в версии JDK 8)
Base64 (добавлен в версии JDK 8)	LinkedHashSet	SplitableRandom (добавлен в версии JDK 8)
BitSet	LinkedList	Stack
Calendar	ListResourceBundle	StringJoiner (добавлен в версии JDK 8)
Collections	Locale	StringTokenizer
Currency	LongSummaryStatistics (добавлен в версии JDK 8)	Timer
Date	Objects	TimerTask
Dictionary	Observable	TimeZone

DoubleSummaryStatistics (добавлен в версии JDK 8)	Optional (добавлен в версии JDK 8)	TreeMap
EnumMap	OptionalDouble (добавлен в версии JDK 8)	TreeSet
EnumSet	OptionalInt (добавлен в версии JDK 8)	UUID
EventListenerProxy	OptionalLong (добавлен в версии JDK 8)	Vector
EventObject	PriorityQueue	WeakHashMap

В пакете `java.util` определены следующие интерфейсы:

Collection	Map.Entry	Set
Comparator	NavigableMap	SortedMap
Deque	NavigableSet	SortedSet
Enumeration	Observer	Splititerator (добавлен в версии JDK 8)
EventListener	PrimitiveIterator (добавлен в версии JDK 8)	Splititerator.OfDouble (добавлен в версии JDK 8)
Formattable	PrimitiveIterator.OfDouble (добавлен в версии JDK 8)	Splititerator.OfInt (добавлен в версии JDK 8)
Iterator	PrimitiveIterator.OfInt (добавлен в версии JDK 8)	Splititerator.OfLong (добавлен в версии JDK 8)
List	PrimitiveIterator.OfLong (добавлен в версии JDK 8)	Splititerator.OfPrimitive (добавлен в версии JDK 8)
ListIterator	Queue	
Map	RandomAccess	

Из-за большого размера пакета `java.util` его описание разделено на две главы. Эта глава посвящена средствам из каркаса коллекций `Collections Framework`, входящего в пакет `java.util`. А в главе 19 рассматриваются остальные классы и интерфейсы из этого пакета.

Краткий обзор коллекций

Каркас коллекций `Collections Framework` в Java стандартизирует способы управления группами объектов в прикладных программах. Коллекции не были частью исходной версии языка Java, но были внедрены в версии J2SE 1.2. До появления каркаса коллекций для хранения групп объектов и манипулирования ими в Java предоставлялись такие специальные классы, как `Dictionary`, `Vector`, `Stack` и `Properties`. И хотя эти классы были достаточно удобны, им недоставало общей, объединяющей основы. Так, класс `Vector` отличался способом своего применения от класса `Properties`. Такой первоначальный специализированный подход не был рассчитан на дальнейшее расширение и адаптацию. Для разрешения этого и ряда других затруднений и были внедрены коллекции.

Каркас коллекций был разработан для достижения нескольких целей. Во-первых, он должен был обеспечивать высокую производительность. Реализация основных коллекций (динамических массивов, связанных списков, деревьев и хеш-

таблиц) отличается высокой эффективностью. Программировать один из таких “механизмов доступа к данным” вручную приходится крайне редко. Во-вторых, каркас должен был обеспечивать единообразное функционирование коллекций с высокой степенью взаимодействия. В-третьих, коллекции должны были допускать простое расширение и/или адаптацию. В этом отношении весь каркас коллекций построен на едином наборе стандартных интерфейсов. Некоторые стандартные реализации этих интерфейсов (например, в классах `LinkedList`, `HashSet` и `TreeSet`) можно использовать в исходном виде. Но при желании можно реализовать и свои коллекции. Для удобства программистов предусмотрены различные реализации специального назначения, а также частичные реализации, которые облегчают создание собственных коллекций. И наконец, в каркас коллекций были внедрены механизмы интеграции стандартных массивов.

Алгоритмы составляют другую важную часть каркаса коллекций. Алгоритмы оперируют коллекциями и определены в виде статических методов в классе `Collections`. Таким образом, они доступны всем коллекциям и не требуют реализации их собственной версии в каждом классе коллекции. Алгоритмы предоставляют стандартные средства для манипулирования коллекциями.

Другим элементом, тесно связанным с каркасом коллекций, является интерфейс `Iterator`, определяющий *итератор*, который обеспечивает общий, стандартизированный способ поочередного доступа к элементам коллекций. Иными словами, итератор предоставляет способ *перебора содержимого коллекций*. А поскольку каждая коллекция предоставляет свой итератор, то элементы любого класса коллекций могут быть доступны с помощью методов, определенных в интерфейсе `Iterator`. Таким образом, код, перебирающий в цикле элементы множества, можно с минимальными изменениями применить, например, для перебора элементов списка.

В версии JDK 8 внедрена другая разновидность итератора, называемая *итератором-разделителем*. Если говорить коротко, то итераторы-разделители обеспечивают параллельную итерацию. Итераторы-разделители поддерживаются в интерфейсе `SplitIterator` и ряде вложенных в него интерфейсов, которые, в свою очередь, поддерживают примитивные типы данных. В версии JDK 8 внедрены также интерфейсы итераторов, предназначенные для применения вместе с примитивными типами данных. К их числу относятся интерфейсы `PrimitiveIterator` и `PrimitiveIterator.OfDouble`.

Помимо коллекций, в каркасе Collections Framework определен ряд интерфейсов и классов *отображений*, в которых хранятся пары “ключ–значение”. Несмотря на то что отображения входят в состав каркаса коллекций, строго говоря, они не являются коллекциями. Тем не менее для отображения можно получить *представление коллекции*. Такое представление содержит элементы отображения, хранящиеся в коллекции. Таким образом, содержимое отображения можно при желании обрабатывать как коллекцию.

Механизм коллекций был усовершенствован для некоторых классов, изначально определенных в пакете `java.util` таким образом, чтобы интегрировать их в новую систему. Но несмотря на то что внедрение коллекций изменило архитектуру многих первоначальных служебных классов, они не стали от этого нерекомендованными к употреблению. Коллекции просто предлагают лучшее решение некоторых задач.

На заметку! Если у вас имеется некоторый опыт программирования на C++, то вам может помочь сходство коллекций в Java со стандартной библиотекой шаблонов (STL), определенной в C++. То, что в C++ называется *контейнером*, в Java именуется *коллекцией*. Но у каркаса коллекций Collections Framework и библиотеки STL имеются существенные отличия. Поэтому не делайте поспешных выводов.

Изменения каркаса коллекций в версии JDK 5

С выходом версии JDK 5 в каркасе коллекций Collections Framework произошел ряд существенный изменений, значительно повысивших его эффективность и упростивших его применение. К числу этих изменений относится внедрение обобщений, автоматическая упаковка и распаковка, а также организация цикла `for` в стиле `for each`. И хотя JDK 8 является уже третьей основной версией Java, появившейся после выпуска JDK 5, последствия изменений, внесенных в каркас коллекций в версии JDK 5, настолько значительны, что они до сих пор заслуживают особого внимания. Основная причина состоит в том, что по-прежнему существует и эксплуатируется большой объем кода, написанного до версии JDK 5. Понимать последствия и причины этих изменений очень важно на тот случай, если придется сопровождать или обновлять устаревший код.

Обобщения коренным образом изменили каркас коллекций

Внедрение обобщений коренным образом изменило каркас коллекций, поскольку он был полностью переделан в связи этим нововведением. Все коллекции теперь являются обобщенными, и многие методы, оперирующие коллекциями, также принимают обобщенные параметры. Проще говоря, внедрение обобщений коснулось всех частей каркаса коллекций.

Обобщения внесли в коллекции именно то, чего им явно недоставало: типовую безопасность. Раньше во всех коллекциях хранились ссылки на класс `Object`, а это означало, что в любой коллекции могли храниться объекты любого типа. Таким образом, в одной коллекции можно было непреднамеренно сохранить несовместимые типы данных. А это могло привести к ошибкам несовместимости типов во время выполнения. Благодаря обобщениям теперь можно явным образом указать тип сохраняемых данных и тем самым избежать подобных ошибок во время выполнения.

Несмотря на то что внедрение обобщений изменило объявления большинства классов и интерфейсов, а также некоторых их методов, в целом каркас коллекций действует таким же образом, как и до появления обобщений. Безусловно, чтобы извлечь выгоду, которую обобщения приносят коллекциям, придется переписать устаревший код. Это важно сделать еще и потому, что при компилировании кода, написанного до появления обобщений, современный компилятор будет выдавать предупреждающие сообщения. Во избежание подобных сообщений в код всех создаваемых коллекций придется ввести сведения об их типе.

В средствах автоматической упаковки используются примитивные типы данных

Автоматическая упаковка и распаковка упрощают сохранение данных примитивных типов в коллекциях. Как будет показано далее, в коллекциях можно сохранять только ссылки, но не значения примитивных типов. Если раньше требовалось сохранить в коллекции значение примитивного типа вроде `int`, то его приходилось вручную упаковывать в оболочку данного типа. А когда значение извлекалось из оболочки, нужно было его вручную распаковывать, приводя его явным образом к соответствующему примитивному типу. Упаковка и распаковка осуществляются теперь в Java автоматически, когда требуется сохранять или извлекать данные примитивных типов. Таким образом, выполнять эти операции вручную больше не нужно.

Цикл `for` в стиле `for each`

Все классы в каркасе коллекций усовершенствованы таким образом, чтобы реализовывать интерфейс `Iterable`. Это означает, что содержимое коллекции можно перебрать, организовав цикл `for` в стиле `for each`. Раньше для перебора содержимого коллекции нужно было использовать итератор, рассматриваемый далее в этой главе, организуя цикл вручную. Несмотря на то что итераторы по-прежнему применяются для некоторых целей, во многих случаях циклы на основе итераторов могут быть заменены циклами `for` в стиле `for each`.

Интерфейсы коллекций

В каркасе коллекций определяется несколько интерфейсов. В этом разделе дается краткий обзор каждого из них. Начать рассмотрение коллекций с их интерфейсов следует потому, что они определяют саму сущность классов коллекций. А конкретные классы лишь предоставляют различные реализации стандартных интерфейсов. Интерфейсы, поддерживающие коллекции, перечислены в табл. 18.1.

Таблица 18.1. Интерфейсы, поддерживающие коллекции

Интерфейс	Описание
Collection	Позволяет работать с группами объектов. Находится на вершине иерархии коллекций
Deque	Расширяет интерфейс Queue для организации двусторонних очередей
List	Расширяет интерфейс Collection для управления последовательностями (списками объектов)
NavigableSet	Расширяет интерфейс SortedSet для извлечения элементов по результатам поиска ближайшего совпадения
Queue	Расширяет интерфейс Collection для управления специальными типами списков, где элементы удаляются только из начала списка
Set	Расширяет интерфейс Collection для управления множествами, которые должны содержать однозначные элементы
SortedSet	Расширяет интерфейс Set для управления отсортированными множествами

Помимо перечисленных выше интерфейсов, для составления коллекций используются интерфейсы `Comparator`, `RandomAccess`, `Iterator` и `ListIterator`, которые подробнее рассматриваются далее в этой главе. А начиная с версии JDK 8 к их числу принадлежит также интерфейс `Spliterator`. Если говорить кратко, то интерфейс `Comparator` определяет два сравниваемых объекта, а интерфейсы `Iterator`, `ListIterator` и `Spliterator` перечисляют объекты в коллекции. Если же список реализует интерфейс `RandomAccess`, то тем самым он поддерживает эффективный произвольный доступ к своим элементам.

Ради обеспечения максимальных удобств применения интерфейсов коллекций некоторые методы в них могут быть необязательными. Необязательные методы позволяют видоизменять содержимое коллекций. Коллекции, поддерживающие такие методы, называются *изменяемыми*, а коллекции, не позволяющие изменять свое содержимое, — *неизменяемыми*. Если предпринимается попытка вызвать один из этих методов для неизменяемой коллекции, то генерируется исключение типа `UnsupportedOperationException`. Все встроенные коллекции являются изменяемыми. В последующих разделах подробно рассматриваются интерфейсы коллекций.

Интерфейс `Collection`

Этот интерфейс служит основанием, на котором построен весь каркас коллекций, поскольку он должен быть реализован всеми классами коллекций. Интерфейс `Collection` является обобщенным и объявляется следующим образом:

```
interface Collection<E>
```

где **E** обозначает тип объектов, которые будет содержать коллекция. Интерфейс `Collection` расширяет интерфейс `Iterable`. Это означает, что все коллекции можно перебирать, организовав цикл `for` в стиле `for each`. (Напомним, что только те классы, которые реализуют интерфейс `Iterable`, позволяют перебирать их элементы в цикле `for`.)

В интерфейсе `Collection` определяются основные методы, которые должны иметь все коллекции. Эти методы перечислены в табл. 18.2. В связи с тем что все коллекции реализуют интерфейс `Collection`, знакомство с его методами требуется для ясного понимания каркаса коллекций. Некоторые из этих методов могут генерировать исключение типа `UnsupportedOperationException`. Как пояснялось ранее, это исключение возникает в том случае, если коллекция не может быть изменена. Исключение типа `ClassCastException` генерируется в том случае, если объекты несовместимы, например, при попытке ввести несовместимый объект в коллекцию. Исключение типа `NullPointerException` генерируется при попытке ввести пустое значение `null` в коллекцию, не допускающую наличие пустых элементов. Исключение типа `IllegalArgumentException` генерируется в том случае, если указан неверный аргумент. А исключение типа `IllegalStateException` генерируется при попытке ввести новый элемент в заполненную коллекцию фиксированной длины.

Таблица 18.2. Методы из интерфейса Collection

Метод	Описание
<code>boolean add(E объект)</code>	Вводит заданный <i>объект</i> в вызывающую коллекцию. Возвращает логическое значение true , если <i>объект</i> успешно введен в коллекцию. А если <i>объект</i> уже присутствует в коллекции, которая не допускает дублирование объектов, то возвращается логическое значение false
<code>boolean addAll(Collection<? extends E> c)</code>	Вводит все элементы заданной коллекции <i>c</i> в вызывающую коллекцию. Возвращает логическое значение true , если коллекция изменена (т.е. все элементы введены), а иначе — логическое значение false
<code>void clear()</code>	Удаляет все элементы из вызывающей коллекции
<code>boolean contains(Object объект)</code>	Возвращает логическое значение true , если заданный <i>объект</i> является элементом вызывающей коллекции, а иначе — логическое значение false
<code>boolean containsAll(Collection<?> c)</code>	Возвращает логическое значение true , если вызывающая коллекция содержит все элементы заданной коллекции <i>c</i> , а иначе — логическое значение false
<code>boolean equals(Object объект)</code>	Возвращает логическое значение true , если вызывающая коллекция и заданный <i>объект</i> равнозначны, а иначе — логическое значение false
<code>int hashCode()</code>	Возвращает хеш-код вызывающей коллекции
<code>boolean isEmpty()</code>	Возвращает логическое значение true , если вызывающая коллекция пуста, а иначе — логическое значение false
<code>Iterator<E> iterator()</code>	Возвращает итератор для вызывающей коллекции
<code>default Stream<E> parallelStream()</code>	Возвращает поток, использующий вызывающую коллекцию в качестве источника для ввода-вывода элементов. В этом потоке поддерживаются параллельные операции ввода-вывода, если это вообще возможно (добавлен в версии JDK 8)
<code>boolean remove(Object объект)</code>	Удаляет один экземпляр <i>объекта</i> из вызывающей коллекции. Возвращает логическое значение true , если элемент удален, а иначе — логическое значение false
<code>boolean removeAll(Collection<?> c)</code>	Удаляет все элементы заданной коллекции <i>c</i> из вызывающей коллекции. Возвращает логическое значение true , если в конечном итоге коллекция изменяется (т.е. элементы из нее удалены), а иначе — логическое значение false
<code>default boolean removeIf(Predicate<? super E> предикат)</code>	Удаляет из вызывающей коллекции элементы, удовлетворяющие условию, которое задает <i>предикат</i> (добавлен в версии JDK 8)

Метод	Описание
boolean retainAll(Collection<?> c)	Удаляет из вызывающей коллекции все элементы, кроме элементов заданной коллекции <i>c</i> . Возвращает логическое значение true , если в конечном итоге коллекция изменяется (т.е. элементы из нее удалены), а иначе — логическое значение false
int size()	Возвращает количество элементов, содержащихся в коллекции
default Splitter<E> splitter()	Возвращает итератор-разделитель для вызывающей коллекции (добавлен в версии JDK 8)
default Stream<E> stream()	Возвращает поток, использующий вызывающую коллекцию в качестве источника для ввода-вывода элементов. В этом потоке поддерживаются последовательные операции ввода-вывода (добавлен в версии JDK 8)
Object[] toArray()	Возвращает массив, содержащий все элементы вызывающей коллекции. Элементы массива являются копиями элементов коллекции
<T> T[] toArray(T массив[])	Возвращает массив, содержащий элементы вызывающей коллекции. Элементы массива являются копиями элементов коллекции. Если размер заданного <i>массива</i> равен количеству элементов в коллекции, они возвращаются в этом <i>массиве</i> . Если же размер <i>массива</i> меньше количества элементов в коллекции, то создается и возвращается новый массив нужного размера. А если размер <i>массива</i> больше количества элементов в коллекции, то во всех элементах, следующих за последним из коллекции, устанавливается пустое значение null . И если любой элемент коллекции относится к типу, не являющемуся подтипом <i>массива</i> , то генерируется исключение типа ArrayStoreException

Объекты вводятся в коллекции методом `add()`. Следует, однако, иметь в виду, что метод `add()` принимает аргумент типа *E*. Следовательно, добавляемые в коллекцию объекты должны быть совместимы с предполагаемым типом данных в коллекции. Вызвав метод `addAll()`, можно ввести все содержимое одной коллекции в другую.

Вызвав метод `remove()`, можно удалить из коллекции отдельный объект. Для того чтобы из коллекции удалить группу объектов, достаточно вызвать метод `removeAll()`. А для того чтобы удалить из коллекции все элементы, кроме указанных, следует вызвать метод `retainAll()`. В версии JDK 8 появилась возможность удалить элемент из коллекции, если он удовлетворяет условию, которое задается в качестве параметра *предикат* при вызове метода `removeIf()`. Этот параметр относится к типу функционального интерфейса `Predicate`, внедренного в версии JDK 8, как поясняется в главе 19. И наконец, для очистки коллекции достаточно вызвать метод `clear()`.

Имеется также возможность определить, содержит ли коллекция определенный объект, вызвав метод `contains()`. Чтобы определить, содержит ли одна коллекция все члены другой, следует вызвать метод `containsAll()`. А определить, пуста ли коллекция, можно с помощью метода `isEmpty()`. Количество элементов, содержащихся в данный момент в коллекции, возвращает метод `size()`.

Оба метода `toArray()` возвращают массив, который содержит элементы, хранящиеся в коллекции. Первый из них возвращает массив класса `Object`, а второй — массив элементов того же типа, что и массив, указанный в качестве параметра этого метода. Обычно второй метод более предпочтителен, поскольку он возвращает массив элементов нужного типа. Эти методы оказываются важнее, чем может показаться на первый взгляд. Ведь обрабатывать содержимое коллекции с использованием синтаксиса массивов нередко оказывается очень выгодно. Обеспечив связь коллекции с массивом, можно выгодно воспользоваться преимуществами обоих языковых средств Java.

Две коллекции можно сравнить на равенство, вызвав метод `equals()`. Точный смысл равенства может зависеть от конкретной коллекции. Например, метод `equals()` можно реализовать таким образом, чтобы он сравнивал значения элементов, хранимых в коллекции. В качестве альтернативы методу `equals()` можно сравнивать ссылки на эти элементы.

Еще один очень важный метод `iterator()` возвращает итератор коллекции. Новый метод `spliterator()` возвращает итератор-разделитель для коллекции. Итераторы очень часто используются для обращения с коллекциями. И наконец, методы `stream()` и `parallelStream()` возвращают поток в виде объекта интерфейса `Stream`, использующий коллекцию для ввода-вывода элементов (подробнее новый интерфейс `Stream` рассматривается в главе 29).

Интерфейс List

Этот интерфейс расширяет интерфейс `Collection` и определяет такое поведение коллекций, которое сохраняет последовательность элементов. Элементы могут быть введены или извлечены по индексу их позиции в списке, начиная с нуля. Список может содержать повторяющиеся элементы. Интерфейс `List` является обобщенным и объявляется приведенным ниже образом, где `E` обозначает тип объектов, которые должен содержать список.

```
interface List<E>
```

Помимо методов, объявленных в интерфейсе `Collection`, в интерфейсе `List` определяется ряд своих методов, перечисленных в табл. 18.3. Однако некоторые из этих методов генерируют исключение типа `UnsupportedOperationException`, если коллекция не может быть видоизменена, а исключение типа `ClassCastException` генерируется, если объекты несовместимы, например, при попытке ввести в список элемент несовместимого типа. Кроме того, некоторые методы генерируют исключение типа `IndexOutOfBoundsException`, если указан неверный индекс. А исключение типа `NullPointerException` генерируется при попытке ввести в список пустой объект со значением `null`, когда пустые элементы в данном списке не допускаются. И наконец, исключение типа `IllegalArgumentException` генерируется при указании неверного аргумента.

Таблица 18.3. Методы из интерфейса `List`

Метод	Описание
<code>void add(int индекс, E объект)</code>	Вводит заданный <i>объект</i> на позиции вызывающего списка по указанному <i>индексу</i> . Любые введенные ранее элементы смещаются, начиная с указанной позиции и далее к началу списка. Это означает, что элементы в списке не перезаписываются
<code>boolean addAll(int индекс, Collection<? extends E> c)</code>	Вводит все элементы из коллекции <i>c</i> в вызывающий список, начиная с позиции по указанному <i>индексу</i> . Введенные ранее элементы смещаются, начиная с указанной позиции и далее к началу списка. Это означает, что элементы в списке не перезаписываются. Возвращает логическое значение true , если вызывающий список изменяется, а иначе — логическое значение false
<code>E get(int индекс)</code>	Возвращает объект, хранящийся в вызывающем списке на позиции по указанному <i>индексу</i>
<code>int indexOf(Object объект)</code>	Возвращает индекс первого экземпляра заданного <i>объекта</i> в вызывающем списке. Если заданный <i>объект</i> отсутствует в списке, возвращается значение -1
<code>int lastIndexOf(Object объект)</code>	Возвращает индекс последнего экземпляра заданного <i>объекта</i> в вызывающем списке. Если заданный <i>объект</i> отсутствует в списке, возвращается значение -1
<code>ListIterator<E> listIterator()</code>	Возвращает итератор для обхода элементов с начала вызывающего списка
<code>ListIterator<E> listIterator(int индекс)</code>	Возвращает итератор для обхода элементов вызывающего списка, начиная с позиции по указанному <i>индексу</i>
<code>E remove(int индекс)</code>	Удаляет элемент из вызывающего списка на позиции по указанному <i>индексу</i> и возвращает удаленный элемент. Результирующий список уплотняется, т.е. элементы, следующие за удаленным, смещаются на одну позицию назад
<code>default void replaceAll(UnaryOperator<E> opToApply)</code>	Обновляет каждый элемент списка значением, получаемым из функции, определяемой параметром <i>opToApply</i> (добавлен в версии JDK 8)
<code>E set(int индекс, E объект)</code>	Присваивает заданный <i>объект</i> элементу, находящемуся в списке на позиции по указанному <i>индексу</i> . Возвращает прежнее значение
<code>default void sort(Comparator<? super E> компаратор)</code>	Сортирует список, используя заданный <i>компаратор</i> (добавлен в версии JDK 8)
<code>List<E> subList(int начало, int конец)</code>	Возвращает список, включающий элементы от позиции <i>начало</i> до позиции <i>конец</i> –1 из вызывающего списка. Ссылки на элементы из возвращаемого списка сохраняются и в вызывающем списке

Варианты методов `add()` и `addAll()`, определенные в интерфейсе `Collection`, дополняются в интерфейсе `List` методами `add(int, E)` и `addAll(int, Collection)`. Эти методы вводят элементы на позиции по указанному индексу. Кроме того, семантика методов `add(E)` и `addAll(Collection)`, определенная в интерфейсе `Collection`, изменяется в интерфейсе `List` таким образом, что они вводят элементы в конце списка. Изменить каждый элемент в коллекции можно с помощью метода `replaceAll()`. (Для этой цели в версии JDK 8 внедрен функциональный интерфейс `UnaryOperator`, как поясняется в главе 19.)

Из данного списка можно получить подсписок, вызвав метод `subList()` и указав начальный и конечный индексы подсписка. Нетрудно догадаться, что благодаря методу `subList()` обращаться со списками намного удобнее. Отсортировать список можно, в частности, с помощью метода `sort()`, определенного в интерфейсе `List`.

Интерфейс Set

В интерфейсе `Set` определяется множество. Он расширяет интерфейс `Collection` и определяет поведение коллекций, не допускающих дублирования элементов. Таким образом, метод `add()` возвращает логическое значение `false` при попытке ввести в множество дублирующий элемент. В этом интерфейсе не определяется никаких дополнительных методов. Интерфейс `Set` является обобщенным и объявляется приведенным ниже образом, где `E` обозначает тип объектов, которые должно содержать множество.

```
interface Set<E>
```

Интерфейс SortedSet

Интерфейс `SortedSet` расширяет интерфейс `Set` и определяет поведение множеств, отсортированных в порядке возрастания. Интерфейс `SortedSet` является обобщенным и объявляется приведенным ниже образом, где `E` обозначает тип объектов, которые должно содержать множество.

```
interface SortedSet<E>
```

Помимо методов, предоставляемых интерфейсом `Set`, в интерфейсе `SortedSet` объявляются методы, перечисленные в табл. 18.4. Некоторые из них генерируют исключение типа `NoSuchElementException`, если в вызывающем множестве отсутствуют какие-нибудь элементы. Исключение типа `ClassCastException` генерируется, если заданный объект несовместим с элементами множества. Исключение типа `NullPointerException` генерируется при попытке использовать пустой объект, когда пустое значение `null` в множестве недопустимо. А при указании неверного аргумента генерируется исключение типа `IllegalArgumentException`.

В интерфейсе `SortedSet` определен ряд методов, упрощающих обработку элементов множеств. Чтобы получить первый элемент в отсортированном множестве, достаточно вызвать метод `first()`, а чтобы получить последний элемент — метод `last()`. Из отсортированного множества можно получить подмножество, вызвав метод `subSet()` и указав первый и последний элементы множества. Если

требуется получить подмножество, которое начинается с первого элемента существующего множества, следует вызвать метод `headSet()`. А если требуется получить подмножество, которое начинается с последнего элемента существующего множества, следует вызвать метод `tailSet()`.

Таблица 18.4. Методы из интерфейса `SortedSet`

Метод	Описание
<code>Comparator<? super E> comparator()</code>	Возвращает компаратор отсортированного множества. Если для множества выбирается естественный порядок сортировки, то возвращается пустое значение null
<code>E first()</code>	Возвращает первый элемент вызывающего отсортированного множества
<code>SortedSet<E> headSet(E конец)</code>	Возвращает объект типа SortedSet , содержащий элементы из вызывающего отсортированного множества, которые предшествуют элементу, определяемому параметром <i>конец</i> . Ссылки на элементы из возвращаемого отсортированного множества сохраняются и в вызывающем отсортированном множестве
<code>E last()</code>	Возвращается последний элемент из вызывающего отсортированного множества
<code>SortedSet<E> subSet(E начало, E конец)</code>	Возвращает объект типа SortedSet , который включает в себя элементы, начиная с позиции <i>начало</i> и до позиции <i>конец</i> -1. Ссылки на элементы из возвращаемого отсортированного множества сохраняются и в вызывающем отсортированном множестве
<code>SortedSet<E> tailSet(E начало)</code>	Возвращает объект типа SortedSet , содержащий элементы из вызывающего множества, которые следуют после элемента на заданной позиции <i>начало</i> . Ссылки на элементы из возвращаемого отсортированного множества сохраняются и в вызывающем отсортированном множестве

Интерфейс `NavigableSet`

Этот интерфейс расширяет интерфейс `SortedSet` и определяет поведение коллекции, извлечение элементов из которой осуществляется на основании наиболее точного совпадения с заданным значением или несколькими значениями. Интерфейс `NavigableSet` является обобщенным и объявляется следующим образом:

```
interface NavigableSet<E>
```

где **E** обозначает тип объектов, содержащихся в множестве. Помимо методов, наследуемых из интерфейса `SortedSet`, в интерфейсе `NavigableSet` определяются методы, перечисленные в табл. 18.5. Они генерируют исключение типа `ClassCastException`, если заданный объект несовместим с элементами множества. Исключение типа `NullPointerException` генерируется при попытке ввести пустой объект, когда в множестве не допускаются пустые значения `null`. А при указании неверного аргумента передается исключение типа `IllegalArgumentException`.

Таблица 18.5. Методы из интерфейса **NavigableSet**

Метод	Описание
E ceiling (<i>E объект</i>)	Выполняет поиск в множестве наименьшего элемента <i>e</i> по критерию <i>e</i> \geq <i>объект</i> . Если такой элемент найден, он возвращается, в противном случае — пустое значение null
Iterator < E > descendingIterator ()	Возвращает итератор, выполняющий обход от большего элемента множества к меньшему, т.е. обратный итератор
NavigableSet < E > descendingSet ()	Возвращает объект типа NavigableSet , обратный по отношению к вызывающему множеству. Результирующее множество поддерживается вызывающим множеством
E floor (<i>E объект</i>)	Выполняет поиск в множестве наибольшего элемента <i>e</i> по критерию <i>e</i> \leq <i>объект</i> . Если такой элемент найден, он возвращается, в противном случае — пустое значение null
NavigableSet < E > headSet (<i>E верхняя_граница</i> , boolean <i>включительно</i>)	Возвращает объект типа NavigableSet , содержащий все элементы вызывающего множества, меньше заданной <i>верхней_границы</i> . Если же параметр <i>включительно</i> принимает логическое значение true , то в возвращаемое множество включается и элемент, равный заданной <i>верхней_границе</i> . Результирующее множество поддерживается вызывающим множеством
E higher (<i>E объект</i>)	Выполняет поиск в множестве наибольшего элемента <i>e</i> по критерию <i>e</i> $>$ <i>объект</i> . Если такой элемент найден, он возвращается, а иначе — пустое значение null
E lower (<i>E объект</i>)	Выполняет поиск в множестве наименьшего элемента <i>e</i> по критерию <i>e</i> $<$ <i>объект</i> . Если такой элемент найден, он возвращается, а иначе — пустое значение null
E pollFirst ()	Возвращает первый элемент, попутно удаляя его из множества. Это элемент с наименьшим значением, поскольку множество отсортировано. Если же множество оказывается пустым, то возвращается пустое значение null
E pollLast ()	Возвращает последний элемент, попутно удаляя его из множества. Это элемент с наибольшим значением, поскольку множество отсортировано. Если же множество оказывается пустым, то возвращается пустое значение null
NavigableSet < E > subSet (<i>E нижняя_граница</i> , boolean <i>включая_нижнюю_границу</i> , <i>E верхняя_граница</i> , boolean <i>включая_верхнюю_границу</i>)	Возвращает объект типа NavigableSet , включающий все элементы вызывающего множества, которые больше заданной <i>нижней_границы</i> и меньше заданной <i>верхней_границы</i> . Если же параметр <i>включая_нижнюю_границу</i> принимает логическое значение true , то в результирующее множество включается элемент, равный заданной <i>нижней_границе</i> . А если параметр <i>включая_верхнюю_границу</i> принимает логическое значение true , то в результирующее множество включается и элемент, равный заданной <i>верхней_границе</i> . Результирующее множество поддерживается вызывающим множеством

Метод	Описание
NavigableSet<E> tailSet (<i>E</i> <i>нижняя_граница</i> , <i>boolean</i> <i>включительно</i>)	Возвращает объект типа NavigableSet , включающий все элементы из вызывающего множества, которые больше заданной <i>нижней_границы</i> . Если же параметр <i>включительно</i> принимает логическое значение true , то в результирующее множество включается элемент, равный заданной <i>нижней_границе</i> . Результирующее множество поддерживается вызывающим множеством

Интерфейс Queue

Этот интерфейс расширяет интерфейс **Collection** и определяет поведение очереди, которая действует как список по принципу “первым вошел — первым обслужен”. Тем не менее имеются разные виды очередей, порядок организации в которых основывается на некотором критерии. Интерфейс **Queue** является обобщенным и объявляется следующим образом:

```
interface Queue<E>
```

где **E** обозначает тип объектов, которые будут храниться в очереди. Методы, определенные в интерфейсе **Queue**, перечислены в табл. 18.6.

Таблица 18.6. Методы из интерфейса Queue

Метод	Описание
E element()	Возвращает элемент из головы очереди. Возвращаемый элемент не удаляется. Если очередь пуста, генерируется исключение типа NoSuchElementException
boolean offer (<i>E</i> <i>объект</i>)	Пытается ввести заданный <i>объект</i> в очередь. Возвращает логическое значение true , если <i>объект</i> введен, а иначе — логическое значение false
E peek()	Возвращает элемент из головы очереди. Если очередь пуста, возвращает пустое значение null . Возвращаемый элемент не удаляется из очереди
E poll()	Возвращает элемент из головы очереди и удаляет его. Если очередь пуста, возвращает пустое значение null .
E remove()	Удаляет элемент из головы очереди, возвращая его. Генерирует исключение типа NoSuchElementException , если очередь пуста

Некоторые методы из данного интерфейса генерируют исключение типа **ClassCastException**, если заданный объект несовместим с элементами очереди. Исключение типа **NullPointerException** генерируется, когда предпринимается попытка сохранить пустой объект, а пустые элементы в очереди не разрешены. Исключение типа **IllegalArgumentException** генерируется при указании неверного аргумента. Исключение типа **IllegalStateException** генерируется при попытке ввести объект в заполненную очередь фиксированной длины. И наконец,

исключение типа `NoSuchElementException` генерируется при попытке удалить элемент из пустой очереди.

Несмотря на всю свою простоту, интерфейс `Queue` представляет интерес с нескольких точек зрения. Во-первых, элементы могут удаляться только из начала очереди. Во-вторых, имеются два метода, `poll()` и `remove()`, с помощью которых можно получать и удалять элементы из очереди. Отличаются они тем, что метод `poll()` возвращает пустое значение `null`, если очередь пуста, тогда как метод `remove()` генерирует исключение. И в-третьих, имеются еще два метода, `element()` и `peek()`, которые получают элемент из головы очереди, но не удаляют его. Отличаются они тем, что при пустой очереди метод `element()` генерирует исключение, тогда как метод `peek()` возвращает пустое значение `null`. И наконец, следует иметь в виду, что метод `offer()` только пытается ввести элемент в очередь. А поскольку некоторые очереди имеют фиксированную длину и могут быть заполнены, то вызов метода `offer()` может завершиться неудачно.

Интерфейс `Deque`

Интерфейс `Deque` расширяет интерфейс `Queue` и определяет поведение двусторонней очереди, которая может функционировать как стандартная очередь по принципу “первым вошел — первым обслужен” или как стек по принципу “последним вошел — первым обслужен”. Интерфейс `Deque` является обобщенным и объявляется приведенным ниже образом, где **E** обозначает тип объектов, которые будет содержать двусторонняя очередь.

```
interface Deque<E>
```

Помимо методов, наследуемых из интерфейса `Queue`, в интерфейсе `Deque` определяются методы, перечисленные в табл. 18.7. Некоторые из этих методов генерируют исключение типа `ClassCastException`, если заданный объект несовместим с элементами двусторонней очереди. Исключение типа `NullPointerException` генерируется, когда предпринимается попытка сохранить пустой объект, а пустые элементы двусторонней очереди не допускаются. При указании неверного аргумента генерируется исключение типа `IllegalArgumentException`. Исключение типа `IllegalStateException` генерируется при попытке ввести объект в заполненную двустороннюю очередь фиксированной длины. И наконец, исключение типа `NoSuchElementException` генерируется при попытке удалить элемент из пустой очереди.

Таблица 18.7. Методы из интерфейса `Deque`

Метод	Описание
<code>void addFirst(E объект)</code>	Вводит заданный <i>объект</i> в голову двусторонней очереди. Генерирует исключение типа <code>IllegalStateException</code> , если в очереди фиксированной длины нет свободного места
<code>void addLast(E объект)</code>	Вводит заданный <i>объект</i> в хвост двусторонней очереди. Генерирует исключение типа <code>IllegalStateException</code> , если в очереди фиксированной длины нет свободного места

Метод	Описание
Iterator<E> descendingIterator()	Возвращает итератор для обхода элементов от хвоста к голове двусторонней очереди. Иными словами, возвращает обратный итератор
E getFirst()	Возвращает первый элемент двусторонней очереди. Возвращаемый элемент из очереди не удаляется. Генерирует исключение типа NoSuchElementException , если двусторонняя очередь пуста
E getLast()	Возвращает последний элемент двусторонней очереди. Возвращаемый элемент из очереди не удаляется. Генерирует исключение типа NoSuchElementException , если двусторонняя очередь пуста
boolean offerFirst(E объект)	Пытается ввести заданный <i>объект</i> в голову двусторонней очереди. Возвращает логическое значение true , если <i>объект</i> введен, а иначе — логическое значение false . Таким образом, этот метод возвращает логическое значение false при попытке ввести заданный <i>объект</i> в заполненную двустороннюю очередь фиксированной длины
boolean offerLast(E объект)	Пытается ввести заданный <i>объект</i> в хвост двусторонней очереди. Возвращает логическое значение true , если <i>объект</i> введен, а иначе — логическое значение false
E peekFirst()	Возвращает элемент, находящийся в голове двусторонней очереди. Если очередь пуста, возвращает пустое значение null . Возвращаемый элемент из очереди не удаляется
E peekLast()	Возвращает элемент, находящийся в хвосте двусторонней очереди. Если очередь пуста, возвращает пустое значение null . Возвращаемый элемент из очереди не удаляется
E pollFirst()	Возвращает элемент, находящийся в голове двусторонней очереди, одновременно удаляя его из очереди. Если очередь пуста, возвращает пустое значение null
E pollLast()	Возвращает элемент, находящийся в хвосте двусторонней очереди, одновременно удаляя его из очереди. Если очередь пуста, возвращает пустое значение null
E pop()	Возвращает элемент, находящийся в голове двусторонней очереди, одновременно удаляя его из очереди. Генерирует исключение типа NoSuchElementException , если очередь пуста
void push(E объект)	Вводит заданный <i>объект</i> в голову двусторонней очереди. Если в очереди фиксированной длины нет свободного места, генерирует исключение типа IllegalStateException
E removeFirst()	Возвращает элемент из головы двусторонней очереди, одновременно удаляя его из очереди. Генерирует исключение типа NoSuchElementException , если очередь пуста

Окончание табл. 18.7

Метод	Описание
boolean removeFirstOccurrence (Object <i>объект</i>)	Удаляет первый экземпляр заданного <i>объекта</i> из очереди. Возвращает логическое значение true при удачном исходе операции, а если двусторонняя очередь не содержит заданный <i>объект</i> — логическое значение false
E removeLast()	Возвращает элемент из хвоста двусторонней очереди, одновременно удаляя его из очереди. Генерирует исключение типа NoSuchElementException , если очередь пуста
boolean removeLastOccurrence (Object <i>объект</i>)	Удаляет последний экземпляр заданного <i>объекта</i> из очереди. Возвращает логическое значение true при удачном исходе операции, а если двусторонняя очередь не содержит заданный <i>объект</i> — логическое значение false

Обратите внимание на то, что в состав интерфейса Deque входят методы push() и pop(), благодаря которым этот интерфейс может функционировать как стек. Обратите также внимание на метод descendingIterator(), возвращающий итератор, который обходит элементы очереди в обратном порядке, т.е. от хвоста очереди к ее голове (или конца данного вида коллекции к ее началу). Реализация интерфейса Deque в виде двусторонней очереди может быть *ограниченной по емкости*, т.е. в такую очередь может быть введено ограниченное количество элементов. В этом случае попытка ввести элемент в очередь может оказаться неудачной. Неудачный исход подобных операций интерпретируется в интерфейсе Deque двумя способами. Во-первых, методы вроде addFirst() и addLast() генерируют исключение типа IllegalStateException, если двусторонняя очередь имеет ограниченную емкость. И во-вторых, методы наподобие offerFirst() и offerLast() возвращают логическое значение false, если элемент не может быть введен в очередь.

Классы коллекций

А теперь, когда представлены интерфейсы коллекций, можно приступить к рассмотрению стандартных классов, которые их реализуют. Одни из этих классов предоставляют полную реализацию соответствующих интерфейсов и могут применяться без изменений. А другие являются абстрактными, предоставляя только шаблонные реализации соответствующих интерфейсов, которые используются в качестве отправной точки для создания конкретных коллекций. Как правило, классы коллекций не синхронизированы, но если требуется, то можно получить их синхронизированные варианты, как показано далее в этой главе. Базовые классы коллекций перечислены в табл. 18.8.

В последующих разделах рассматриваются конкретные классы коллекций и демонстрируется их применение.

На заметку! Помимо классов коллекций, некоторые классы, унаследованные из прежних версий, например Vector, Stack и Hashtable, были переделаны для поддержки коллекций. Они также рассматриваются далее в этой главе.

Таблица 18.8. Базовые классы коллекций

Класс	Описание
AbstractCollection	Реализует большую часть интерфейса Collection
AbstractList	Расширяет класс AbstractCollection и реализует большую часть интерфейса List
AbstractQueue	Расширяет класс AbstractCollection и реализует отдельные части интерфейса Queue
AbstractSequentialList	Расширяет класс AbstractList для применения в коллекциях, использующих последовательности вместо случайного доступа к элементам
LinkedList	Реализует связный список, расширяя класс AbstractSequentialList
ArrayList	Реализует динамический массив, расширяя класс AbstractList
ArrayDeque	Реализует динамическую двухстороннюю очередь, расширяя класс AbstractCollection и реализуя интерфейс Deque
AbstractSet	Расширяет класс AbstractCollection и реализует большую часть интерфейса Set
EnumSet	Расширяет класс AbstractSet для применения вместе с элементами типа enum
HashSet	Расширяет класс AbstractSet для применения вместе с хеш-таблицами
LinkedHashSet	Расширяет класс HashSet , разрешая итерацию с вводом элементов в определенном порядке
PriorityQueue	Расширяет класс AbstractQueue для поддержки очередей по приоритетам
TreeSet	Реализует множество, хранимое в древовидной структуре. Расширяет класс AbstractSet

Класс ArrayList

Класс `ArrayList` расширяет класс `AbstractList` и реализует интерфейс `List`. Класс `ArrayList` является обобщенным и объявляется приведенным ниже образом, где параметр `E` обозначает тип сохраняемых объектов.

```
class ArrayList<E>
```

В классе `ArrayList` поддерживаются динамические массивы, которые могут наращиваться по мере надобности. Стандартные массивы в Java имеют фиксированную длину. После того как массив создан, он не может увеличиваться или уменьшаться, а следовательно, нужно заранее знать, сколько элементов требуется в нем хранить. Но иногда еще до стадии выполнения неизвестно, насколько большой

массив потребуется. В качестве выхода из данного положения в каркасе коллекций определяется класс `ArrayList`. По существу, класс `ArrayList` представляет собой списочный массив объектных ссылок переменной длины. Это означает, что размер объекта типа `ArrayList` может динамически увеличиваться или уменьшаться. Списочные массивы создаются с некоторым начальным размером. Когда же этого первоначального размера оказывается недостаточно, коллекция автоматически расширяется. А когда из коллекции удаляются объекты, она может сокращаться.

В классе `ArrayList` определены следующие конструкторы:

```
ArrayList()  
ArrayList(Collection <? extends E> c)  
ArrayList(int емкость)
```

Первый конструктор создает пустой списочный массив, второй — списочный массив, инициализируемый элементами из заданной коллекции `c`, а третий — списочный массив, имеющий начальную *емкость*. Под *емкостью* здесь подразумевается размер базового массива, используемого для хранения элементов данного вида коллекции. Емкость наращивается автоматически по мере ввода элементов в списочный массив.

В следующем примере программы демонстрируется простое применение класса `ArrayList`. В этой программе сначала создается списочный массив объектов типа `String`, затем в него вводится несколько символьных строк. (Напомним, что символьные строки, заключенные в кавычки, преобразуются в объекты типа `String`.) Полученный в итоге список символьных строк выводится на экран. Некоторые элементы удаляются из этого списка, после чего он выводится снова.

```
/// Продемонстрировать применение класса ArrayList  
import java.util.*;  
  
class ArrayListDemo {  
    public static void main (String args[]) {  
        // создать списочный массив  
        ArrayList<String> al = new ArrayList<String>();  
  
        System.out.println(  
            "Начальный размер списочного массива al: " + al.size());  
  
        // ввести элементы в списочный массив  
        al.add("C");  
        al.add("A");  
        al.add("E");  
        al.add("B");  
        al.add("D");  
        al.add("F");  
        al.add(1, "A2");  
        System.out.println(  
            "Размер списочного массива al после ввода элементов: " +  
            al.size());  
  
        // вывести списочный массив  
        System.out.println("Содержимое списочного массива al: " + al);  
  
        // удалить элементы из списочного массива  
        al.remove("F");  
        al.remove(2);
```

```

        System.out.println(
            "Размер списочного массива al после удаления элементов: " +
            al.size());
    }
    System.out.println("Содержимое списочного массива al: " + al);
}

```

Ниже приведен результат, выводимый данной программой. Обратите внимание на то, что списочный массив `al` изначально пуст и увеличивается по мере ввода в него элементов. Когда же элементы удаляются из списочного массива, его размер сокращается.

```

Начальный размер списочного массива al: 0
Размер списочного массива al после ввода элементов: 7
Содержимое списочного массива al: [C, A2, A, E, B, D, F]
Размер списочного массива al после удаления элементов: 5
Содержимое списочного массива al: [C, A2, E, B, D]

```

В приведенном выше примере содержимое коллекции выводится с преобразованием типов, выполняемым по умолчанию методом `toString()`, который наследуется от класса `AbstractCollection`. И хотя этот способ удобен для написания коротких примеров программ, на практике им редко пользуются для вывода содержимого настоящих коллекций. Обычно для этой цели программисты предоставляют свои процедуры вывода. Но для нескольких последующих примеров вполне подходит вывод, выполняемый методом `toString()` по умолчанию.

Несмотря на то что емкость объектов типа `ArrayList` наращивается автоматически, ее можно увеличивать и вручную, вызывая метод `ensureCapacity()`. Это может потребоваться в том случае, если заранее известно, что в коллекции предполагается сохранить намного больше элементов, чем она содержит в данный момент. Увеличив емкость списочного массива в самом начале его обработки, можно избежать дополнительного перераспределения оперативной памяти впоследствии. Ведь перераспределение оперативной памяти — дорогостоящая операция с точки зрения затрат времени, и поэтому исключение лишних операций подобного рода способствует повышению производительности. Ниже приведена общая форма метода `ensureCapacity()`, где параметр *емкость* обозначает новую минимальную емкость коллекции.

```
void ensureCapacity(int емкость)
```

С другой стороны, если требуется уменьшить размер базового массива, на основе которого строится объект типа `ArrayList`, до текущего количества хранящихся в действительности объектов, следует вызвать метод `trimToSize()`. Ниже приведена общая форма этого метода.

```
void trimToSize()
```

Получение массива из коллекции типа `ArrayList`

При обработке списочного массива типа `ArrayList` иногда требуется получить обычный массив, содержащий все элементы списка. Это можно сделать, вызвав метод `toArray()`, определенный в интерфейсе `Collection`.

Имеется несколько причин, по которым возникает потребность преобразовать коллекцию в массив.

- Ускорение выполнения некоторых операций.
- Передача массива в качестве параметра методам, которые не перегружаются, чтобы принимать коллекции непосредственно.
- Интеграция нового кода, основанного на коллекциях, с унаследованным кодом, который не распознает коллекции.

Независимо от конкретной причины преобразовать коллекцию типа `ArrayList` в массив не составляет особого труда. Как пояснялось ранее, имеются два варианта метода `toArray()`, общие формы которых приведены ниже.

```
Object[] toArray()  
<T> T[] toArray(T массив[])
```

В первой форме метод `toArray()` возвращает массив объектов типа `Object`, а во второй форме — массив элементов, относящихся к типу `T`. Обычно вторая форма данного метода удобнее, поскольку в ней возвращается надлежащий тип массива. В следующем примере программы демонстрируется применение именно этой формы метода `toArray()`.

```
// Преобразовать списочный массив ArrayList в обычный массив  
import java.util.*;  
  
class ArrayListToArray {  
    public static void main(String args[]) {  
        // создать списочный массив  
        ArrayList<Integer> al = new ArrayList<Integer>();  
  
        // ввести элементы в списочный массив  
        al.add(1);  
        al.add(2);  
        al.add(3);  
        al.add(4);  
  
        System.out.println("Содержимое списочного массива al: " + al);  
  
        // получить обычный массив  
        Integer ia[] = new Integer[al.size()];  
        ia = al.toArray(ia);  
  
        int sum = 0;  
  
        // суммировать элементы массива  
        for(int i : ia) sum += i;  
  
        System.out.println("Сумма: " + sum);  
    }  
}
```

Данная программа выводит следующий результат:

```
Содержимое списочного массива al: [1, 2, 3, 4]  
Сумма: 10
```

Эта программа начинается с создания коллекции целых чисел. Затем вызывается метод `toArray()` и получается массив элементов типа `Integer`. Далее содержимое массива суммируется в цикле `for` в стиле `for each`. У этой программы имеется еще одна любопытная особенность. Как вам должно быть уже известно, коллекции могут содержать только ссылки, а не значения примитивных типов. Но автоматическая упаковка позволяет передавать методу `add()` значения типа `int`, не прибегая к необходимости заключать их в оболочку типа `Integer`, как это демонстрируется в данной программе. Таким образом, автоматическая упаковка ощутимо облегчает сохранение в коллекциях значений примитивных типов.

Класс `LinkedList`

Этот класс расширяет класс `AbstractSequentialList` и реализует интерфейсы `List`, `Deque` и `Queue`. Он предоставляет структуру данных связанного списка. Класс `LinkedList` является обобщенным и объявляется следующим образом:

```
class LinkedList<E>
```

где **E** обозначает тип сохраняемых в списке объектов. У класса `LinkedList` имеются два конструктора:

```
LinkedList()  
LinkedList(Collection<? extends E> c)
```

Первый конструктор создает пустой связный список, а второй — связный список и инициализирует его содержимым коллекции `c`. В классе `LinkedList` реализуется интерфейс `Deque`, и благодаря этому становятся доступными методы, определенные в интерфейсе `Deque`. Например, чтобы ввести элементы в начале списка, достаточно вызвать метод `addFirst()` или `offerFirst()`, а для того чтобы ввести элементы в конце списка — метод `addLast()` или `offerLast()`. Чтобы получить первый элемент из списка, следует вызвать метод `getFirst()` или `peekFirst()`, а для того чтобы удалить первый элемент из списка — метод `removeFirst()` или `pollFirst()`. И наконец, чтобы получить последний элемент из списка, следует вызвать метод `getLast()` или `peekLast()`, а для того чтобы удалить последний элемент из списка — метод `removeLast()` или `pollLast()`.

В следующем примере программы демонстрируется применение класса `LinkedList`:

```
// Продемонстрировать применение класса LinkedList  
import java.util.*;  
  
class LinkedListDemo {  
    public static void main(String args[]) {  
        // создать связный список  
        LinkedList<String> ll = new LinkedList<String>();  
  
        // ввести элементы в связный список  
        ll.add("F");  
        ll.add("B");  
        ll.add("D");  
        ll.add("E");  
        ll.add("C");  
        ll.addLast("Z");  
    }  
}
```

```

ll.addFirst("A");

ll.add(1, "A2");

System.out.println(
    "Исходное содержимое связанного списка ll: " + ll);

// удалить элементы из связанного списка
ll.remove("F");
ll.remove(2);
System.out.println(
    "Содержимое связанного списка ll " +
    "после удаления элементов: " + ll);
// удалить первый и последний элементы из связанного списка
ll.removeFirst();
ll.removeLast();

System.out.println(
    "Содержимое связанного списка ll после удаления " +
    "первого и последнего элементов: "+ ll);

// получить и присвоить значение
String val = ll.get(2);
ll.set(2, val + " изменено");

System.out.println(
    "Содержимое связанного списка ll после изменения: " + ll);
}
}

```

Ниже приведен результат, выводимый данной программой.

```

Исходное содержимое связанного списка ll: [A, A2, F, B, D, E, C, Z]
Содержимое связанного списка ll после удаления элементов: [A, A2, D, E, C, Z]
Содержимое связанного списка ll после удаления первого и последнего
элементов: [A2, D, E, C]
Содержимое связанного списка ll после изменения: [A2, D, E изменено, C]

```

В классе `LinkedList` реализуется интерфейс `List`, и поэтому в результате вызова метода `add(E)` элементы вводятся в конце списка, как это делается и при вызове метода `addLast()`. Чтобы ввести элементы в определенном месте списка, следует воспользоваться формой метода `add(int, E)`, как продемонстрировано выше на примере вызова `add(1, "A2")`.

Обратите внимание, как третий элемент связанного списка `ll` изменяется с помощью методов `get()` и `set()`. Чтобы получить текущее значение элемента, методу `get()` передается индекс позиции, на которой расположен нужный элемент. А для того чтобы присвоить новое значение элементу на этой позиции, методу `set()` передается соответствующий индекс и новое значение.

Класс `HashSet`

Класс `HashSet` расширяет класс `AbstractSet` и реализует интерфейс `Set`. Он служит для создания коллекции, для хранения элементов которой используется хеш-таблица. Класс `HashSet` является обобщенным и объявляется приведенным ниже образом, где `E` обозначает тип объектов, которые будут храниться в хеш-множестве.

```
class HashSet<E>
```


Как известно, для хранения данных в хеш-таблице применяется механизм так называемого *хеширования*, где содержимое ключа служит для определения однозначного значения, называемого *хеш-кодом*. Этот хеш-код служит далее в качестве индекса, по которому сохраняются данные, связанные с ключом. Преобразование ключа в хеш-код выполняется автоматически, хотя сам хеш-код недоступен. Кроме того, в прикладном коде нельзя индексировать хеш-таблицу непосредственно. Преимущество хеширования заключается в том, что оно обеспечивает постоянство времени выполнения методов `add()`, `contains()`, `remove()` и `size()` — даже для крупных множеств.

В классе `HashSet` определены следующие конструкторы:

```
HashSet()
HashSet(Collection<? extends E> c)
HashSet(int емкость)
HashSet(int емкость, float коэффициент_заполнения)
```

В первой форме конструктора хеш-множество создается по умолчанию. Во второй форме хеш-множество иницируется содержимым заданной коллекции `c`. В третьей форме задается *емкость* хеш-множества (по умолчанию — 16), а в четвертой в качестве аргументов конструктора задается *емкость* хеш-множества и *коэффициент_заполнения*, иначе называемый *емкостью загрузки*. Коэффициент заполнения должен быть в пределах от 0,0 до 1,0, которые определяют, насколько заполненным должно быть хеш-множество, прежде чем будет изменен его размер. В частности, когда количество элементов становится больше емкости хеш-множества, умноженной на коэффициент заполнения, такое хеш-множество расширяется. В конструкторах, которые не принимают коэффициент заполнения в качестве параметра, выбирается значение этого коэффициента, равное 0,75.

В классе `HashSet` не определяется никаких дополнительных методов, помимо тех, что предоставляют его суперклассы и интерфейсы. Следует также иметь в виду, что класс `HashSet` не гарантирует упорядоченности элементов, поскольку процесс хеширования сам по себе обычно не приводит к созданию отсортированных множеств. Если же требуются сортированные множества, то для этой цели лучше выбрать другой вид коллекции, например `TreeSet`. Ниже приведен пример программы, демонстрирующий применение класса `HashSet`.

```
// Продемонстрировать применение класса HashSet
import java.util.*;

class HashSetDemo {
    public static void main(String args[]) {
        // создать хеш-множество
        HashSet<String> hs = new HashSet<String>();

        // ввести элементы в хеш-множество
        hs.add("Бета");
        hs.add("Альфа");
        hs.add("Эта");
        hs.add("Гамма");
        hs.add("Эпсилон");
        hs.add("Омега");

        System.out.println(hs);
    }
}
```

Ниже приведен результат, выводимый данной программой. Как пояснялось ранее, элементы не сохраняются в хеш-множестве в отсортированном порядке, поэтому порядок их вывода может варьироваться.

[Гамма, Эта, Альфа, Эпсилон, Омега, Бета]

Класс `LinkedHashSet`

Класс `LinkedHashSet` расширяет класс `HashSet`, не добавляя никаких новых методов. Этот класс является обобщенным и объявляется следующим образом:

```
class LinkedHashSet<E>
```

где **E** обозначает тип объектов, которые будут храниться в хеш-множестве. У этого класса такие же конструкторы, как и у класса `HashSet`.

В классе `LinkedHashSet` поддерживается связный список элементов хеш-множества в том порядке, в каком они введены в него. Это позволяет организовать итерацию с вводом элементов в определенном порядке. Следовательно, когда перебор элементов хеш-множества типа `LinkedHashSet` производится с помощью итератора, элементы извлекаются из этого множества в том порядке, в каком они были введены. Именно в этом порядке они будут также возвращены методом `toString()`, вызываемым для объекта типа `LinkedHashSet`. Чтобы увидеть эффект от применения класса `LinkedHashSet`, попробуйте подставить его в исходный код предыдущего примера программы вместо класса `HashSet`. После этого выводимый программой результат будет выглядеть так, как показано ниже, отражая тот порядок, в каком элементы были введены в хеш-множество.

[Бета, Альфа, Эта, Гамма, Эпсилон, Омега]

Класс `TreeSet`

Класс `TreeSet` расширяет класс `AbstractSet` и реализует интерфейс `NavigableSet`. Он создает коллекцию, где для хранения элементов применяет древовидная структура. Объекты сохраняются в отсортированном порядке по нарастающей. Время доступа и извлечения элементов достаточно мало, благодаря чему класс `TreeSet` оказывается отличным выбором для хранения больших объемов отсортированных данных, которые должны быть быстро найдены. Класс `TreeSet` является обобщенным классом и объявляется приведенным ниже образом, где **E** обозначает тип объектов, которые будут храниться в древовидном множестве.

```
class TreeSet<E>
```

В классе `TreeSet` определены следующие конструкторы:

```
TreeSet()  
TreeSet(Collection? extends E> c)  
TreeSet(Comparator? super E> компаратор)  
TreeSet(SortedSet<E> ss)
```

В первой форме конструктора создается пустое древовидное множество, элементы которого будут отсортированы в естественном порядке по нарастающей. Во второй форме создается древовидное множество, содержащее элементы заданной кол-

лекции с. В третьей форме создается пустое древовидное множество, элементы которого будут отсортированы заданным *компаратором*. (Компараторы рассматриваются далее в этой главе.) И наконец, в четвертой форме создается древовидное множество, содержащее элементы заданного отсортированного множества *ss*. В приведенном ниже примере программы демонстрируется применение класса `TreeSet`.

```
// Продемонстрировать применение класса TreeSet
import java.util.*;

class TreeSetDemo {
    public static void main(String args[]) {
        // создать древовидное множество типа TreeSet
        TreeSet<String> ts = new TreeSet<String>();

        // ввести элементы в древовидное множество типа TreeSet
        ts.add("C");
        ts.add("A");
        ts.add("B");
        ts.add("E");
        ts.add("F");
        ts.add("D");

        System.out.println(ts);
    }
}
```

Эта программа выводит следующий результат:

```
[A, B, C, D, E, F]
```

Как пояснялось ранее, класс `TreeSet` сохраняет элементы в древовидной структуре. Они автоматически располагаются в отсортированном порядке, что и подтверждает выводимый программой результат. А поскольку класс `TreeSet` реализует интерфейс `NavigableSet`, внедренный в версии Java SE 6, то для извлечения элементов из древовидного множества типа `TreeSet` становятся доступными методы, определенные в интерфейсе `NavigableSet`. Допустим, в исходный код программы из предыдущего примера была добавлена следующая строка кода, где для получения множества *ts*, содержащего элементы от C (включительно) до F (исключительно), сначала вызывается метод `subSet()`, а затем выводится результирующее множество:

```
System.out.println(ts.subSet("C", "F"));
```

Ниже приведен результат выполнения этой строки кода. При желании можете поэкспериментировать с другими методами, определенными в интерфейсе `NavigableSet`.

```
[C, D, E]
```

Класс `PriorityQueue`

Класс `PriorityQueue` расширяет класс `AbstractQueue` и реализует интерфейс `Queue`. Он служит для создания очереди по приоритетам на основании компаратора очереди. Класс `PriorityQueue` является обобщенным и объявляется следующим образом:

```
class PriorityQueue<E>
```

где **E** обозначает тип объектов, которые будут храниться в очереди. Объект типа `PriorityQueue` представляет собой динамическую очередь, которая может при необходимости расширяться.

В классе `PriorityQueue` определяются следующие шесть конструкторов:

```
PriorityQueue()  
PriorityQueue(int емкость)  
PriorityQueue(int емкость, Comparator<? super E> компаратор)  
PriorityQueue(Collection<? extends E> c)  
PriorityQueue(PriorityQueue<? extends E> c)  
PriorityQueue(SortedSet<? extends E> c)
```

Первый конструктор данного класса создает пустую очередь. Ее первоначальная емкость равна 11. Второй конструктор создает очередь с заданной начальной емкостью. Третий конструктор создает очередь заданной емкости с указанным компаратором. Последние три конструктора создают очереди, иницилируемые элементами коллекций, задаваемых в качестве параметра *c*. Но в любом случае по мере ввода элементов в очередь ее емкость автоматически наращивается.

Если при построении очереди типа `PriorityQueue` компаратор не указан, то применяется компаратор, выбираемый по умолчанию для того типа данных, который сохраняется в очереди. Компаратор по умолчанию размещает элементы очереди по нарастающей. Таким образом, в начале (голове) очереди окажется элемент с наименьшим значением. Но, предоставляя свой компаратор, можно задать другую схему сортировки элементов в очереди. Например, когда в очереди сохраняются элементы, содержащие метку времени, для этой очереди можно задать приоритеты таким образом, чтобы самые давние элементы располагались в начале очереди.

Вызвав метод `comparator()` из класса `PriorityQueue`, можно получить ссылку на компаратор, используемый в очереди, как показано ниже.

```
Comparator<? super E> comparator()
```

Этот метод возвращает компаратор. Если в данной очереди применяется естественный порядок сортировки, то возвращается пустое значение `null`. Следует, однако, иметь в виду, что порядок перебора элементов очереди типа `PriorityQueue` не определен, несмотря на то, что их можно перебрать, используя итератор. Чтобы правильно воспользоваться классом `PriorityQueue`, следует вызывать такие методы, как `offer()` и `poll()`, определенные в интерфейсе `Queue`.

Класс `ArrayDeque`

Класс `ArrayDeque` расширяет класс `AbstractCollection` и реализует интерфейс `Deque`. Он не добавляет свои методы. Класс `ArrayDeque` создает динамический массив, не имеющий ограничений по емкости. (Интерфейс `Deque` поддерживает реализации с ограниченной емкостью, но не накладывает на ее величину никаких ограничений.) Класс `ArrayDeque` является обобщенным и объявляется следующим образом:

```
class ArrayDeque<E>
```

где **E** обозначает тип объекта, сохраняемого в коллекции. В классе `ArrayDeque` определяются следующие конструкторы:

```
ArrayDeque()
ArrayDeque(int размер)
ArrayDeque(Collection<? extends E> c)
```

Первый конструктор создает пустую двустороннюю очередь, первоначальная емкостью которой равна 16. Второй конструктор создает двустороннюю очередь указанной емкости. Третий конструктор создает двустороннюю очередь, инициализируемую заданной коллекцией *c*. Но в любом случае емкость увеличивается при вводе новых элементов в двустороннюю очередь по мере надобности.

В приведенном ниже примере программы демонстрируется применение класса `ArrayDeque` для организации стека.

```
// Продемонстрировать применения класса ArrayDeque
import java.util.*;

class ArrayDequeDemo {
    public static void main(String args[]) {
        // создать двухстороннюю очередь
        ArrayDeque<String> adq = new ArrayDeque<String>();

        // использовать класс ArrayDeque для организации стека
        adq.push("A");
        adq.push("B");
        adq.push("D");
        adq.push("E");
        adq.push("F");

        System.out.print("Извлечение из стека: ");

        while(adq.peek() != null)
            System.out.print(adq.pop() + " ");

        System.out.println();
    }
}
```

Эта программа выводит следующий результат:

Извлечение из стека: F E D B A

Класс EnumSet

Класс `EnumSet` расширяет класс `AbstractSet` и реализует интерфейс `Set`. Он служит для создания множества, предназначенного для применения вместе с ключами перечислимого типа `enum`. Это обобщенный класс, объявляемый следующим образом:

```
class EnumSet<E extends Enum<E>>
```

где **E** обозначает элементы перечислимого типа. Следует иметь в виду, что класс **E** должен расширять класс `Enum<E>`, а это требует, чтобы элементы относились к указанному перечислимому типу `enum`.

В классе EnumSet конструкторы не определяются. Вместо этого для создания объектов используются фабричные методы, перечисленные в табл. 18.9. Обратите внимание на неоднократную перегрузку метода of(). Это делается из соображений эффективности. Передать известное количество аргументов, когда оно невелико, можно быстрее, чем делать это с помощью параметра переменной длины.

Таблица 18.9. Методы из класса EnumSet

Метод	Описание
<code>static <E extends Enum<E>> EnumSet<E> allOf(Class<E> t)</code>	Создает множество типа EnumSet , содержащее элементы заданного перечисления <i>t</i>
<code>static <E extends Enum<E>> EnumSet<E> complementOf(EnumSet<E> e)</code>	Создает множество типа EnumSet , дополняющее элементы, отсутствующие в заданном множестве <i>e</i>
<code>static <E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> c)</code>	Создает множество типа EnumSet , содержащее элементы из заданного множества <i>c</i>
<code>static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> c)</code>	Создает множество типа EnumSet , содержащее элементы из заданной коллекции <i>c</i>
<code>static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> t)</code>	Создает множество типа EnumSet , содержащее элементы, которые не входят в заданное перечисление <i>t</i> , которое по определению является пустым множеством
<code>static <E extends Enum<E>> EnumSet<E> of(E v, E ... аргументы переменной длины)</code>	Создает множество типа EnumSet , содержащее элементы <i>v</i> и нуль или дополнительные значения перечислимого типа
<code>static <E extends Enum<E>> EnumSet<E> of(E v)</code>	Создает множество типа EnumSet , содержащее элементы <i>v</i>
<code>static <E extends Enum<E>> EnumSet<E> of(E v1, E v2)</code>	Создает множество типа EnumSet , содержащее элементы <i>v1</i> и <i>v2</i>
<code>static <E extends Enum<E>> EnumSet<E> of(E v1, E v2, E v3)</code>	Создает множество типа EnumSet , содержащее элементы от <i>v1</i> до <i>v3</i>
<code>static <E extends Enum<E>> EnumSet<E> of(E v1, E v2, E v3, E v4)</code>	Создает множество типа EnumSet , содержащее элементы от <i>v1</i> до <i>v4</i>
<code>static <E extends Enum<E>> EnumSet<E> of(E v1, E v2, E v3, E v4, E v5)</code>	Создает множество типа EnumSet , содержащее элементы от <i>v1</i> до <i>v5</i>
<code>static <E extends Enum<E>> EnumSet<E> range(E начало, E конец)</code>	Создает множество типа EnumSet , содержащее элементы в заданных пределах от <i>начала</i> и до <i>конца</i>

Доступ к коллекциям через итератор

Нередко требуется перебрать все элементы коллекции, например, вывести каждый ее элемент. Для этого можно, например, воспользоваться *итератором* — объектом класса, реализующего один из двух интерфейсов: *Iterator* или *ListIterator*. В частности, интерфейс *Iterator* позволяет организовать цикл для перебора коллекции, извлекая или удаляя из нее элементы. А интерфейс

`ListIterator` расширяет интерфейс `Iterator` для двустороннего обхода списка и видоизменения его элементов. Интерфейсы `Iterator` и `ListIterator` являются обобщенными и объявляются следующим образом:

```
interface Iterator<E>
interface ListIterator<E>
```

где **E** обозначает тип перебираемых объектов. В интерфейсе `Iterator` объявляются методы, перечисленные в табл. 18.10. А методы, объявляемые в интерфейсе `ListIterator`, перечислены в табл. 18.11. В обоих случаях операции, видоизменяющие базовую коллекцию, необязательны. Например, метод `remove()` сгенерирует исключение типа `UnsupportedOperationException`, если вызвать его для коллекции, доступной только для чтения. Возможны и другие исключения.

Таблица 18.10. Методы из интерфейса `Iterator`

Метод	Описание
default void forEachRemaining(Consumer<? super E> действие)	Выполняет заданное <i>действие</i> над каждым необработанным элементом коллекции (добавлен в версии JDK 8)
boolean hasNext()	Возвращает логическое значение true , если в коллекции еще имеются элементы, а иначе — логическое значение false
E next()	Возвращает следующий элемент из коллекции. Генерирует исключение типа NoSuchElementException , если в коллекции больше нет элементов
void remove()	Удаляет текущий элемент из коллекции. Генерирует исключение типа IllegalStateException , если предпринимается попытка вызвать метод remove() , которому не предшествовал вызов метода next() . В варианте этого метода по умолчанию генерируется исключение типа UnsupportedOperationException

Таблица 18.11. Методы из интерфейса `ListIterator`

Метод	Описание
void add(E объект)	Вводит заданный <i>объект</i> перед элементом, который должен быть возвращен в результате последующего вызова метода next()
default void forEachRemaining(Consumer<? super E> действие)	Выполняет заданное <i>действие</i> над каждым необработанным элементом коллекции (добавлен в версии JDK 8)
boolean hasNext()	Возвращает логическое значение true , если в списке имеется следующий элемент, а иначе — логическое значение false
boolean hasPrevious()	Возвращает логическое значение true , если в списке имеется предыдущий элемент, а иначе — логическое значение false

Окончание табл. 18.11

Метод	Описание
E next()	Возвращает следующий элемент из списка. Если следующий элемент отсутствует, то генерируется исключение типа NoSuchElementException
int nextIndex()	Возвращает индекс следующего элемента в списке. Если следующий элемент отсутствует, то возвращается длина списка
E previous()	Возвращает предыдущий элемент из списка. Если предыдущий элемент отсутствует, то генерируется исключение типа NoSuchElementException
int previousIndex()	Возвращает индекс предыдущего элемента в списке. Если предыдущий элемент отсутствует, то возвращается значение -1
void remove()	Удаляет текущий элемент из списка. Если метод remove() вызывается до метода next() или previous() , то генерируется исключение типа IllegalStateException
void set (E объект)	Присваивает заданный <i>объект</i> текущему элементу списка. Это элемент, возвращаемый в результате последнего вызова метода next() или previous()

На заметку! В версии JDK 8 появилась возможность для циклического обхода коллекции средствами интерфейса `Spliterator`. Данный интерфейс действует иначе, чем интерфейс `Iterator`, и будет описан далее.

Применение интерфейса `Iterator`

Прежде чем обратиться к коллекции через итератор, следует получить его. В каждом классе коллекций предоставляется метод `iterator()`, возвращающий итератор на начало коллекции. Используя объект итератора, можно получить доступ к каждому элементу коллекции по очереди. В общем, применение итератора для перебора содержимого коллекции сводится к выполнению следующих действий.

- Установить итератор на начало коллекции, получив его из метода `iterator()`, вызываемого для коллекции.
- Организовать цикл, в котором вызывается метод `hasNext()`. Перебирать содержимое коллекции до тех пор, пока метод `hasNext()` не возвратит логическое значение `true`.
- Получить в цикле каждый элемент коллекции, вызывая метод `next()`.

Что касается видов коллекций, реализующих интерфейс `List`, то получить для них итератор можно, вызывая метод `listIterator()`. Как пояснялось ранее, итератор списка обеспечивает доступ к элементам такой коллекции, как в прямом, так и в обратном направлении, а также позволяет видоизменять элементы списка. В остальном интерфейс `ListIterator` применяется таким же образом, как и интерфейс `Iterator`.

В следующем примере программы выполняются все перечисленные выше действия и демонстрируется применение обоих интерфейсов, `Iterator` и `List Iterator`. В данном примере в качестве перебираемой коллекции используется объект типа `ArrayList`, но общие принципы перебора содержимого с помощью итераторов применимы к коллекциям любого вида. Безусловно, интерфейс `ListIterator` доступен только тем видам коллекций, которые реализуют интерфейс `List`.

```
// Продемонстрировать применение итераторов
import java.util.*;

class IteratorDemo {
    public static void main(String args[]) {
        // создать списочный массив
        ArrayList<String> al = new ArrayList<String>();

        // ввести элементы в списочный массив
        al.add("C");
        al.add("A");
        al.add("E");
        al.add("B");
        al.add("D");
        al.add("F");

        // использовать итераторы для вывода содержимого
        // списочного массива al
        System.out.print(
            "Исходное содержимое списочного массива al: ");
        Iterator<String> itr = al.iterator();
        while(itr.hasNext()) {
            String element = itr.next();
            System.out.print(element + " ");
        }
        System.out.println();

        //видоизменить перебираемые объекты
        ListIterator<String> litr = al.listIterator();
        while(litr.hasNext()) {
            String element = litr.next();
            litr.set(element + "+");
        }

        System.out.print(
            "Измененное содержимое списочного массива al: ");
        itr = al.iterator();
        while(itr.hasNext()) {
            String element = itr.next();
            System.out.print(element + " ");
        }
        System.out.println();

        // а теперь отобразить список в обратном порядке
        System.out.print("Измененный в обратном порядке список: ");
        while(litr.hasPrevious()) {
            String element = litr.previous();
            System.out.print(element + " ");
        }
        System.out.println();
    }
}
```

Ниже приведен результат, выводимый данной программой.

Исходное содержимое списочного массива al: C A E B D F

Модифицированное содержимое списочного массива al: C+ A+ E+ B+ D+ F+

Модифицированный в обратном порядке список: F+ D+ B+ E+ A+ C+

Обратите особое внимание на вывод списка в обратном порядке. После видоизменения списка итератор `litr` указывает на конец списка. (Напомним, что метод `litr.hasNext()` возвращает логическое значение `false`, если достигнут конец списка.) Для перебора списка в обратном порядке в данной программе по-прежнему используется итератор `litr`, но на этот раз в ней проверяется, существует ли предшествующий элемент. И до тех пор, пока это делается, выводится каждый элемент, получаемый из списка.

Цикл `for` в стиле `for each` как альтернатива итераторам

Если не требуется видоизменять содержимое коллекции или извлекать из нее элементы в обратном порядке, в таком случае цикл `for` в стиле `for each` может оказаться более удобной альтернативой итераторам. Напомним, что в цикле `for` можно перебирать любую коллекцию объектов, реализующую интерфейс `Iterable`. А поскольку все классы коллекций реализуют этот интерфейс, то ими можно оперировать в цикле `for`.

В следующем примере программы цикл `for` в стиле `for each` используется для суммирования содержимого коллекции:

```
// Применение цикла for в стиле for each
// для перебора элементов коллекции
import java.util.*;

class ForEachDemo {
    public static void main(String args[]) {
        // создать списочный массив для целых чисел
        ArrayList<Integer> vals = new ArrayList<Integer>();

        // ввести числовые значения в списочный массив
        vals.add(1);
        vals.add(2);
        vals.add(3);
        vals.add(4);
        vals.add(5);

        // организовать цикл для вывода числовых значений
        System.out.print(
            "Исходное содержимое списочного массива vals: ");
        for(int v : vals)
            System.out.print(v + " ");

        System.out.println();

        // суммировать числовые значения в цикле for
        int sum = 0;
        for(int v : vals)
            sum += v;

        System.out.println("Сумма числовых значений: " + sum);
    }
}
```

Ниже приведен результат выполнения данной программы.

```
Исходное содержимое списочного массива vals: 1    3 4 5
Сумма числовых значений: 15
```

Как видите, организовать цикл `for` значительно проще и короче, чем использовать итератор. Но он подходит для перебора элементов коллекции только в прямом направлении и не позволяет видоизменять элементы коллекции.

Итераторы-разделители

В версии JDK 8 внедрен новый тип итератора, называемый *итератором-разделителем* и определяемый в интерфейсе `Splititerator`. Итераторы-разделители позволяют перебирать последовательность элементов, и в этом отношении они подобны описанным выше итераторам. Но применяются они иначе. Кроме того, в интерфейсе `Splititerator` предоставляются значительно более широкие функциональные возможности, чем в интерфейсе `Iterator` или `ListIterator`. Вероятно, наиболее важной особенностью интерфейса `Splititerator` является его способность поддерживать параллельную итерацию отдельных частей последовательности элементов, а следовательно, и параллельное программирование, подробнее рассматриваемое в главе 28. Но интерфейс `Splititerator` можно применять и в том случае, когда распараллеливание выполняемых операций не требуется. Одной из причин для такого применения интерфейса `Splititerator` может служить то обстоятельство, что в одном его методе сочетаются операции, выполняемые методами `hasNext()` и `next()` над перебираемыми элементами.

Интерфейс `Splititerator` является обобщенным и объявляется следующим образом:

```
interface Splititerator<T>
```

где `T` обозначает тип перебираемых элементов. В интерфейсе `Splititerator` объявляются методы, перечисленные в табл. 18.12.

Таблица 18.12. Методы из интерфейса `Splititerator`

Метод	Описание
<code>int characteristics()</code>	Возвращает характеристики вызывающего итератора-разделителя, представленные в виде целочисленного значения
<code>long estimateSize()</code>	Оценивает количество элементов, которое осталось перебрать, и возвращает полученный результат. Если это количество нельзя получить по какой-нибудь причине, то возвращается значение константы <code>Long.MAX_VALUE</code>
<code>default void forEachRemaining(Consumer<? super T> deicmaue)</code>	Выполняет заданное <i>deicmaue</i> над каждым необработанным элементом в источнике данных

Окончание табл. 18.12

Метод	Описание
default <code>Comparator<? super T> getComparator()</code>	Возвращает компаратор, используемый вызывающим итератором-разделителем, или пустое значение null , если используется естественное упорядочение. А если последовательность не упорядочена, то генерируется исключение типа IllegalStateException
default <code>long getExactSizeIfKnown()</code>	Если установлен размер вызывающего итератора-разделителя, то возвращается количество элементов, которое осталось перебрать, а иначе — значение -1
default <code>boolean hasCharacteristics(int val)</code>	Возвращает логическое значение true , если у вызывающего итератора-разделителя имеются характеристики, передаваемые в качестве параметра <i>val</i> , а иначе — логическое значение false
<code>boolean tryAdvance(Consumer<? super T> действие)</code>	Выполняет заданное <i>действие</i> над следующим элементом в итерации. Возвращает логическое значение true , если следующий элемент присутствует, а иначе — логическое значение false
<code>Splititerator<T> trySplit()</code>	Разделяет, если это возможно, вызывающий итератор-разделитель, возвращая ссылку на новый итератор-разделитель для последующего разделения, а иначе — пустое значение null . При удачном исходе разделения исходный итератор-разделитель перебирает одну часть последовательности, а возвращаемый итератор-разделитель — остальную ее часть

Интерфейс `Splititerator` применяется для решения основных задач итерации очень просто. Для этого достаточно вызывать метод `tryAdvance()` до тех пор, пока он не возвратит логическое значение **false**. Если же требуется выполнить одно и то же действие над каждым элементом последовательности, то для этой цели имеется более простая альтернатива — вызвать метод `forEachRemaining()`. В обоих случаях действие, происходящее на каждом шаге итерации, определяется тем, что именно объект типа `Consumer` собирается делать с каждым элементом, где `Consumer` — это функциональный интерфейс, выполняющий действие над объектом. Этот обобщенный функциональный интерфейс определен в пакете `java.util.function`, подробно рассматриваемом в главе 19. В функциональном интерфейсе `Consumer` определяется единственный абстрактный метод `accept()`, общая форма которого приведена ниже.

```
void accept(T ссылка_на_объект)
```

Если на каждом шаге итерации вызывается метод `tryAdvance()`, то следующий элемент последовательности передается по ссылке, обозначаемой параметром *ссылка_на_объект*. Зачастую интерфейс `Consumer` проще всего реализовать с помощью лямбда-выражения.

В приведенной ниже программе демонстрируется простой пример применения интерфейса `Splititerator`. В этой программе демонстрируется также применение обоих методов, `tryAdvance()` и `forEachRemaining()`. Обратите внимание на то, что эти методы сочетают в одном своем вызове действия методов `next()` и `hasNext()` из интерфейса `Iterator`.

```
// Продемонстрировать простое применение интерфейса Splititerator
import java.util.*;

class SplititeratorDemo {

    public static void main(String args[]) {
        // создать списочный массив числовых значений типа double
        ArrayList<Double> vals = new ArrayList<>();

        // ввести значения в списочный массив
        vals.add(1.0);
        vals.add(2.0);
        vals.add(3.0);
        vals.add(4.0);
        vals.add(5.0);

        // вызвать метод tryAdvance() для вывода содержимого
        // списочного массива vals
        System.out.print("Содержимое списочного массива vals:\n");
        Splititerator<Double> splitr = vals.splititerator();
        while(splitr.tryAdvance((n) -> System.out.println(n)));
        System.out.println();

        // создать новый списочный массив, содержащий квадратные
        // корни числовых значений из списочного массива vals
        splitr = vals.splititerator();
        ArrayList<Double> sqrs = new ArrayList<>();
        while(splitr.tryAdvance((n) -> sqrs.add(Math.sqrt(n))));

        // вызвать метод forEachRemaining() для вывода содержимого
        // списочного массива sqrs
        System.out.print("Содержимое списочного массива sqrs:\n");
        splitr = sqrs.splititerator();
        splitr.forEachRemaining((n) -> System.out.println(n));
        System.out.println();
    }
}
```

Ниже приведен результат, выводимый данной программой.

Содержимое списочного массива vals:

```
1.0
2.0
3.0
4.0
5.0
```

Содержимое списочного массива sqrs:

```
1.0
1.4142135623730951
1.7320508075688772
2.0
2.23606797749979
```

Несмотря на то что в данной программе демонстрируется механизм применения интерфейса `Splititerator`, она не раскрывает весь его потенциал. Как упомина-

налось выше, наибольшую выгоду применение интерфейса `Splititerator` приносит в тех случаях, когда требуется параллельная обработка.

Обратите внимание на методы `characteristics()` и `hasCharacteristics()`, перечисленные в табл. 18.12. У каждого итератора-разделителя типа `Splititerator` имеются свойства, называемые *характеристиками*. Эти характеристики определяются в статических полях типа `int` интерфейса `Splititerator`, в том числе `SORTED`, `DISTINCT`, `SIZED` и `IMMUTABLE` и прочих полях. Для получения характеристик итератора-разделителя достаточно вызвать метод `characteristics()`, а для выявления отдельной характеристики у итератора-разделителя — метод `hasCharacteristics()`. Зачастую получать характеристики итератора-разделителя не требуется, но иногда они помогают в написании эффективного, устойчивого кода.

На заметку! Рассмотрение интерфейса `Splititerator` будет продолжено в главе 29, где обсуждается его применение в контексте нового прикладного программного интерфейса API потоков данных. Лямбда-выражения рассматриваются в главе 15, а распараллеливание и параллельное программирование — в главе 28.

Несколько подчиненных интерфейсов, вложенных в интерфейс `Splititerator`, предназначены для применения вместе с примитивными типами данных `double`, `int` и `long`. Это интерфейсы `Splititerator.OfDouble`, `Splititerator.OfInt` и `Splititerator.OfLong`. Имеется также обобщенный вариант интерфейса `Splititerator.OfPrimitive`, предоставляющий дополнительные удобства и подчиненный упомянутым выше интерфейсам.

Сохранение объектов пользовательских классов в коллекциях

Ради простоты во всех приведенных ранее примерах программ в коллекциях сохранялись объекты встроенных классов наподобие `String` или `Integer`. Безусловно, сохранение в коллекции не ограничивается только встроенными объектами встроенных классов. Напротив, эффективность коллекций в том и состоит, что в них можно хранить любой тип объектов, включая объекты тех классов, которые вы создаете сами. Рассмотрим в качестве примера следующую программу, где класс `LinkedList` используется для сохранения почтовых адресов:

```
// Простой пример обработки списка почтовых адресов
import java.util.*;
```

```
class Address {
    private String name;
    private String street;
    private String city;
    private String state;
    private String code;
    Address(String n, String s, String c,
            String st, String cd) {
        name = n;
        street = s;
        city = c;
```

```

        state = st;
        code = cd;
    }

    public String toString() {
        return name + "\n" + street + "\n" +
            city + " " + state + " " + code;
    }
}

class MailList {
    public static void main(String args[]) {
        LinkedList<Address> ml = new LinkedList<Address>();

        // ввести элементы в связный список
        ml.add(new Address("J.W. West", "11 Oak Ave",
                           "Urbana", "IL", "61801"));
        ml.add(new Address("Ralph Baker", "1142 Maple Lane",
                           "Mahomet", "IL", "61853"));
        ml.add(new Address("Tom Carlton", "867 Elm St",
                           "Champaign", "IL", "61820"));

        // вывести список почтовых адресов
        for(Address element : ml)
            System.out.println(element + "\n");

        System.out.println();
    }
}

```

Ниже приведен результат, выводимый данной программой.

```

J.W. West
11 Oak Ave
Urbana IL 61801

Ralph Baker
1142 Maple Lane
Mahomet IL 61853

Tom Carlton
867 Elm St
Champaign IL 61820

```

Помимо сохранения объектов пользовательских классов в коллекции, данная программа заслуживает внимания еще и потому, что она довольно короткая. Если принять во внимание, что для сохранения, извлечения и обработки почтовых адресов понадобилось всего около 50 строк кода, то станет очевидной эффективность каркаса коллекций. Ведь если запрограммировать все эти функциональные возможности вручную, то программа окажется в несколько раз длиннее. Коллекции предлагают готовые решения для широкого круга задач программирования. Их следует использовать всякий раз, когда позволяет ситуация.

Интерфейс RandomAccess

Этот интерфейс не содержит ни одного члена. Но, реализуя его, коллекция извещает о том, что она поддерживает эффективный произвольный доступ к своим

элементам. Даже если в коллекции и поддерживается произвольный доступ к ее элементам, такой доступ может оказаться недостаточно эффективным. Проверяя интерфейс `RandomAccess` во время выполнения, в прикладном коде можно выяснить, допускает ли конкретная коллекция некоторые виды операций произвольного доступа, и, в частности, насколько они применимы к крупным коллекциям. (Чтобы определить, реализует ли класс коллекции интерфейс `RandomAccess`, можно воспользоваться оператором `instanceof`.) Интерфейс `RandomAccess` реализуется в классе `ArrayList` и, среди прочего, в унаследованном классе `Vector`.

Обращение с отображениями

Отображение представляет собой объект, сохраняющий связи между ключами и значениями в виде *пар* “ключ–значение”. По заданному ключу можно найти его значение. Ключи и значения являются объектами. Ключи могут быть однозначными, а значения — дублированными. В одних отображениях допускаются пустые ключи и пустые значения, а в других — они не допускаются.

В отношении отображений необходимо иметь в виду следующее: они не реализуют интерфейс `Iterable`. Это означает, что перебрать содержимое отображения, организовав цикл `for` в стиле `for each`, не удастся. Более того, нельзя получить итератор отображения. Но, как будет показано ниже, можно получить представление отображения в виде коллекции, которое допускает перебор содержимого в цикле или с помощью итератора.

Интерфейсы отображений

Интерфейсы отображений определяют их характер и особенности, поэтому начать рассмотрение отображений следует с их интерфейсов. Отображения поддерживаются в интерфейсах, перечисленных в табл. 18.13. Каждый из этих интерфейсов рассматривается далее по отдельности.

Таблица 18.13. Интерфейсы, поддерживающие отображения

Интерфейс	Описание
<code>Map</code>	Отображает однозначные ключи на значения
<code>Map.Entry</code>	Описывает элемент отображения (пару “ключ–значение”). Это внутренний класс интерфейса <code>Map</code>
<code>NavigableMap</code>	Расширяет интерфейс <code>SortedMap</code> для извлечения элементов из отображения по критерию поиска наиболее точного совпадения
<code>SortedMap</code>	Расширяет интерфейс <code>Map</code> таким образом, чтобы ключи располагались по нарастающей

Интерфейс `Map`

Интерфейс `Map` отображает однозначные ключи на значения. *Ключ* — это объект, используемый для последующего извлечения данных. Задавая ключ и значение, можно размещать значение в отображении, представленном объектом типа

Map. Сохранив значение по ключу, можно получить его обратно по этому же ключу. Интерфейс Map является обобщенным и объявляется приведенным ниже образом, где *K* обозначает тип ключей, а *V* — тип хранимых в отображении значений.

```
interface Map<K, V>
```

Методы, объявляемые в интерфейсе Map, перечислены в табл. 18.14. Некоторые из них генерируют исключение типа `ClassCastException`, если заданный объект несовместим с элементами отображения. Исключение типа `NullPointerException` генерируется при попытке использовать пустой объект, когда данное отображение этого не допускает. А исключение типа `UnsupportedOperationException` генерируется при попытке изменить неизменяемое отображение.

Таблица 18.14. Методы из интерфейса Map

Метод	Описание
<code>default V compute(K k, BiFunction<? super K, ? super V, ? extends V> функция)</code>	Вызывает заданную <i>функцию</i> для построения нового значения. Если <i>функция</i> возвращает непустое значение, то в отображение вводится новая пара “ключ–значение”, удаляется любая ранее существовавшая пара и возвращается новое значение. А если <i>функция</i> возвращает пустое значение <code>null</code> , то удаляется любая ранее существовавшая пара и возвращается пустое значение <code>null</code> (добавлен в версии JDK 8)
<code>default V computeIfAbsent(K k, Function<? super K, ? extends V> функция)</code>	Возвращает значение, связанное с указанным ключом <i>k</i> . В противном случае создается новое значение, для чего вызывается заданная <i>функция</i> , в отображение вводится новая пара “ключ–значение” и возвращается созданное значение. Если же новое значение создать нельзя, то возвращается пустое значение <code>null</code> (добавлен в версии JDK 8)
<code>default V computeIfPresent(K k, BiFunction<? super K, ? super V, ? extends V> функция)</code>	Если в отображении присутствует указанный ключ <i>k</i> , то для создания нового значения вызывается заданная <i>функция</i> и новое значение заменяет прежнее в отображении. В этом случае возвращается новое значение. Если же заданная <i>функция</i> возвращает пустое значение <code>null</code> , то из отображения удаляются существующие в нем ключ и значение и затем возвращается пустое значение <code>null</code> (добавлен в версии JDK 8)
<code>void clear()</code>	Удаляет все пары “ключ–значение” из вызывающего отображения
<code>boolean containsKey(Object k)</code>	Возвращает логическое значение <code>true</code> , если вызывающее отображение содержит указанный ключ <i>k</i> , а иначе — логическое значение <code>false</code>
<code>boolean containsValue(Object v)</code>	Возвращает логическое значение <code>true</code> , если вызывающее отображение содержит значение <i>v</i> , а иначе — логическое значение <code>false</code>

Продолжение табл. 18.14

Метод	Описание
Set<Map.Entry<K, V>> entrySet()	Возвращает множество типа Set , содержащее все записи из вызывающего отображения в виде объектов типа Map.Entry . Следовательно, этот метод возвращает представление вызывающего отображения в виде множества
boolean equals(Object объект)	Возвращает логическое значение true , если заданный <i>объект</i> является отображением типа Map , содержащим одинаковые значения, а иначе — логическое значение false
default void forEach(BiConsumer<? super K, ? super V> действие)	Выполняет заданное <i>действие</i> над каждым элементом вызывающего отображения. Если в ходе этого процесса удаляется элемент, то генерируется исключение типа ConcurrentModificationException (добавлен в версии JDK 8)
V get(Object k)	Возвращает значение, связанное с указанным ключом <i>k</i> . Если же ключ не найден, то возвращается пустое значение null
default V getOrDefault(Object k, V заданное_значение)	Возвращает значение, связанное с указанным ключом <i>k</i> , если оно присутствует в вызывающем отображении, а иначе — <i>заданное_значение</i> (добавлен в версии JDK 8)
int hashCode()	Возвращает хеш-код вызывающего отображения
boolean isEmpty()	Возвращает логическое значение true , если вызывающее отображение пусто, а иначе — логическое значение false
Set<K> keySet()	Возвращает множество, содержащее ключи из вызывающего отображения. Следовательно, этот метод возвращает представление ключей в вызывающем отображении в виде множества
default V merge(K k, V v, BiFunction<? super V, ? super V, ? extends V> функция)	Если в вызывающем отображении отсутствует указанный ключ <i>k</i> , то в него вводится пара “ключ–значение”, определяемая параметрами <i>k</i> , <i>v</i> , а затем возвращается значение <i>v</i> . В противном случае заданная <i>функция</i> возвращает новое значение, исходя из прежнего значения и ключ обновляется для доступа к этому значению, а затем оно возвращается из метода merge() . Если же заданная <i>функция</i> возвращает пустое значение null , то ключ и значения, существующие в вызывающем отображении, удаляются из него и затем возвращается пустое значение null (добавлен в версии JDK 8)
V put(K k, V v)	Вводит новое значение <i>v</i> в вызывающее отображение, перезаписывая любое предшествующее значение, связанное с заданным ключом <i>k</i> . Возвращает пустое значение null , если ключ ранее не существовал. В противном случае возвращается предыдущее значение, связанное с ключом

Метод	Описание
<code>void putAll (Map<? extends K, ? extends V> m)</code>	Вводит все записи из заданного отображения <i>m</i> в вызывающее отображение
<code>default V putIfAbsent (K k, V v)</code>	Вводит пару “ключ–значение”, определяемую параметрами <i>k</i> , <i>v</i> , в вызывающее отображение, если она отсутствует в нем или же если значение, связанное с заданным ключом <i>k</i> , оказывается пустым. В этом случае возвращает пустое значение null , а иначе – прежнее значение (добавлен в версии JDK 8)
<code>V remove (Object k)</code>	Удаляет запись по заданному ключу <i>k</i>
<code>default boolean remove (Object k, Object v)</code>	Если пара “ключ–значение”, определяемая параметрами <i>k</i> , <i>v</i> , присутствует в вызывающем отображении, то она удаляется и затем возвращается логическое значение true , а иначе – логическое значение false (добавлен в версии JDK 8)
<code>default boolean replace (K k, V прежнее_значение, V новое_значение)</code>	Если пара “ключ–значение”, определяемая параметрами <i>v</i> и <i>прежнее_значение</i> , присутствует в вызывающем отображении, то это значение заменяется на <i>новое_значение</i> и затем возвращается логическое значение true , а иначе – логическое значение false (добавлен в версии JDK 8)
<code>default V replace (K k, V v)</code>	Если в вызывающем отображении присутствует заданный ключ <i>k</i> , то значение по этому ключу заменяется новым значением <i>v</i> и возвращается прежнее значение, а иначе – пустое значение null (добавлен в версии JDK 8)
<code>default void replaceAll (BiFunction<? super K, ? super V, ? extends V> функция)</code>	Выполняет заданную <i>функцию</i> для каждого элемента в вызывающем отображении, заменяя элемент результатом, возвращаемым заданной <i>функцией</i> . Если в ходе этого процесса удаляется элемент, то генерируется исключение типа ConcurrentModificationException (добавлен в версии JDK 8)
<code>int size ()</code>	Возвращает количество пар “ключ–значение” в вызывающем отображении
<code>Collection<V> values ()</code>	Возвращает коллекцию, содержащую значения из вызывающего отображения. Следовательно, этот метод возвращает представление значений в вызывающем отображении в виде коллекции

Обращение с отображениями опирается на две основные операции, выполняемые методами `get ()` и `put ()`. Чтобы ввести значение в отображение, следует вызвать метод `put ()`, указав ключ и значение, а для того чтобы получить значение из отображения – вызвать метод `get ()`, передав ему ключ в качестве аргумента. По этому ключу будет возвращено связанное с ним значение.

Как упоминалось ранее, отображения не реализуют интерфейс `Collection`, хотя являются частью каркаса коллекций. Тем не менее можно получить представ-

ление отображения в виде коллекции. Для этого можно воспользоваться методом `entrySet()`, возвращающим множество, содержащее элементы отображения. Чтобы получить представление ключей в отображении в виде коллекции, следует вызвать метод `keySet()`, а для того чтобы получить представление значений в отображении в виде коллекции — метод `values()`. Все три представления отображений и их элементов в виде коллекций относятся к тем средствам, с помощью которых отображения интегрируются в крупный каркас коллекций.

Интерфейс `SortedMap`

Этот интерфейс расширяет интерфейс `Map`. Он обеспечивает размещение записей в отображении по порядку нарастания ключей. Интерфейс `SortedMap` является обобщенным и объявляется приведенным ниже образом, где **K** обозначает тип ключей, а **V** — тип хранимых в отображении значений.

```
interface SortedMap<K, V>
```

Методы, объявляемые в интерфейсе `SortedMap`, перечислены в табл. 18.15. Некоторые из них генерируют исключение типа `NoSuchElementException`, если вызывающее отображение пусто. Исключение типа `ClassCastException` генерируется в том случае, если заданный объект несовместим с элементами, хранящимися в отображении. Исключение типа `NullPointerException` генерируется при попытке использовать пустой объект, когда пустые объекты в данном отображении не допускаются. А исключение типа `IllegalArgumentException` генерируется в том случае, если указан неверный аргумент.

Таблица 18.15. Методы из интерфейса `SortedMap`

Метод	Значение
<code>Comparator<? super K> comparator()</code>	Возвращает компаратор вызывающего отсортированного отображения. Если в отображении применяется естественное упорядочение элементов, то возвращается пустое значение <code>null</code>
<code>K firstKey()</code>	Возвращает первый ключ из вызывающего отображения
<code>SortedMap<K, V> headMap(K конец)</code>	Возвращает отсортированное отображение, содержащее те элементы из вызывающего отображения, ключи которых меньше, чем указанный <i>конец</i>
<code>K lastKey()</code>	Возвращает последний ключ в вызывающем отображении
<code>SortedMap<K, V> subMap(K начало, K конец)</code>	Возвращает отображение, содержащее элементы вызывающего отображения, ключ которых больше, чем <i>начало</i> , или равен ему и меньше, чем <i>конец</i>
<code>SortedMap<K, V> tailMap(K начало)</code>	Возвращает отсортированное отображение, содержащее те элементы вызывающего отображения, ключ которых больше, чем указанное <i>начало</i>

Отсортированные отображения обеспечивают очень эффективное манипулирование *подотображениями* (иными словами, подмножествами отображений). Для получения подотображений служат методы `headMap()`, `tailMap()` или `subMap()`. Подотображение, возвращаемое этими методами, поддерживается вызывающим

отображением. При изменении одного изменяется другое. Для получения первого ключа из подмножества отображения следует вызвать метод `firstKey()`, а для получения последнего ключа — метод `lastKey()`.

Интерфейс `NavigableMap`

Интерфейс `NavigableMap` расширяет интерфейс `SortedMap` и определяет поведение отображения, поддерживающего извлечение записей из него по наиболее точному совпадению с заданным ключом или несколькими ключами. Интерфейс `NavigableMap` является обобщенным и объявляется следующим образом:

```
interface NavigableMap<K, V>
```

где **K** обозначает тип ключей, а **V** — тип значений, связанных с ключами. Помимо методов, наследуемых из интерфейса `SortedMap`, в интерфейс `NavigableMap` введены методы, перечисленные в табл. 18.16. Некоторые из них генерируют исключение типа `ClassCastException`, если объект несовместим с ключами отображения. Исключение типа `NullPointerException` генерируется при попытке использовать пустой объект, когда пустые ключи в отображении не допускаются. А исключение типа `IllegalArgumentException` передается при указании неверного аргумента.

Таблица 18.16. Методы из интерфейса `NavigableMap`

Метод	Значение
<code>Map.Entry<K, V></code> <code>ceilingEntry(K объект)</code>	Выполняет поиск в отображении наименьшего ключа <i>k</i> по критерию <i>k</i> \geq <i>объект</i> . Если такой ключ найден, то возвращается запись по этому ключу, а иначе — пустое значение null
K <code>ceilingKey(K объект)</code>	Выполняет поиск в отображении наименьшего ключа <i>k</i> по критерию <i>k</i> \geq <i>объект</i> . Если такой ключ найден, то он возвращается, а иначе — пустое значение null
<code>NavigableSet<K></code> <code>descendingKeySet()</code>	Возвращает множество типа <code>NavigableSet</code> , содержащее ключи в вызывающем отображении в обратном порядке. Следовательно, этот метод возвращает обратное представление ключей в отображении в виде множества. Результирующее множество опирается на вызывающее отображение
<code>NavigableMap<K, V></code> <code>descendingMap()</code>	Возвращает множество типа <code>NavigableSet</code> , обратное вызывающему отображению. Результирующее множество опирается на вызывающее отображение
<code>Map.Entry<K, V></code> <code>firstEntry()</code>	Возвращает первую запись в отображении. Это запись с наименьшим ключом
<code>Map.Entry<K, V></code> <code>floorEntry(K объект)</code>	Выполняет поиск в отображении наибольшего ключа <i>k</i> по критерию <i>k</i> \leq <i>объект</i> . Если такой ключ найден, то возвращается запись по этому ключу, а иначе — пустое значение null
K <code>floorKey(K объект)</code>	Выполняет поиск в отображении наибольшего ключа <i>k</i> по критерию <i>k</i> \leq <i>объект</i> . Если такой ключ найден, то он возвращается, а иначе — пустое значение null

Продолжение табл. 18.16

Метод	Значение
NavigableMap<K, V> headMap (K <i>верхняя_граница</i> , boolean <i>включительно</i>)	Возвращает множество типа NavigableSet , содержащее все записи из вызывающего отображения по ключам меньше, чем заданная <i>верхняя_граница</i> . Если параметр <i>включительно</i> принимает логическое значение true , то в результирующее множество включается элемент, равный заданной <i>верхней_границе</i> . Результирующее множество опирается на вызывающее отображение
Map.Entry<K, V> higherEntry (K <i>объект</i>)	Выполняет поиск в отображении наибольшего ключа <i>k</i> по критерию <i>k</i> > <i>объект</i> . Если такой ключ найден, то возвращается запись по этому ключу, а иначе — пустое значение null
K higherKey (K <i>объект</i>)	Выполняет поиск в отображении наибольшего ключа <i>k</i> по критерию <i>k</i> > <i>объект</i> . Если такой ключ найден, то он возвращается, а иначе — пустое значение null
Map.Entry<K, V> lastEntry ()	Возвращает последнюю запись в отображении. Это запись с наибольшим ключом
Map.Entry<K, V> lowerEntry (K <i>объект</i>)	Выполняет поиск в отображении наибольшего ключа <i>k</i> по критерию <i>k</i> < <i>объект</i> . Если такой ключ найден, то возвращается запись по этому ключу, а иначе — пустое значение null
K lowerKey (K <i>объект</i>)	Выполняет поиск в отображении наибольшего ключа <i>k</i> по критерию <i>k</i> < <i>объект</i> . Если такой ключ найден, то он возвращается, а иначе — пустое значение null
NavigableSet<K> navigableKeySet ()	Возвращает множество типа NavigableSet , содержащее ключи из вызывающего отображения. Результирующее множество опирается на вызывающее отображение
Map.Entry<K, V> pollFirstEntry ()	Возвращает первую запись в отображении, попутно удаляя ее. Это запись по наименьшему ключу, поскольку отображение отсортировано. Если же отображение оказывается пустым, то возвращается пустое значение null
Map.Entry<K, V> pollLastEntry ()	Возвращает последнюю запись в отображении, попутно удаляя ее. Это запись по наибольшему ключу, поскольку отображение отсортировано. Если же отображение оказывается пустым, то возвращается пустое значение null
NavigableMap<K, V> subMap (K <i>нижняя_граница</i> , boolean <i>включая_нижнюю_границу</i> , K <i>верхняя_граница</i> , boolean <i>включая_верхнюю_границу</i>)	Возвращает отображение типа NavigableSet , содержащее все записи из вызывающего отображения по ключам меньше, чем <i>верхняя_граница</i> , и больше, чем <i>нижняя_граница</i> . Если параметр <i>включая_нижнюю_границу</i> принимает логическое значение true , то в результирующее отображение включается элемент, равный заданной <i>нижней_границе</i> . А если параметр <i>включая_нижнюю_границу</i> принимает логическое значение true , то в результирующее отображение включается элемент, равный заданной <i>верхней_границе</i> . Результирующее отображение опирается на вызывающее отображение

Метод	Значение
<code>NavigableMap<K, V></code> <code>tailMap</code> (<i>K</i> <i>нижняя_граница</i> , <i>boolean</i> <i>включительно</i>)	Возвращает отображение типа <code>NavigableSet</code> , содержащее все записи из вызывающего отображения по ключам больше, чем заданная <i>нижняя_граница</i> . Если параметр <i>включительно</i> принимает логическое значение <code>true</code> , то в результирующее отображение включается элемент, равный заданной <i>нижней_границе</i> . Результирующее отображение опирается на вызывающее отображение

Интерфейс `Map.Entry`

Этот интерфейс позволяет обращаться с отдельными записями в отображении. Напомним, что метод `entrySet()`, объявляемый в интерфейсе `Map`, возвращает множество типа `Set`, содержащее записи из отображения. Каждый элемент этого множества представляет собой объект типа `Map.Entry`. Интерфейс `Map.Entry` является обобщенным и объявляется следующим образом:

```
interface Map.Entry<K, V>
```

где *K* обозначает тип ключей, а *V* — тип хранимых в отображении значений. В табл. 18.17 перечислены нестатические методы, объявляемые в интерфейсе `Map.Entry`. В версии JDK 8 в интерфейс введены два статических метода. Первый из них называется `comparingByKey()` и возвращает компаратор типа `Comparator`, сравнивающий записи в отображении по заданному ключу. А второй называется `comparingByValue()` и возвращает компаратор типа `Comparator`, сравнивающий записи в отображении по указанному значению.

Таблица 18.17. Методы из интерфейса `Map.Entry`

Метод	Значение
<code>boolean</code> <code>equals</code> (<i>Object</i> <i>объект</i>)	Возвращает логическое значение <code>true</code> , если заданный <i>объект</i> представляет запись из отображения типа <code>Map.Entry</code> , ключ и значение в которой такие же, как и у вызывающего объекта
<code>K</code> <code>getKey</code> ()	Возвращает ключ данной записи из отображения
<code>V</code> <code>getValue</code> ()	Возвращает значение данной записи из отображения
<code>int</code> <code>hashCode</code> ()	Возвращает хеш-код данной записи из отображения
<code>V</code> <code>setValue</code> (<i>V</i> <i>v</i>)	Устанавливает указанное значение <i>v</i> в данной записи из отображения. Если значение <i>v</i> не относится к типу, допустимому для данного отображения, то генерируется исключение типа <code>ClassCastException</code> . Если же значение <i>v</i> указано неверно, то генерируется исключение типа <code>IllegalArgumentException</code> . А если значение <i>v</i> оказывается пустым (<code>null</code>) и в отображении нельзя хранить пустые ключи, то генерируется исключение типа <code>NullPointerException</code> . И наконец, если в отображение нельзя вносить изменения, то генерируется исключение типа <code>UnsupportedOperationException</code>

Классы отображений

Интерфейсы отображений реализуются в нескольких классах. Классы, которые могут быть использованы для отображений, перечислены в табл. 18.18. Следует иметь в виду, что класс `AbstractMap` служит суперклассом для всех конкретных реализаций отображений.

Класс `WeakHashMap` реализует отображение, в котором используются так называемые “слабые ключи”, что позволяет собирать в “мусор” запись из отображения как ненужный объект, когда ее ключ больше не используется. Этот класс подробно здесь не обсуждается, а прочие классы отображений описываются далее.

Таблица 18.18. Классы отображений

Класс	Описание
<code>AbstractMap</code>	Реализует большую часть интерфейса <code>Map</code>
<code>EnumMap</code>	Расширяет класс <code>AbstractMap</code> для применения вместе с ключами типа <code>enum</code>
<code>HashMap</code>	Расширяет класс <code>AbstractMap</code> для применения хеш-таблицы
<code>TreeMap</code>	Расширяет класс <code>AbstractMap</code> для применения древовидной структуры
<code>WeakHashMap</code>	Расширяет класс <code>AbstractMap</code> для применения хеш-таблицы со слабыми ключами
<code>LinkedHashMap</code>	Расширяет класс <code>HashMap</code> , разрешая итерацию с вводом элементов в определенном порядке
<code>IdentityHashMap</code>	Расширяет класс <code>AbstractMap</code> и использует результаты проверки ссылок на равенство при сравнении документов

Класс `HashMap`

Этот класс расширяет класс `AbstractMap` и реализует интерфейс `Map`. В нем используется хеш-таблица для хранения отображения, и благодаря этому обеспечивается постоянное время выполнения методов `get()` и `put()` даже в обращении к крупным отображениям. Класс `HashMap` является обобщенным и объявляется приведенным ниже образом, где *K* обозначает тип ключей, а *V* — тип хранимых в отображении значений.

```
class HashMap<K, V>
```

В классе определены следующие конструкторы:

```
HashMap()
HashMap(Map<? extends K, ? extends V> m)
HashMap(int емкость)
HashMap(int емкость, float коэффициент_заполнения)
```

В первой форме конструктора создается хеш-отображение по умолчанию. Во второй форме конструктора хеш-отображение инициализируется элементами заданного отображения *m*. В третьей форме задается *емкость* хеш-отображения. И в четвертой форме *емкость* и *коэффициент_заполнения* хеш-отображения задаются в качестве аргументов конструктора. Назначение емкости и коэффициен-

та заполнения такое же, как и в описанном ранее классе `HashSet`. По умолчанию емкость составляет 16, а коэффициент заполнения — 0,75.

Класс `HashMap` реализует интерфейс `Map` и расширяет класс `AbstractMap`, не дополняя их своими методами. Следует иметь в виду, что хеш-отображение не гарантирует порядок расположения своих элементов. Следовательно, порядок, в котором элементы вводятся в хеш-отображение, не обязательно соответствует тому порядку, в котором они извлекаются итератором. В следующем примере программы демонстрируется применение класса `HashMap`. В этой программе имена вкладчиков отображаются на остатки на их банковских счетах. Обратите внимание, каким образом получается и используется представление хеш-отображения в виде множества:

```
import java.util.*;

class HashMapDemo {
    public static void main(String args[]) {
        // создать хеш-отображение
        HashMap<String, Double> hm = new HashMap<String, Double>();

        // ввести элементы в хеш-отображение
        hm.put("Джон Доу", new Double(3434.34));
        hm.put("Том Смит", new Double(123.22));
        hm.put("Джейн Бейкер", new Double(1378.00));
        hm.put("Тод Холл", new Double(99.22));
        hm.put("Ральф Смит", new Double(-19.08));

        // получить множество записей
        Set<Map.Entry<String, Double>> set = hm.entrySet();

        // вывести множество записей
        for(Map.Entry<String, Double> me : set) {
            System.out.print(me.getKey() + ": ");
            System.out.println(me.getValue());
        }
        System.out.println();

        // внести сумму 1000 на счет Джона Доу
        double balance = hm.get("Джон Доу");
        hm.put("Джон Доу", balance + 1000);
        System.out.println("Новый остаток на счете Джона Доу: " +
            hm.get("Джон Доу"));
    }
}
```

Ниже приведен результат, выводимый данной программой (точный порядок следования записей может отличаться).

```
Ральф Смит: -19.08
Том Смит: 123.22
Джон Доу: 3434.34
Тод Холл: 99.22
Джейн Бейкер: 1378.0
Новый остаток на счете Джона Доу: 4434.34
```

Выполнение данной программы начинается с создания хеш-отображения, в которое затем вводятся имена и фамилии вкладчиков, отображаемые на остатках на их банковских счетах. Далее содержимое хеш-отображения выводится

с помощью его представления в виде множества, получаемого из метода `entrySet()`. Ключи и значения выводятся в результате вызова методов `getKey()` и `getValue()`, определенных в интерфейсе `Map.Entry`. Обратите особое внимание на порядок внесения суммы на счет Джона Доу. Метод `put()` автоматически заменяет новым значением любое существовавшее ранее значение, связанное с указанным ключом. Таким образом, после обновления остатка на счете Джона Доу хеш-отображение по-прежнему содержит только один счет Джона Доу.

Класс `TreeMap`

Класс `TreeMap` расширяет класс `AbstractMap` и реализует интерфейс `NavigableMap`. В нем создается отображение, размещаемое в древовидной структуре. В классе `TreeMap` предоставляются эффективные средства для хранения пар “ключ–значение” в отсортированном порядке и обеспечивается их быстрое извлечение. Следует заметить, что, в отличие от хеш-отображения, древовидное отображение гарантирует, что его элементы будут отсортированы по порядку нарастания ключей. Класс `TreeMap` является обобщенным и объявляется следующим образом:

```
class TreeMap<K, V>
```

где `K` обозначает тип ключей, а `V` — тип хранимых в отображении значений. В классе `TreeMap` определены следующие конструкторы:

```
TreeMap()  
TreeMap(Comparator<? super K> компаратор)  
TreeMap(Map<? extends K, ? extends V> m)  
TreeMap(SortedMap<K, ? extends V> sm)
```

В первой форме конструктора создается пустое древовидное отображение, которое будет отсортировано с естественным упорядочением ключей. Во второй форме конструктора создается пустое древовидное отображение, которое будет отсортировано с помощью заданного *компаратора* типа `Comparator`. (Компараторы обсуждаются далее в этой главе.) В третьей форме древовидное отображение инициализируется элементами из отображения `m`, которые будут отсортированы с естественным упорядочением ключей. И наконец, в четвертой форме создается древовидное отображение с элементами из отображения `sm`, которые будут отсортированы в том же порядке, что и в отображении `sm`.

В классе `TreeMap` не определяются дополнительные методы, помимо тех, что имеются в интерфейсе `NavigableMap` и классе `AbstractMap` для обращения с отображениями. Ниже приведен вариант предыдущего примера программы, переделанный с целью продемонстрировать применение класса `TreeMap`.

```
import java.util.*;  
  
class TreeMapDemo {  
    public static void main(String args[]) {  
        // создать древовидное отображение  
        TreeMap<String, Double> tm = new TreeMap<String, Double>();  
  
        // ввести элементы в древовидное отображение  
        tm.put("Джон Доу", new Double(3434.34));  
        tm.put("Том Смит", new Double(123.22));  
    }  
}
```

```

tm.put("Джейн Бейкер", new Double(1378.00));
tm.put("Тод Халл", new Double(99.22));
tm.put("Ральф Смит", new Double(-19.08));

// получить множество записей
Set<Map.Entry<String, Double>> set = tm.entrySet();

// вывести множество записей
for(Map.Entry<String, Double> me : set) {
    System.out.print(me.getKey() + ": ");
    System.out.println(me.getValue());
}
System.out.println();

// внести сумму 1000 на счет Джона Доу
double balance = tm.get("Джон Доу");
tm.put("Джон Доу", balance + 1000);
System.out.println("Новый остаток на счете Джона Доу: " +
tm.get("Джон Доу"));
}
}

```

Ниже приведен результат, выводимый данной программой.

```

Джейн Бейкер: 1378.0
Джон Доу: 3434.34
Ральф Смит: -19.08
Тод Халл: 99.22
Том Смит: 123.22

```

```

Новый остаток на счете Джона Доу: 4434.34

```

Обратите внимание на то, что класс `TreeMap` сортирует ключи. Но в данном случае они сортируются по имени вместо фамилии. Такое поведение можно изменить, указав компаратор при создании отображения, как поясняется далее.

Класс `LinkedHashMap`

Класс `LinkedHashMap` расширяет класс `HashMap`. Он создает связный список элементов, располагаемых в отображении в том порядке, в котором они вводились в него. Это позволяет организовать итерацию с вводом элементов в отображение в определенном порядке. Следовательно, при итерации представления отображения типа `LinkedHashMap` в виде коллекции его элементы будут возвращаться в том порядке, в котором они вводились в него. Можно также создать отображение типа `LinkedHashMap`, возвращающее свои элементы в том порядке, в котором к ним осуществлялся доступ в последний раз. Класс `LinkedHashMap` является обобщенным и объявляется приведенным ниже образом, где **K** обозначает тип ключей, а **V** — тип хранимых в отображении значений.

```
class LinkedHashMap<K, V>
```

В классе `LinkedHashMap` определяются следующие конструкторы:

```

LinkedHashMap()
LinkedHashMap(Map<? extends K, ? extends V> m)
LinkedHashMap(int емкость)
LinkedHashMap(int емкость, float коэффициент_заполнения)
LinkedHashMap(int емкость, float коэффициент_заполнения, boolean порядок)

```

В первой форме конструктора создается отображение типа `LinkedHashMap` по умолчанию. Во второй форме конструктора отображение типа `LinkedHashMap` инициализируется элементами из заданного отображения *m*. В третьей форме задается *емкость* отображения, в четвертой — *емкость* и *коэффициент заполнения* отображения. Назначение этих параметров такое же, как и у класса `HashMap`. По умолчанию емкость составляет 16, а коэффициент заполнения — 0,75. В последней форме конструктора можно указать *порядок* расположения элементов в связанном списке: ввода или последнего доступа. Если параметр *порядок* принимает логическое значение `true`, то используется порядок доступа, а если он принимает логическое значение `false` — порядок ввода элементов.

В классе `LinkedHashMap` добавляется только один новый метод к тем, что определены в классе `HashMap`. Это метод `removeEldestEntry()`, общая форма которого приведена ниже.

```
protected boolean removeEldestEntry(Map.Entry<K, V> e)
```

Этот метод вызывается из методов `put()` и `putAll()`. Самая старая запись в отображении передается в качестве параметра *e*. По умолчанию этот метод возвращает логическое значение `false` и ничего не делает. Но если переопределить его, то можно удалить самую старую запись из отображения типа `LinkedHashMap`. Для этого переопределенный метод должен вернуть логическое значение `true`. А для того чтобы сохранить самую старую запись в отображении, из переопределенного метода следует вернуть логическое значение `false`.

Класс `IdentityHashMap`

Класс `IdentityHashMap` расширяет класс `AbstractMap` и реализует интерфейс `Map`. Он аналогичен классу `HashMap`, за исключением того, что при сравнении элементов отображения в нем выполняется проверка ссылок на равенство. Класс `IdentityHashMap` является обобщенным и объявляется следующим образом:

```
class IdentityHashMap<K, V>
```

где *K* обозначает тип ключей, а *V* — тип хранимых в отображении значений. В документации на прикладной программный интерфейс API ясно сказано, что класс `IdentityHashMap` не предназначен для общего применения.

Класс `EnumMap`

Класс `EnumMap` расширяет класс `AbstractMap` и реализует интерфейс `Map`. Он специально предназначен для применения вместе с ключами типа `enum`. Это обобщенный класс, объявляемый следующим образом:

```
class EnumMap<K extends Enum<K>, V>
```

где *K* обозначает тип ключей, а *V* — тип хранимых в отображении значений. Следует иметь в виду, что класс *K* должен расширять класс `Enum<K>`, а для этого ключи должны быть типа `enum`.

В классе `EnumMap` определены следующие конструкторы:

```
EnumMap(Class<K> тип_ключа)
EnumMap(Map<K, ? extends V> m)
EnumMap(EnumMap<K, ? extends V> em)
```

Первый конструктор создает пустое отображение типа EnumMap для хранения элементов, имеющих заданный *тип_ключа*. Второй конструктор создает отображение типа EnumMap с теми же записями, что и в заданном отображении *m*. А третий конструктор создает отображение типа EnumMap, инициализируемое значениями из отображения *em*. Свои методы в классе EnumMap не определяются.

Компараторы

Классы TreeSet и TreeMap сохраняют элементы в отсортированном порядке. Однако понятие “порядок сортировки” точно определяет применяемый ими компаратор. По умолчанию эти классы сохраняют элементы, используя то, что в Java называется *естественным упорядочением*, т.е. ожидаемым упорядочением, когда после A следует B, а после 1 — 2 и т.д. Если же элементы требуется упорядочить иным образом, то при создании множества или отображения следует указать компаратор типа Comparator. Это дает возможно точно управлять порядком сохранения элементов в отсортированных коллекциях.

Интерфейс Comparator является обобщенным и объявляется приведенным ниже образом, где T обозначает тип сравниваемых объектов.

```
interface Comparator<T>
```

До версии JDK 8 в интерфейсе Comparator определялись только два метода: compare() и equals(). Метод compare(), общая форма которого приведена ниже, сравнивает два элемента по порядку.

```
int compare(T объект1, T объект2)
```

Здесь параметры *объект1* и *объект2* обозначают сравниваемые объекты. Обычно этот метод возвращает нулевое значение, если объекты равны; положительное значение, если *объект1* больше, чем *объект2*, а иначе — отрицательное значение. Этот метод может сгенерировать исключение типа ClassCastException, если типы сравниваемых объектов несовместимы. Реализуя метод compare(), можно изменить порядок расположения объектов. Например, чтобы отсортировать объекты в обратном порядке, можно создать компаратор, который обращает результат их сравнения.

Метод equals(), общая форма которого приведена ниже, проверяет объект на равенство вызывающему компаратору.

```
boolean equals(Object объект)
```

Здесь параметр *объект* обозначает проверяемый на равенство объект. Метод equals() возвращает логическое значение true, если заданный *объект* и вызывающий объект относятся к типу Comparator и упорядочиваются одним и тем же способом. В противном случае этот метод возвращает логическое значение false. Переопределение метода equals() не требуется, и большинство простых компараторов в этом не нуждается.

Многие годы в интерфейсе `Comparator` были доступны только оба упомянутых выше метода. Но после выпуска версии JDK 8 это положение коренным образом изменилось в лучшую сторону. В версии JDK 8 функциональные возможности интерфейса `Comparator` были значительно расширены благодаря внедрению методов по умолчанию и статических методов, рассматриваемых ниже по отдельности.

Используя метод `reversed()`, можно получить компаратор, изменяющий на обратное упорядочение сравниваемых объектов, с которым этот компаратор вызывался. Ниже приведена общая форма данного метода.

```
default Comparator<T> reversed()
```

Этот метод возвращает компаратор с обратным упорядочением. Так, если в исходном компараторе используется естественное упорядочение символов от **A** до **Z**, то компаратор с обратным упорядочением расположит букву **B** перед **A**, букву **C** перед **B** и т.д. С методом `reversed()` тесно связан метод `reverseOrder()`, общая форма которого выглядит следующим образом:

```
static <T extends Comparable<? super T>> Comparator<T> reverseOrder()
```

Этот метод возвращает компаратор, изменяющий на обратное естественное упорядочение сравниваемых элементов. С другой стороны, можно получить компаратор с естественным упорядочением сравниваемых элементов, вызвав статический метод `naturalOrder()`. Ниже приведена его общая форма.

```
static <T extends Comparable<? super T>> Comparator<T> naturalOrder()
```

Если же требуется компаратор, способный обрабатывать пустые значения `null`, то для этой цели служит метод `nullsFirst()` или `nullsLast()`. Ниже приведены общие формы этих методов.

```
static <T> Comparator<T> nullsFirst(Comparator<? super T> компаратор)
static <T> Comparator<T> nullsLast(Comparator<? super T> компаратор)
```

Метод `nullsFirst()` возвращает компаратор, рассматривающий пустые значения `null` как меньшие остальных значений. А метод `nullsLast()` возвращает компаратор, рассматривающий пустые значения `null` как большие остальных значений. Но в любом случае заданный *компаратор* выполняет сравнение, если оба сравниваемых значения не являются пустыми. Если же заданному *компаратору* передается пустое значение `null`, то все непустые значения рассматриваются им как равнозначные.

В версии JDK 8 в интерфейсе `Comparator` появился еще один метод по умолчанию, выполняющий второе сравнение, если результат первого сравнения указывает на равенство сравниваемых объектов. Следовательно, с помощью этого метода можно составить последовательность “сравнить сначала по *X*, а затем по *Y*”. Например, при сравнении городов можно сначала сравнивать их названия, а затем названия штатов. Так, в естественном алфавитном порядке название Спрингфилд, Иллинойс, будет предшествовать названию Спрингфилд, Миссури. У метода `thenComparing()` имеются три общие формы объявления. Ниже приведена первая общая форма, позволяющая указать второй компаратор, передав экземпляр объекта типа `Comparator`. В этой форме параметр *второй компаратор* обозначает компаратор, вызываемый в том случае, если первый компаратор возвращает признак равенства сравниваемых объектов.

```
default Comparator<T> thenComparing(Comparator<? super T> второй_компаратор)
```

В двух других формах метода `thenComparing()` можно указать стандартный функциональный интерфейс `Function`, определенный в пакете `java.util.function`. Обе эти формы приведены ниже.

```
default <U extends Comparable<? super U> Comparator<T>
    thenComparing(Function<? super T, ? extends U> получить_ключ)
default <U> Comparator<T>
    thenComparing(Function<? super T, ? extends U> получить_ключ,
        Comparator<? super U> компаратор_ключей)
```

В обеих формах параметр *получить_ключ* обозначает функцию, получающую следующий ключ для сравнения. Этот ключ используется в том случае, если в результате первого сравнения возвращается признак равенства сравниваемых объектов. В последней форме данного метода параметр *компаратор_ключей* обозначает компаратор, используемый для сравнения ключей. (Здесь и далее `U` обозначает тип ключа.)

Интерфейс `Comparator` дополнен также специальными вариантами методов последующего сравнения примитивных типов данных. Их общие формы приведены ниже, где параметр *получить_ключ* обозначает функцию, получающую следующий ключ для сравнения.

```
default Comparator<T>
    thenComparingDouble(ToDoubleFunction<? super T> получить_ключ)
default Comparator<T>
    thenComparingInt(ToIntFunction<? super T> получить_ключ)
default Comparator<T>
    thenComparingLong(ToLongFunction<? super T> получить_ключ)
```

И наконец, в версии JDK 8 интерфейс `Comparator` дополнен методом `comparing()`. Этот метод возвращает компаратор, получающий ключ для сравнения от функции, передаваемой данному методу в качестве параметра. Ниже приведены обе формы объявления метода `comparing()`.

```
static <T, U extends Comparable<? super U>> Comparator<T>
    comparing(Function<? super T, ? extends U> получить_ключ)
static <T, U> Comparator<T>
    comparing(Function<? super T, ? extends U> получить_ключ,
        Comparator<? super U> компаратор_ключей)
```

В обеих формах параметр *получить_ключ* обозначает функцию, получающую следующий ключ для сравнения. Во второй форме данного метода параметр *компаратор_ключей* обозначает компаратор, используемый для сравнения ключей. Интерфейс `Comparator` дополнен также специальными вариантами метода `comparing()` для сравнения примитивных типов данных. Их общие формы приведены ниже, где параметр *получить_ключ* обозначает функцию, получающую следующий ключ для сравнения.

```
static <T> Comparator<T>
    ComparingDouble(ToDoubleFunction<? super T> получить_ключ)
static <T> Comparator<T>
    ComparingInt(ToIntFunction<? super T> получить_ключ)
static <T> Comparator<T>
    ComparingLong(ToLongFunction<? super T> получить_ключ)
```

Применение компараторов

Ниже приведен пример программы, демонстрирующий эффективность специальных компараторов. В этой программе реализуется метод `compare()` для сравнения символьных строк в порядке, обратном обычному. Это означает, что элементы древовидного множества сортируются в обратном порядке.

```
// Использовать специальный компаратор
import java.util.*;

// Компаратор для сравнения символьных строк в обратном порядке
class MyComp implements Comparator<String> {
    public int compare(String a, String b) {
        String aStr, bStr;
        aStr = a;
        bStr = b;
        // выполнить сравнение в обратном порядке
        return bStr.compareTo(aStr);
    }
    // переопределять метод equals() не требуется
}

class CompDemo {
    public static void main(String args[]) {
        // создать древовидное множество типа TreeSet
        TreeSet<String> ts = new TreeSet<String>(new MyComp());
        // ввести элементы в древовидное множество
        ts.add("C");
        ts.add("A");
        ts.add("B");
        ts.add("E");
        ts.add("F");
        ts.add("D");

        // вывести элементы из древовидного множества
        for(String element : ts)
            System.out.print(element + " ");
        System.out.println();
    }
}
```

Как показывает приведенный ниже результат выполнения данной программы, древовидное множество теперь отсортировано в обратном порядке.

F E D C B A

Обратите внимание на класс `MyComp`, реализующий интерфейс `Comparator` и метод `compare()`. Как упоминалось выше, переопределение метода `equals()` не требуется и вообще не принято. Переопределение не требуется и методам по умолчанию, внедренным в интерфейсы, начиная с версии JDK 8.) В теле метода `compare()` для объекта типа `String` вызывается метод `compareTo()`, сравнивающий две символьные строки. Но метод `compareTo()` вызывается для строкового объекта `bStr`, а не `aStr`. Благодаря этому результат сравнения получается обратным.

Несмотря на то что способ реализации компаратора с обратным упорядочением в предыдущем примере программы вполне пригоден, в версии JDK 8 появился еще один способ сделать то же самое — просто вызвать метод `reversed()` для ком-

паратора с естественным упорядочением. Этот метод возвратит эквивалентный компаратор, который, однако, сравнивает объекты в обратном порядке. Так, предыдущий пример программы можно переделать, внося следующие изменения в класс `MyClass`:

```
class MyComp implements Comparator<String> {
    public int compare(String aStr, String bStr)
        return aStr.compareTo(bStr);
}
```

Далее можно воспользоваться следующим фрагментом кода для создания древовидного множества типа `TreeSet`, где строчные элементы располагаются в обратном порядке:

```
MyComp mc = new MyComp(); // создать компаратор

// передать вариант компаратора типа MyComp с обратным
// упорядочением древовидному множеству типа TreeSet
TreeSet<String> ts = new TreeSet<String>(mc.reversed());
```

Если ввести этот новый код в предыдущий пример программы, то будет получен тот же самый результат ее выполнения. В данном примере применение метода `reversed()` не дает никаких преимуществ, но в тех случаях, когда требуется создать компараторы с естественным и обратным упорядочением, метод `reversed()` упрощает получение компаратора с обратным упорядочением, не требуя написания дополнительного кода специально для этой цели.

Начиная с версии JDK 8 создавать класс `MyComp`, как показано в предыдущих примерах, фактически не требуется, поскольку его можно легко заменить соответствующим лямбда-выражением. В частности, класс `MyComp` можно полностью удалить, а вместо него создать компаратор символьных строк, используя следующий фрагмент кода:

```
// использовать лямбда-выражение для реализации
// компаратора типа Comparator<String>
Comparator<String> mc = (aStr, bStr) -> aStr.compareTo(bStr);
```

Следует также заметить, что в данном простом примере компаратор с обратным упорядочением можно задать и с помощью лямбда-выражения непосредственно в вызове конструктора класса `TreeSet`, как показано ниже.

```
// передать компаратор с обратным упорядочением конструктору
// класса TreeSet через лямбда-выражение
TreeSet<String> ts = new TreeSet<String>(
    (aStr, bStr) -> bStr.compareTo(aStr));
```

Внося эти изменения, можно значительно сократить исходный код упомянутой выше программы, как показывает приведенный ниже окончательный ее вариант.

```
// Использовать лямбда-выражение для создания компаратора
// с обратным упорядочением
import java.util.*;

class CompDemo2 {
    public static void main(String args[]) {

        // передать компаратор с обратным упорядочением
```

```
// древовидному множеству типа TreeSet
TreeSet<String> ts = new TreeSet<String>(
    (aStr, bStr) -> bStr.compareTo(aStr));

// ввести элементы в древовидное множество
ts.add("C");
ts.add("A");
ts.add("B");
ts.add("E");
ts.add("F");
ts.add("D");
// вывести элементы из древовидного множества
for(String element : ts)
    System.out.print(element + " ");

System.out.println();
}
}
```

Рассмотрим более полезное применение компараторов на примере переделанного варианта представленной ранее программы, где имена и фамилии вкладчиков и остатки на их банковских счетах сохраняются в древовидном отображении типа TreeMap. В предыдущем варианте этой программы счета сортировались по имени каждого вкладчика, а в новом, приведенном ниже ее варианте — по его фамилии. Для этого в ней используется компаратор, сравнивающий фамилии каждого вкладчика. В итоге получается отображение, отсортированное по фамилиям вкладчиков.

```
// Использовать компаратор для сортировки счетов
// по фамилиям вкладчиков
import java.util.*;

// сравнить последние слова в обеих символьных строках
class TComp implements Comparator<String> {
    public int compare(String aStr, String bStr) {
        int i, j, k;

        // найти индекс символа, с которого начинается фамилия
        i = aStr.lastIndexOf(' ');
        j = bStr.lastIndexOf(' ');
        k = aStr.substring(i).compareTo(bStr.substring(j));
        if(k==0) // фамилии совпадают, проверить имя и фамилию полностью
            return aStr.compareTo(bStr);
        else
            return k;
    }
    // переопределять метод equals() не требуется
}

class TreeMapDemo2 {
    public static void main(String args[]) {
        // создать древовидное отображение
        TreeMap<String, Double> tm =
            new TreeMap<String, Double>(new TComp());

        // ввести элементы в древовидное отображение
        tm.put("Джон Доу", new Double(3434.34));
        tm.put("Том Смит", new Double(123.22));
        tm.put("Джейн Бейкер", new Double(1378.00));
    }
}
```

```

tm.put("Тод Халл", new Double(99.22));
tm.put("Ральф Смит", new Double(-19.08));

// получить множество элементов
Set<Map.Entry<String, Double>> set = tm.entrySet();

// вывести элементы из множества
for (Map.Entry<String, Double> me : set) {
    System.out.print(me.getKey() + ": ");
    System.out.println(me.getValue());
}

System.out.println();

// внести сумму 1000 на счет Джона Доу
double balance = tm.get("Джон Доу");
tm.put("Джон Доу ", balance + 1000);
System.out.println("Новый остаток на счете Джона Доу: " +
    tm.get("Джон Доу"));
}
}

```

Ниже приведен результат, выводимый данной программой. Обратите внимание на то, что счета теперь отсортированы по фамилиям вкладчиков.

```

Джейн Бейкер: 1378.0
Джон Доу: 3434.34
Ральф Смит: -19.08
Том Смит: 123.22
Тод Халл: 99.22

```

Новый остаток на счете Джона Доу: 4434.34

Компаратор типа `TComp` сравнивает две символьные строки, содержащие имя и фамилию. Сначала он сравнивает фамилии. Для этого осуществляется поиск позиции последнего пробела в каждой строке с последующим сравнением подстроки, начинающейся с этой позиции. В том случае, если фамилии одинаковы, сравниваются имена. В итоге получается древовидное отображение, отсортированное по фамилиям вкладчиков, а в пределах одинаковых фамилий — по именам. Можете убедиться в этом сами, поскольку имя и фамилия Ральф Смит в приведенном выше результате выполнения данной программы располагаются выше имени и фамилии Том Смит.

Начиная с версии JDK 8 предыдущий пример программы можно переделать таким образом, чтобы отсортировать древовидное отображение сначала по фамилии, а затем по имени вкладчика, используя метод `thenComparing()`. Напомним, что метод `thenComparing()` позволяет указать второй компаратор, который используется в том случае, если вызывающий компаратор возвращает признак равенства сравниваемых объектов. Именно такой способ сортировки счетов вкладчиков реализован в приведенном ниже переделанном варианте предыдущего примера программы.

```

// Использовать метод thenComparing() для сортировки
// счетов вкладчиков сначала по фамилии, а затем по имени
import java.util.*;

// Компаратор, сравнивающий фамилии вкладчиков

```

```

class CompLastNames implements Comparator<String> {
    public int compare(String aStr, String bStr) {
        int i, j;

        // найти индекс символа, с которого начинается фамилия
        i = aStr.lastIndexOf(' ');
        j = bStr.lastIndexOf(' ');
        return aStr.substring(i).compareToIgnoreCase(bStr.substring(j));
    }
}

// отсортировать счета вкладчиков по Ф.И.О., если фамилии одинаковы
class CompThenByFirstName implements Comparator<String> {
    public int compare(String aStr, String bStr) {
        int i, j;

        return aStr.compareToIgnoreCase(bStr);
    }
}

class TreeMapDemo2A {
    public static void main(String args[]) {
        // использовать метод thenComparing() для создания
        // компаратора, сравнивающего сначала фамилии, а затем
        // Ф.И.О. вкладчиков, если фамилии одинаковы
        CompLastNames compLN = new CompLastNames();
        Comparator<String> compLastThenFirst =
            compLN.thenComparing(new CompThenByFirstName());

        // создать древовидное отображение
        TreeMap<String, Double> tm =
            new TreeMap<String, Double>(compLastThenFirst);
        // ввести элементы в древовидное отображение
        tm.put("Джон Доу", new Double(3434.34));
        tm.put("Том Смит", new Double(123.22));
        tm.put("Джейн Бейкер", new Double(1378.00));
        tm.put("Тод Халл", new Double(99.22));
        tm.put("Ральф Смит", new Double(-19.08));

        // получить множество элементов
        Set<Map.Entry<String, Double>> set = tm.entrySet();

        // вывести элементы из множества
        for (Map.Entry<String, Double> me : set) {
            System.out.print(me.getKey() + ": ");
            System.out.println(me.getValue());
        }
        System.out.println();

        // внести сумму 1000 на счет Джона Доу
        double balance = tm.get("Джон Доу");
        tm.put("Джон Доу", balance + 1000);

        System.out.println("Новый остаток на счете Джона Доу: " +
            tm.get("Джон Доу"));
    }
}

```

Результат выполнения данного варианта рассматриваемой здесь программы такой же, как и предыдущего ее варианта. Их отличие лишь в том, как реализуется сортировка счетов вкладчиков. Прежде всего обратите внимание на создание компаратора

типа `CompLastNames`. Этот компаратор сравнивает только фамилии вкладчиков, тогда как второй компаратор типа `CompThenByFirstName` — Ф.И.О. вкладчиков, начиная с имени. Ниже показано, каким образом создаются оба эти компаратора.

```
CompLastNames compLN = new CompLastNames();
Comparator<String> compLastThenFirst =
    compLN.thenComparing(new CompThenByFirstName());
```

Здесь первый компаратор присваивается переменной `compLN` в виде экземпляра класса `CompLastNames`. Для него вызывается метод `thenComparing()`, которому в качестве параметра передается экземпляр класса `CompThenByFirstName`. Полученный результат присваивается переменной `compLastThenFirst`. Этот второй компаратор используется для построения древовидного множества типа `TreeMap`, как показано ниже.

```
TreeMap<String, Double> tm =
    new TreeMap<String, Double>(compLastThenFirst);
```

Теперь сортировка счетов вкладчиков выполняется по Ф.И.О., если их фамилии оказываются одинаковыми. Это означает, что имена вкладчиков упорядочиваются по фамилиям, а в пределах одинаковых фамилий — по именам.

И последнее замечание: ради наглядности в данном примере оба компаратора создаются явным образом в виде классов `CompLastNames` и `ThenByFirstNames`, но вместо них можно воспользоваться лямбда-выражениями. Попробуйте сделать это сами в качестве упражнения, следуя тому же самому общему образцу из приведенного выше примера с классом `CompDemo2`.

Алгоритмы коллекций

В каркасе коллекций определяется ряд алгоритмов, которые можно применять к коллекциям и отображениям. Эти алгоритмы определены в виде статических методов из класса `Collections`, перечисленных в табл. 18.19. Как упоминалось ранее, в версии JDK 5 все алгоритмы были переделаны с учетом обобщений.

Таблица 18.19. Алгоритмы, определенные в классе `Collections`

Метод	Описание
<code>static <T> boolean addAll(Collection<? super T> c, T... элементы)</code>	Вставляет заданные <i>элементы</i> в указанную коллекцию <i>c</i> . Возвращает логическое значение true , если элементы были добавлены, а иначе — логическое значение false
<code>static <T> Queue<T>asLifoQueue(Deque<T> c)</code>	Возвращает представление заданной коллекции <i>c</i> в виде стека, действующего по принципу “последним пришел — первым обслужен”
<code>static <T> int binarySearch(List<? extends T> список, T значение, Comparator<? super T> c)</code>	Осуществляет поиск указанного <i>значения</i> в заданном <i>списке</i> , упорядоченном в соответствии заданным компаратором <i>c</i> . Возвращает позицию указанного <i>значения</i> в заданном <i>списке</i> или отрицательное значение, если <i>значение</i> не найдено

Продолжение табл. 18.19

Метод	Описание
<code>static <T> int binarySearch(List<? Extends Comparable<? super T>> список, T значение)</code>	Осуществляет поиск указанного <i>значения</i> в заданном <i>списке</i> , который должен быть отсортирован. Возвращает позицию указанного <i>значения</i> в заданном <i>списке</i> или отрицательное значение, если <i>значение</i> не найдено
<code>static <E> Collection<E> checkedCollection(Collection<E> c, Class<E> t)</code>	Возвращает динамически типизируемое представление коллекции. Попытка ввести несовместимый элемент в коллекцию вызовет исключение типа <code>ClassCastException</code>
<code>static <E> List<E> checkedList(List<E> c, Class<E> t)</code>	Возвращает динамически типизируемое представление списка типа <code>List</code> . Попытка ввести несовместимый элемент в список вызовет исключение типа <code>ClassCastException</code>
<code>static <K, V> Map<K, V>checkedMap(Map<K, V> c, Class<K> тип_ключа, Class<V> тип_значения)</code>	Возвращает динамически типизируемое представление отображения типа <code>Map</code> . Попытка ввести несовместимый элемент в отображение вызовет исключение типа <code>ClassCastException</code>
<code>static <K, V> NavigableMap<K, V> checkedNavigableMap (NavigableMap<K, V> nm, Class<E> тип_ключа, Class<V> тип_значения)</code>	Возвращает динамически типизируемое представление отображения типа <code>NavigableMap</code> . Попытка ввести несовместимый элемент в отображение вызовет исключение типа <code>ClassCastException</code> (добавлен в версии JDK 8)
<code>static <E> NavigableSet<E> checkedNavigableSet (NavigableSet<E> ns, Class<E> t)</code>	Возвращает динамически типизируемое представление в множество типа <code>NavigableSet</code> . Попытка ввести несовместимый элемент в множество вызовет исключение типа <code>ClassCastException</code> (добавлен в версии JDK 8)
<code>static <E> Queue<E> checkedQueue(Queue<E> q, Class<E> t)</code>	Возвращает динамически типизируемое представление очереди типа <code>checkedQueue</code> . Попытка ввести несовместимый элемент в множество вызовет исключение типа <code>ClassCastException</code> (добавлен в версии JDK 8)
<code>static <E> List<E> checkedSet(Set<E> c, Class<E> t)</code>	Возвращает динамически типизируемое представление множества типа <code>Set</code> . Попытка ввести несовместимый элемент в множество вызовет исключение типа <code>ClassCastException</code>
<code>static <K, V> SortedMap<K, V>checkedSortedMap(SortedMap<K, V> c, Class<K> тип_ключа, Class<V> тип_значения)</code>	Возвращает динамически типизируемое представление отображения типа <code>SortedMap</code> . Попытка ввести несовместимый элемент в отображение вызовет исключение типа <code>ClassCastException</code>
<code>static <E> SortedSet<E> checkedSortedSet(SortedSet<E> c, Class<E> t)</code>	Возвращает динамически типизируемое представление множества типа <code>SortedSet</code> . Попытка ввести несовместимый элемент в множество вызовет исключение типа <code>ClassCastException</code>

Метод	Описание
<code>static <T> void copy(List<? super T> <i>список1</i>, List<? Extends T> <i>список2</i>)</code>	Копирует элементы из <i>списка2</i> в <i>список1</i>
<code>static boolean disjoint(Collection<?> <i>a</i>, Collection<?> <i>b</i>)</code>	Сравнивает элементы коллекций <i>a</i> и <i>b</i> . Возвращает логическое значение true , если обе коллекции не содержат общие элементы (т.е. не пересекающиеся множества элементов), а иначе – логическое значение false
<code>static <T> Enumeration<T> emptyEnumeration()</code>	Возвращает пустое перечисление, т.е. перечисление без элементов
<code>static <T> Iterator<T> emptyIterator()</code>	Возвращает пустой итератор, т.е. итератор без элементов
<code>static <T> List<T> emptyList()</code>	Возвращает неизменяемый, пустой список типа, выводимого из интерфейса List
<code>static <T> ListIterator<T> emptyListIterator()</code>	Возвращает пустой итератор списка, т.е. итератор списка без элементов
<code>static <K, V> Map<K, V> emptyMap()</code>	Возвращает неизменяемое, пустое отображение типа, выводимого из интерфейса Map
<code>static <K, V> NavigableMap<K, V> emptyNavigableMap()</code>	Возвращает неизменяемое, пустое отображение типа, выводимого из интерфейса NavigableMap (добавлен в версии JDK 8)
<code>static <E> NavigableSet<E> emptyNavigableSet()</code>	Возвращает неизменяемое, пустое множество типа, выводимого из интерфейса NavigableSet (добавлен в версии JDK 8)
<code>static <T> Set<T> emptySet()</code>	Возвращает неизменяемое, пустое множество типа, выводимого из интерфейса Set
<code>static <K, V> SortedMap<K, V> emptySortedMap()</code>	Возвращает неизменяемое, пустое отображение типа, выводимого из интерфейса SortedMap (добавлен в версии JDK 8)
<code>static <E> SortedSet<E> emptySortedSet()</code>	Возвращает неизменяемое, пустое множество типа, выводимого из интерфейса SortedSet (добавлен в версии JDK 8)
<code>static <T> Enumeration<T> enumeration(Collection<T> <i>c</i>)</code>	Возвращает перечисление элементов из заданной коллекции <i>c</i> . (См. раздел “Интерфейс Enumeration ” далее в этой главе)
<code>static <T> void fill(List<? super T> <i>список</i>, T <i>объект</i>)</code>	Присваивает указанный <i>объект</i> каждому элементу заданного <i>списка</i>
<code>static int frequency(Collection<?> <i>c</i>, Object <i>объект</i>)</code>	Подсчитывает количество вхождений указанного <i>объекта</i> в заданной коллекции <i>c</i> и возвращает результат
<code>static int indexOfSubList(List<?> <i>список</i>, List<?> <i>подсписок</i>)</code>	Осуществляет поиск первого вхождения указанного <i>подсписка</i> в заданный <i>список</i> . Возвращает индекс первого совпадения или значение -1 , если совпадение не обнаружено

Продолжение табл. 18.19

Метод	Описание
static int lastIndexOfSubList(List<? <i>список</i>, List<? <i>подсписок</i>)	Осуществляет поиск последнего вхождения указанного <i>подсписка</i> в заданный <i>список</i> . Возвращает индекс первого совпадения или значение -1, если совпадение не обнаружено
static <T> ArrayList<T>list(Enumeration<T> <i>перечисление</i>)	Возвращает списочный массив типа ArrayList , содержащий элементы заданного <i>перечисления</i>
static <T> T max(Collection<? extends T> c, Comparator<? super T> <i>компаратор</i>)	Возвращает максимальный элемент из указанной коллекции <i>c</i> , определяемый заданным <i>компаратором</i>
static <T extends Object & Comparable<? super T>> Tmax(Collection<? extends T> c)	Возвращает максимальный элемент из указанной коллекции <i>c</i> , определяемый естественным упорядочением. Коллекция должна быть отсортированной
static <T> T min(Collection<? extends T> c, Comparator<? super T> <i>компаратор</i>)	Возвращает минимальный элемент из указанной коллекции <i>c</i> , определяемый заданным <i>компаратором</i> . Коллекция необязательно должна быть отсортированной
static <T extends Object & Comparable<? super T>>T min(Collection<? extends T> c)	Возвращает максимальный элемент из указанной коллекции <i>c</i> , определяемый естественным упорядочением
static <T> List<T> nCopies(int <i>количество</i>, T <i>объект</i>)	Возвращает заданное <i>количество</i> копий указанного <i>объекта</i> , содержащихся в неизменяемом списке. Значение параметра <i>количество</i> должно быть больше или равно нулю
static <E> Set<E> newSetFromMap(Map<E, Boolean> <i>m</i>)	Создает и возвращает множество, исходя из указанного отображения <i>m</i> , которое должно быть пустым на момент вызова данного метода
static <T> boolean replaceAll(List<T> <i>список</i>, T <i>старый</i>, T <i>новый</i>)	Заменяет все вхождения заданного элемента <i>старый</i> на элемент <i>новый</i> в указанном <i>списке</i> . Возвращает логическое значение true , если произведена хотя бы одна замена, а иначе — логическое значение false
static void reverse(List<T> <i>список</i>)	Изменяет на обратный порядок следования элементов в указанном <i>списке</i>
static <T> Comparator<T> reverseOrder(Comparator<T> <i>компаратор</i>)	Возвращает компаратор, обратный переданному <i>компаратору</i> . Следовательно, возвращаемый компаратор обращает результат сравнения, выполненного заданным <i>компаратором</i>
static <T> Comparator<T>reverseOrder()	Возвращает обратный компаратор, который обращает результат сравнения двух элементов коллекции
static void rotate(List<T> <i>список</i>, int <i>n</i>)	Смещает указанный <i>список</i> на <i>n</i> позиций вправо. Для смещения влево следует указать отрицательное значение параметра <i>n</i>

Метод	Описание
<code>static void shuffle(List<T> список, Random r)</code>	Перетасовывает (случайным образом) элементы указанного <i>списка</i> , используя значение параметра <i>r</i> в качестве исходного для получения случайных чисел
<code>static void shuffle(List<T> список)</code>	Перетасовывает (случайным образом) элементы указанного <i>списка</i>
<code>static <T> Set<T> singleton(T объект)</code>	Возвращает заданный <i>объект</i> в виде неизменяемого множества. Это простейший способ преобразовать одиночный объект в множество
<code>static <T> List<T> singletonList(T объект)</code>	Возвращает заданный <i>объект</i> в виде неизменяемого списка. Это простейший способ преобразовать одиночный объект в список
<code>static <K, V> Map<K, V> singletonMap(K k, V v)</code>	Возвращает пару <i>k, v</i> “ключ–значение” в виде неизменяемого отображения. Это простейший способ преобразовать пару “ключ–значение” в отображение
<code>static <T> void sort(List<T> список, Comparator<? super T> компаратор)</code>	Сортирует элементы указанного <i>списка</i> в соответствии с заданным <i>компаратором</i>
<code>static <T extends Comparable<? super T>> void sort(List<T> список)</code>	Сортирует элементы указанного <i>списка</i> в соответствии с естественным упорядочением
<code>static void swap(List<T> список, int индекс1, int индекс2)</code>	Меняет местами элементы указанного <i>списка</i> , обозначаемые параметрами <i>индекс1</i> и <i>индекс2</i>
<code>static <T> Collection<T> synchronizedCollection(Collection<T> c)</code>	Возвращает потокобезопасную коллекцию, исходя из указанной коллекции <i>c</i>
<code>static <T> List<T> synchronizedList(List<T> список)</code>	Возвращает потокобезопасный список, исходя из указанного <i>списка</i>
<code>static <K, V> Map<K, V> synchronizedMap(Map<K, V> m)</code>	Возвращает потокобезопасное отображение, исходя из указанного отображения <i>m</i>
<code>static <K, V> NavigableMap<K, V> synchronizedNavigableMap(NavigableMap<K, V> nm)</code>	Возвращает синхронизированное навигационное отображение, исходя из указанного отображения <i>nm</i> (добавлен в версии JDK 8)
<code>static <T> NavigableSet<T> synchronizedNavigableSet(NavigableSet<T> ns)</code>	Возвращает синхронизированное навигационное множество, исходя из указанного множества <i>ns</i> (добавлен в версии JDK 8)
<code>static <T> Set<T> synchronizedSet(Set<T> s)</code>	Возвращает потокобезопасное множество, исходя из указанного множества <i>s</i>
<code>static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> sm)</code>	Возвращает потокобезопасное отсортированное отображение, исходя из указанного отображения <i>sm</i>

Окончание табл. 18.19

Метод	Описание
<code>static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> ss)</code>	Возвращает потокобезопасное отсортированное множество, исходя из указанного множества <i>ss</i>
<code>static <T> Collection<T> unmodifiableCollection(Collection<? extends T> c)</code>	Возвращает неизменяемую коллекцию, исходя из указанной коллекции <i>c</i>
<code>static <T> List<T> unmodifiableList(List<? extends T> список)</code>	Возвращает неизменяемый список, исходя из указанного <i>списка</i>
<code>static <K, V> Map<K, V> unmodifiableMap(Map<? extends K, ? extends V> m)</code>	Возвращает неизменяемое отображение, исходя из указанного отображения <i>m</i>
<code>static <K, V> NavigableMap<K, V> unmodifiableNavigableMap(NavigableMap<K, ? extends V> nm)</code>	Возвращает неизменяемое навигационное отображение, исходя из указанного отображения <i>nm</i> (добавлен в версии JDK 8)
<code>static <T> NavigableSet<T> unmodifiableNavigableSet(NavigableSet<T> ns)</code>	Возвращает неизменяемое навигационное множество, исходя из указанного множества <i>ns</i> (добавлен в версии JDK 8)
<code>static <T> Set<T> unmodifiableSet(Set<? extends T> s)</code>	Возвращает неизменяемое множество, исходя из указанного множества <i>s</i>
<code>static <K, V> SortedMap<K, V> unmodifiableSortedMap(SortedMap<K, ? extends V> sm)</code>	Возвращает неизменяемое отсортированное отображение, исходя из указанного отображения <i>sm</i>
<code>static <T> SortedSet<T> unmodifiableSortedSet(SortedSet<T> ss)</code>	Возвращает неизменяемое отсортированное множество, исходя из указанного множества <i>ss</i>

При попытке сравнить несовместимые типы некоторые из перечисленных выше методов могут сгенерировать исключение типа `ClassCastException`, а при попытке видоизменить неизменяемые коллекции — исключение типа `UnsupportedOperationException`. В зависимости от конкретного метода возможны и другие исключения.

Особое внимание следует уделить ряду *проверяемых* методов вроде метода `checkedCollection()`, возвращающего то, что в документации на прикладной программный интерфейс API называется “динамически типизируемым представлением” коллекций. Такое представление служит ссылкой на коллекцию, которая во время выполнения контролирует вводимые в коллекцию объекты на предмет совместимости типов. Любая попытка ввести в коллекцию несовместимый объект вызовет исключение типа `ClassCastException`. Пользоваться этим представлением удобно на стадии отладки, поскольку оно гарантирует наличие в коллекции достоверных элементов. К числу проверяемых относятся методы `checkedSet()`, `checkedList()`, `checkedMap()` и т.д. Они позволяют получить динамически типизированное представление указанной коллекции.

Следует заметить, что некоторые методы наподобие `synchronizedList()` и `synchronizedSet()` служат для получения синхронизированных (*потокобезопасных*) копий различных коллекций. Как упоминалось ранее, стандартные реализа-

ции коллекций, как правило, не синхронизированы. Для синхронизации следует применять синхронизирующие алгоритмы. И еще одно замечание: итераторы синхронизированных коллекций должны использоваться в пределах блоков кода с модификатором доступа `synchronized`.

Ряд *неизменяемых* методов, имена которых начинаются с префикса `unmodifiable`, возвращают представления различных коллекций, которые не могут быть изменены. Это может оказаться удобным в том случае, если требуется гарантировать доступ к коллекциям только для чтения и без права записи.

В интерфейсе `Collection` определены три статические переменные: `EMPTY_SET`, `EMPTY_LIST` и `EMPTY_MAP`. Все они неизменяемы.

В приведенном ниже примере программы демонстрируются некоторые алгоритмы коллекций. В этой программе создается и инициализируется связный список. Метод `reverseOrder()` возвращает компаратор типа `Comparator`, который обрабатывает результат сравнения объектов типа `Integer`. Элементы списка сначала сортируются в соответствии с этим компаратором, а затем выводятся. Далее этот список перетасовывается методом `shuffle()`, после чего из него выводятся минимальное и максимальное значения.

```
// Продемонстрировать применение различных алгоритмов коллекций
import java.util.*;

class AlgorithmsDemo {
    public static void main(String args[]) {
        // создать неинициализированный связный список
        LinkedList<Integer> ll = new LinkedList<Integer>();
        ll.add(-8);
        ll.add(20);
        ll.add(-20);
        ll.add(8);

        // создать компаратор с обратным упорядочением
        Comparator<Integer> r = Collections.reverseOrder();

        // отсортировать список с помощью этого компаратора
        Collections.sort(ll, r);

        System.out.print("Список отсортирован в обратном порядке: ");

        for(int i : ll)
            System.out.print(i+ " ");

        System.out.println();

        // перетасовать список
        Collections.shuffle(ll);

        // вывести перетасованный список
        System.out.print("Список перетасован: ");

        for(int i : ll)
            System.out.print(i + " ");

        System.out.println();
        System.out.println("Минимум: " + Collections.min(ll));
        System.out.println("Максимум: " + Collections.max(ll));
    }
}
```

Ниже приведен результат, выводимый данной программой.

```
Список отсортирован в обратном порядке: 20 8 -8 -20  
Список перетасован: 20 -20 8 -8  
Минимум: -20  
Максимум: 20
```

Обратите внимание на то, что методы `min()` и `max()` оперируют списком после того, как он был перетасован. Ни один из этих методов не требует, чтобы список был отсортирован.

Массивы

Класс `Arrays` предоставляет различные методы, удобные для обращения с массивами. Эти методы помогают восполнить пробел между коллекциями и массивами. Каждый метод, определенный в классе `Arrays`, рассматривается далее в этом разделе.

Метод `asList()` возвращает список, исходя из указанного массива. Иными словами, список и массив ссылаются на одно и то же место в оперативной памяти. Ниже приведена общая форма этого метода, где параметр *массив* обозначает конкретный массив, содержащий данные.

```
static <T> List asList(T... массив)
```

В методе `binarySearch()` алгоритм двоичного поиска применяется для обнаружения заданного значения. Этот метод следует применять к отсортированным массивам. Ниже приведены некоторые формы метода `binarySearch()`, дополнительные его формы обеспечивают поиск значений в заданных пределах.

```
static int binarySearch(byte массив[], byte значение)  
static int binarySearch(char массив[], char значение)  
static int binarySearch(double массив[], double значение)  
static int binarySearch(float массив[], float значение)  
static int binarySearch(int массив[], int значение)  
static int binarySearch(long массив[], long значение)  
static int binarySearch(short массив[], short значение)  
static int binarySearch(Object массив[], Object значение)  
static <T> int binarySearch(T[] массив, T значение,  
    Comparator<? super T> c)
```

Здесь параметр *массив* обозначает конкретный массив, в котором осуществляется поиск, а параметр *значение* — искомое значение. В двух последних формах данного метода генерируется исключение типа `ClassCastException`, если указанный *массив* содержит элементы, которые нельзя сравнивать (например, объекты типа `Double` и `StringBuffer`), или же если заданное *значение* несовместимо по типу с элементами указанного *массива*. В последней форме компаратор *c* типа `Comparator` используется для определения порядка расположения элементов в указанном *массиве*. Но в любом случае возвращается индекс элемента, если заданное *значение* содержится в указанном *массиве*, а иначе — отрицательное значение.

Метод `copyOf()` возвращает копию массива и имеет следующие формы:

```

static boolean[] copyOf(boolean[] источник, int длина)
static byte[] copyOf(byte[] источник, int длина)
static char[] copyOf(char[] источник, int длина)
static double[] copyOf(double[] источник, int длина)
static float[] copyOf(float[] источник, int длина)
static int[] copyOf(int[] источник, int длина)
static long[] copyOf(long[] источник, int длина)
static short[] copyOf(short[] источник, int длина)
static <T> T[] copyOf(T[] источник, int длина)
static <T, U> T[] copyOf(U[] источник, int длина,
    Class<? extends T[]> резльтирующий_тип)

```

Исходный массив задается в качестве параметра *источник*, а длина копии – в качестве параметра *длина*. Если копия длиннее, чем указанный *источник*, она дополняется нулями (для числовых массивов), пустыми значениями null (для массивов объектов) или логическими значениями false (для булевых массивов). Если копия короче, чем указанный *источник*, она усекается. В последней форме *резльтирующий_тип* становится типом возвращаемого массива. Если указанная *длина* отрицательна, то генерируется исключение типа `NegativeArraySizeException`. Если указанный *источник* содержит пустое значение null, то генерируется исключение типа `NullPointerException`. А если *резльтирующий_тип* несовместим с типом указанного *источника*, то генерируется исключение типа `ArrayStoreException`.

Метод `copyOfRange()` также возвращает копию массива в заданных пределах и имеет следующие формы:

```

static boolean[] copyOfRange(boolean[] источник, int начало, int конец)
static byte[] copyOfRange(byte[] источник, int начало, int конец)
static char[] copyOfRange(char[] источник, int начало, int конец)
static double[] copyOfRange(double[] источник, int начало, int конец)
static float[] copyOfRange(float[] источник, int начало, int конец)
static int[] copyOfRange(int[] источник, int начало, int конец)
static long[] copyOfRange(long[] источник, int начало, int конец)
static short[] copyOfRange(short[] источник, int начало, int конец)
static <T> T[] copyOfRange(T[] источник, int начало, int конец)
static <T, U> T[] copyOfRange(U[] источник, int начало, int конец,
    Class<? extends T[]> резльтирующий_тип)

```

Исходный массив задается в качестве параметра *источник*. Пределы для копирования задаются индексами, передаваемыми в качестве параметров *начало* и *конец*. Заданные пределы простираются от позиции *начало* до позиции *конец*-1. Если заданные пределы длиннее, чем указанный *источник*, то копия дополняется нулями (для числовых массивов), пустыми значениями null (для массивов объектов) или логическими значениями false (для булевых массивов). В последней форме указанный *резльтирующий_тип* становится типом возвращаемого массива.

Если заданное *начало* отрицательное или больше длины указанного *источника*, то генерируется исключение типа `ArrayIndexOutOfBoundsException`. Если заданное *начало* больше, чем заданный *конец*, то генерируется исключение типа `IllegalArgumentException`. Если указанный *источник* содержит пустое значение null, то генерируется исключение типа `NullPointerException`. А если *резльтирующий_тип* несовместим с типом указанного *источника*, то генерируется исключение типа `ArrayStoreException`.

Метод `equals()` возвращает логическое значение `true`, если оба сравниваемых массива равнозначны, а иначе — логическое значение `false`. Ниже приведены различные формы метода `equals()`, где параметры *массив1* и *массив2* обозначают сравниваемые на равенство массивы.

```
static boolean equals(boolean массив1[], boolean массив2[])
static boolean equals(byte массив1[], byte массив2[])
static boolean equals(char массив1[], char массив2[])
static boolean equals(double массив1[], double массив2[])
static boolean equals(float массив1[], float массив2[])
static boolean equals(int массив1[], int массив2[])
static boolean equals(long массив1[], long массив2[])
static boolean equals(short массив1[], short массив2[])
static boolean equals(Object массив1[], Object массив2[])
```

Метод `deepEquals()` позволяет определить, являются ли равнозначными два массива, которые могут содержать вложенные массивы. Этот метод объявляется следующим образом:

```
static boolean deepEquals(Object[] a, Object[] b)
```

Метод `deepEquals()` возвращает логическое значение `true`, если переданные ему массивы *a* и *b* равнозначны. Если массивы *a* и *b* содержат вложенные массивы, они также сравниваются. Если же массивы *a* и *b* или вложенные в них массивы отличаются, данный метод возвращает логическое значение `false`.

Метод `fill()` присваивает значение всем элементам массива. Иными словами, он заполняет массив указанным значением. У метода `fill()` имеются два варианта. В первом варианте, формы которого приведены ниже, заполняется весь массив.

```
static void fill(boolean массив[], boolean значение)
static void fill(byte массив[], byte значение)
static void fill(char массив[], char значение)
static void fill(double массив[], double значение)
static void fill(float массив[], float значение)
static void fill(int массив[], int значение)
static void fill(long массив[], long значение)
static void fill(short массив[], short значение)
static void fill(Object массив[], Object значение)
```

Здесь заданное *значение* присваивается всем элементам указанного массива. Во втором варианте метода `fill()` заданное значение присваивается подмножеству массива.

Метод `sort()` сортирует массив таким образом, чтобы упорядочить его элементы по нарастающей. У метода `sort()` имеются два варианта. В первом варианте, формы которого приведены ниже, сортируется весь массив.

```
static void sort(byte массив[])
static void sort(char массив[])
static void sort(double массив[])
static void sort(float массив[])
static void sort(int массив[])
static void sort(long массив[])
static void sort(short массив[])
static void sort(Object массив[])
static <T> void sort(T массив[], Comparator<? super T> c)
```

Здесь параметр *массив* обозначает сортируемый массив. В последней форме параметр *c* обозначает компаратор типа `Comparator`, используемый для упоря-

дочения элементов указанного массива. В двух последних формах может генерироваться исключение типа `ClassCastException`, если элементы сортируемого массива несовместимы. Во втором варианте метода `sort()` имеется возможность указать пределы, в которых требуется отсортировать массив.

В версии JDK 8 класс `Arrays` дополнен рядом новых методов. Наиболее важным из них, вероятно, является метод `parallelSort()`, поскольку он сначала сортирует параллельно отдельные части массива в порядке нарастания, а затем объединяет полученные результаты. Благодаря этому значительно сокращается время сортировки массива. Как и у метода `sort()`, у метода `parallelSort()` имеются два основных варианта, причем каждый с несколькими перегружаемыми формами. В первом варианте, формы которого приведены ниже, сортируется весь массив.

```
static void parallelSort(byte массив[])
static void parallelSort(char массив[])
static void parallelSort(double массив[])
static void parallelSort(float массив[])
static void parallelSort(int массив[])
static void parallelSort(long массив[])
static void parallelSort(short массив[])
static <T extends Comparable<? super T>> void parallelSort(T массив[])
static <T> void parallelSort(T массив[], Comparator<? super T> c)
```

Здесь параметр *массив* обозначает сортируемый массив. В последней форме параметр *c* обозначает компаратор, используемый для упорядочения элементов массива. В двух последних формах может генерироваться исключение типа `ClassCastException`, если элементы сортируемого массива несовместимы. Во втором варианте метода `parallelSort()` имеется возможность указать пределы, в которых требуется отсортировать массив.

Начиная с версии JDK 8 в классе `Arrays` поддерживаются также итераторы-разделители. И делается это, в частности, с помощью метода `splititerator()`. У этого метода имеются два основных варианта. В первом варианте, формы которого приведены ниже, возвращается итератор-разделитель для всего массива.

```
static Splititerator.OfDouble splititerator(double массив[])
static Splititerator.OfInt splititerator(int массив[])
static Splititerator.OfLong splititerator(long массив[])
static <T> Splititerator splititerator(T массив[])
```

Здесь параметр *массив* обозначает перебираемый итератором-разделителем массив. Во втором варианте метода `splititerator()` имеется возможность указать пределы, в которых требуется перебрать элементы массива.

Начиная с версии JDK 8 в классе `Arrays` поддерживается также новый интерфейс `Stream`, описываемый в главе 29. И делается это, в частности, с помощью метода `stream()`. У этого метода имеются два основных варианта. Ниже приведены перегружаемые формы первого варианта метода `stream()`.

```
static DoubleStream stream(double массив[])
static IntStream stream(int массив[])
static LongStream stream(long массив[])
static <T> Stream stream(T массив[])
```

Здесь параметр *массив* обозначает конкретный массив, на который ссылается поток ввода-вывода. Во втором варианте метода `stream()` имеется возможность

указать пределы, в которых требуется обратиться к элементам массива из потока ввода-вывода.

Помимо упомянутых выше методов, в версии JDK 8 класс `Arrays` дополнен еще тремя новыми методами. Два из них называются `setAll()` и `parallelSetAll()` и присваивают значения всем элементам массива, причем метод `parallelSetAll()` позволяет делать это параллельно. В качестве примера ниже приведена одна из перегружаемых форм каждого из этих методов. У каждого из них имеются также перегружаемые формы для обработки массивов типа `int`, `long` и обобщенного типа.

```
static void setAll(double массив[],
                  IntToDoubleFunction<? extends T> генератор_значений)
static void parallelSetAll(double массив[],
                          IntToDoubleFunction<? extends T> генератор_значений)
```

И наконец, в версии JDK 8 класс `Arrays` имеет еще одно привлекательное дополнение в виде метода `parallelPrefix()`, который видоизменяет массив таким образом, чтобы каждый последующий его элемент содержал накапливаемый результат операции, выполняемой над всеми предыдущими элементами. Так, если это операция умножения, то после возврата из данного метода массив будет содержать значения, связанные с текущим произведением исходных значений. В качестве примера ниже приведена одна из перегружаемых форм метода `parallelPrefix()`.

```
static void parallelPrefix(double массив[],
                          DoubleBinaryOperator функция)
```

Здесь параметр *массив* обозначает обрабатываемый массив, параметр *функция* — операцию, выполняемую над массивом, а тип `DoubleBinaryOperator` — функциональный интерфейс, определенный в пакете `java.util.function`. У метода `parallelPrefix()` имеются и другие перегружаемые формы для обработки массивов типа `int`, `long` и обобщенного типа, а также отдельных элементов массива в заданных пределах.

Кроме того, в классе `Arrays` предоставляются методы `toString()` и `hashCode()` для массивов различных типов. В нем предусмотрены также методы `deepToString()` и `deepHashCode()` для эффективной обработки массивов, содержащих вложенные массивы.

В следующем примере программы демонстрируется применение некоторых методов из класса `Arrays`:

```
// Продемонстрировать применения некоторых методов из класса Arrays
import java.util.*;

class ArraysDemo {
    public static void main(String args[]) {
        // выделить и инициализировать массив
        int array[] = new int[10];
        for(int i = 0; i < 10; i++)
            array[i] = -3 * i;

        // вывести, отсортировать и снова вывести содержимое массива
        System.out.print("Исходный массив: ");
        display(array);
        Arrays.sort(array);
        System.out.print("Отсортированный массив: ");
        display(array);
    }
}
```



```

// заполнить массив и вывести его содержимое
Arrays.fill(array, 2, 6, -1);
System.out.print("Массив после вызова метода fill(): ");
display(array);

// отсортировать и вывести содержимое массива
Arrays.sort(array);
System.out.print("Массив после повторной сортировки: ");
display(array);

// выполнить двоичный поиск значения -9
System.out.print("Значение -9 находится на позиции ");
int index =
    Arrays.binarySearch(array, -9);
System.out.println(index);
}
static void display(int array[]) {
    for(int i: array)
        System.out.print(i + " ");
    System.out.println();
}
}

```

Ниже приведен результат выполнения данной программы.

```

Исходный массив: 0 -3 -6 -9 -12 -15 -18 -21 -24 -27
Отсортированный массив: -27 -24 -21 -18 -15 -12 -9 -6 -3 0
Массив после вызова метода fill(): -27 -24 -1 -1 -1 -1 -9 -6 -3 0
Массив после повторной сортировки: -27 -24 -9 -6 -3 -1 -1 -1 -1 0
Значение -9 находится на позиции 2

```

Унаследованные классы и интерфейсы

Как пояснялось в начале этой главы, в первых версиях пакета `java.util` отсутствовал каркас коллекций. Вместо него был определен ряд классов и интерфейсов, предоставляющих специальные методы для хранения объектов. А когда были внедрены коллекции (начиная с версии J2SE 1.2), то некоторые исходные классы были переделаны для поддержки интерфейсов коллекций. Следовательно, они и теперь формально являются частью каркаса коллекций Collections Framework. И несмотря на то что новые коллекции дублируют функции устаревших классов, на практике обычно используются классы новых коллекций. В общем, унаследованные классы поддерживаются потому, что существует код, в котором они используются.

Следует также иметь в виду, что ни один из современных классов коллекций, описанных в этой главе, не синхронизирован, тогда как все устаревшие классы синхронизированы. В некоторых случаях это отличие может иметь большое значение. Безусловно, коллекции нетрудно синхронизировать, применяя один из алгоритмов, предоставляемых в виде методов класса `Collections`. Ниже перечислены унаследованные классы, определенные в пакете `java.util`.

Dictionary	Hashtable	Properties	Stack	Vector
------------	-----------	------------	-------	--------

Имеется также один унаследованный интерфейс, называемый `Enumeration`. Этот интерфейс и каждый из унаследованных классов рассматриваются по очереди в последующих разделах.

Интерфейс Enumeration

В интерфейсе Enumeration определяются методы, с помощью которых можно перебирать элементы из коллекции объектов, получая их по очереди. Этот унаследованный интерфейс был заменен интерфейсом Iterator. И хотя интерфейс Enumeration применяется до сих пор, он считается не рекомендованным для употребления в новом коде. Тем не менее он применяется в некоторых методах из унаследованных классов (например, из классов Vector или Properties), а также в ряде других классов прикладного программного интерфейса API. В связи с тем что этот интерфейс все еще употребляется в унаследованном коде, он был переделан в обобщенную форму при разработке версии JDK 5. Интерфейс Enumeration объявляется приведенным ниже образом, где E обозначает тип перечисляемых элементов.

```
interface Enumeration<E>
```

В интерфейсе Enumeration определены следующие методы:

```
boolean hasMoreElements()  
E nextElement()
```

При реализации метод hasMoreElements() должен возвращать логическое значение true до тех пор, пока в перечислении еще остаются извлекаемые элементы, а когда элементы уже перечислены — логическое значение false. Метод nextElement() возвращает следующий объект в перечислении. Следовательно, в результате каждого вызова метода nextElement() возвращается следующий объект в перечислении. По окончании перечисления этот метод генерирует исключение типа NoSuchElementException.

Класс Vector

Этот класс реализует динамический массив. Он подобен классу ArrayList, но имеет два отличия: синхронизирован и содержит немало устаревших методов, дублирующих функции методов, определенных в каркасе коллекций Collections Framework. С появлением коллекций класс Vector был переделан как расширение класса AbstractList и дополнен реализацией интерфейса List. В версии JDK 5 он был переделан под синтаксис обобщений и дополнен реализацией интерфейса Iterable. Это означает, что класс Vector стал полностью совместимым с коллекциями, допуская перебор своего содержимого в усовершенствованном цикле for.

Класс Vector объявляется приведенным ниже образом, где E обозначает тип сохраняемых элементов.

```
class Vector<E>
```

Ниже перечислены конструкторы класса Vector.

```
Vector()  
Vector(int размер)  
Vector(int размер, int инкремент)  
Vector(Collection<? extends E> c)
```

В первой форме конструктора создается вектор по умолчанию, имеющий начальный размер 10. Во второй форме конструктора создается вектор, начальная

емкость которого определяется параметром *размер*. В третьей форме создается вектор, имеющий заданный начальный *размер* и *инкремент*. Инкремент определяет количество элементов, которые будут резервироваться всякий раз, когда увеличивается размер вектора. И наконец, в четвертой форме создается вектор, содержащий элементы из указанной коллекции *c*.

Все векторы начинаются с некоторой начальной емкости. Когда эта емкость исчерпана, при последующей попытке сохранить объект в векторе автоматически увеличивается количество выделяемого пространства памяти по мере роста вектора. Это увеличение важно, поскольку выделение памяти — дорогостоящая по времени операция. Общий объем выделяемого каждый раз дополнительного пространства памяти задается величиной инкремента, указываемого при создании вектора. Если не указать величину инкремента, размер вектора будет удваиваться каждый раз при выделении памяти.

В классе `Vector` определяются следующие защищенные элементы данных:

```
int capacityIncrement;
int elementCount;
Object[] elementData;
```

Значение инкремента сохраняется в поле `capacityIncrement`. Количество элементов, находящихся в данный момент в векторе, хранится в поле `elementCount`. Массив, содержащий сам вектор, хранится в поле `elementData`.

Помимо методов коллекций, определенных в интерфейсе `List`, в классе `Vector` определяется ряд унаследованных методов, перечисленных в табл. 18.20.

Класс `Vector` реализует интерфейс `List`, и поэтому вектор можно использовать таким же образом, как и экземпляр класса `ArrayList`. Вектором можно также манипулировать, используя один из унаследованных методов. Например, после создания экземпляра класса `Vector` можно вызвать метод `addElement()`, чтобы ввести элементы в вектор. Чтобы получить элемент, находящийся на определенной позиции в векторе, достаточно вызвать метод `elementAt()`; чтобы получить первый элемент вектора — метод `firstElement()`; а для того чтобы получить последний элемент вектора — метод `lastElement()`. Индекс отдельного элемента можно получить методом `indexOf()` или `lastIndexOf()`. Чтобы удалить элемент из вектора, следует вызвать метод `removeElement()` или `removeElementAt()`.

Таблица 18.20. Унаследованные методы из класса `Vector`

Метод	Описание
<code>void addElement(<i>Е элемент</i>)</code>	Вводит в вектор объект, определяемый параметром <i>элемент</i>
<code>int capacity()</code>	Возвращает емкость вектора
<code>Object clone()</code>	Возвращает дубликат вызывающего объекта
<code>boolean contains(Object <i>элемент</i>)</code>	Возвращает логическое значение true , если заданный <i>элемент</i> содержится в векторе, а иначе — логическое значение false
<code>void copyInto(Object <i>массив</i>[])</code>	Элементы, содержащиеся в вызывающем векторе, копируются в указанный <i>массив</i>

Окончание табл. 18.20

Метод	Описание
E elementAt (int <i>индекс</i>)	Возвращает элемент по указанному <i>индексу</i>
Enumeration < E > elements ()	Возвращает перечисление элементов вектора
void ensureCapacity (int <i>размер</i>)	Устанавливает минимальную емкость вектора равной заданному <i>размеру</i>
E firstElement ()	Возвращает первый элемент вектора
int indexOf (Object <i>элемент</i>)	Возвращает индекс первого вхождения заданного <i>элемента</i> . Если объект не найден в векторе, возвращается значение -1
int indexOf (Object <i>элемент</i> , int <i>начало</i>)	Возвращает индекс первого вхождения заданного <i>элемента</i> , начиная с указанной позиции <i>начало</i> . Если объект не найден в векторе, возвращается значение -1
void insertElementAt (E <i>элемент</i> , int <i>индекс</i>)	Вводит заданный <i>элемент</i> в вектор по указанному <i>индексу</i>
boolean isEmpty ()	Возвращает логическое значение true , если вектор пуст, а иначе — логическое значение false
E lastElement ()	Возвращает последний элемент в векторе
int lastIndexOf (Object <i>элемент</i>)	Возвращает индекс последнего вхождения заданного <i>элемента</i> . Если объект не найден в векторе, возвращается значение -1
int lastIndexOf (Object <i>элемент</i> , int <i>начало</i>)	Возвращает индекс последнего вхождения заданного <i>элемента</i> до указанной позиции <i>начало</i> . Если объект не найден в векторе, возвращается значение -1
void removeAllElements ()	Очищает вектор. После выполнения этого метода размер вектора равен нулю
boolean removeElement (Object <i>элемент</i>)	Удаляет заданный <i>элемент</i> из вектора. Если в векторе содержится больше одного экземпляра заданного элемента, то удаляется только первый из них. Возвращает логическое значение true , если элемент удален, а если объект не найден — логическое значение false
void removeElementAt (int <i>индекс</i>)	Удаляет из вектора элемент по указанному <i>индексу</i>
void setElementAt (E <i>элемент</i> , int <i>индекс</i>)	Устанавливает заданный <i>элемент</i> по указанному <i>индексу</i>
void setSize (int <i>размер</i>)	Устанавливает количество элементов вектора равным заданному <i>размеру</i> . Если новый размер меньше старого, элементы теряются. Если же новый размер больше старого, добавляются пустые элементы
int size ()	Возвращает количество элементов, содержащихся в векторе
String toString ()	Возвращает строковый эквивалент вектора
void trimToSize ()	Устанавливает емкость вектора равной количеству элементов, содержащихся в нем на данный момент

В приведенном ниже примере программы вектор используется для сохранения разных типов числовых объектов. В данном примере демонстрируется ряд унаследованных методов, определенных в классе `Vector`. Кроме того, в данном примере показано, как обращаться с интерфейсом `Enumeration`.

```
// Продемонстрировать различные операции с вектором
import java.util.*;

class VectorDemo {
    public static void main(String args[]) {

        // начальный размер вектора - 3, а инкремент - 2
        Vector<Integer> v = new Vector<Integer>(3, 2);
        System.out.println("Начальный размер вектора: " + v.size());
        System.out.println("Начальная емкость вектора: " + v.capacity());

        v.addElement(1);
        v.addElement(2);
        v.addElement(3);
        v.addElement(4);

        System.out.println(
            "Емкость вектора после ввода четырех элементов: " +
            v.capacity());

        v.addElement(5);
        System.out.println("Текущая емкость вектора: " + v.capacity());

        v.addElement(6);
        v.addElement(7);

        System.out.println("Текущая емкость вектора: " + v.capacity());

        v.addElement(9);
        v.addElement(10);

        System.out.println("Текущая емкость вектора: " + v.capacity());

        v.addElement(11);
        v.addElement(12);

        System.out.println(
            "Первый элемент вектора: " + v.firstElement());
        System.out.println(
            "Последний элемент вектора: " + v.lastElement());
        if(v.contains(3))
            System.out.println("Вектор содержит 3.");

        // перечислить элементы вектора
        Enumeration<Integer> vEnum = v.elements();

        System.out.println("\nЭлементы вектора:");
        while(vEnum.hasMoreElements())
            System.out.print(vEnum.nextElement() + " ");
        System.out.println();
    }
}
```

Ниже приведен результат, выводимый данной программой.

```
Начальный размер вектора: 0
Начальная емкость вектора: 3
```

```
Емкость вектора после ввода четырех элементов: 5
Текущая емкость вектора: 5
Текущая емкость вектора: 7
Текущая емкость вектора: 9
Первый элемент вектора: 1
Последний элемент вектора: 12
Вектор содержит 3.
```

```
Элементы вектора:
1 2 3 4 5 6 7 9 10 11 12
```

Вместо того чтобы полагаться на перебор объектов в цикле, как это делается в приведенном выше примере программы, можно воспользоваться итератором. Для этого в данную программу можно, например, подставить следующий фрагмент кода:

```
// Использовать итератор для вывода содержимого вектора
Iterator<Integer> vItr = v.iterator();

System.out.println("\nЭлементы вектора:");
while(vItr.hasNext())
    System.out.print(vItr.next() + " ");
System.out.println();
```

Для перебора элементов вектора можно также организовать цикл `for` в стиле `for each`, как показано в следующей версии предыдущего фрагмента кода:

```
// Использовать усовершенствованный цикл for в стиле for each
// для вывода элементов вектора
System.out.println("\nЭлементы вектора:");
for(int i : v)
    System.out.print(i + " ");

System.out.println();
```

Интерфейс `Enumeration` не рекомендуется применять в новом коде, поэтому для перебора всех элементов вектора обычно применяются итераторы и циклы `for` в стиле `for each`. Безусловно, существует еще немалый объем кода, в котором используется интерфейс `Enumeration`. Правда, перечисления и итераторы действуют практически одинаково.

Класс Stack

Класс `Stack` является производным от класса `Vector` и реализует стандартный стек, действующий по принципу “последним пришел — первый обслужен”. В классе `Stack` определяется только конструктор по умолчанию, создающий пустой стек. В версии JDK 5 класс `Stack` был переделан под синтаксис обобщений и теперь объявляется следующим образом:

```
class Stack<E>
```

где `E` обозначает тип элементов, сохраняемых в стеке. Класс `Stack` включает в себя все методы, определенные в классе `Vector`, и дополняет их рядом своих методов, перечисленных в табл. 18.21.

Таблица 18.21. Методы из класса `Stack`

Метод	Описание
<code>boolean empty()</code>	Возвращает логическое значение true , если стек пустой, а если он содержит элементы, то логическое значение false
<code>E peek()</code>	Возвращает элемент из вершины стека, но не удаляет его
<code>E pop()</code>	Возвращает элемент из вершины стека, удаляя его
<code>E push(E элемент)</code>	Размещает заданный <i>элемент</i> в стеке и возвращает этот <i>элемент</i>
<code>int search(Object элемент)</code>	Осуществляет поиск заданного <i>элемента</i> в стеке. Если заданный <i>элемент</i> найден, возвращается его смещение относительно вершины стека, а иначе — значение -1

Чтобы разместить объект на вершине стека, следует вызвать метод `push()`; чтобы удалить и вернуть верхний элемент из стека — метод `pop()`, а для возврата верхнего элемента без его удаления из стека — метод `peek()`. Исключение типа `EmptyStackException` генерируется в том случае, если вызвать метод `pop()` или `peek()` для пустого стека. Метод `empty()` возвращает логическое значение **true**, если стек пуст. Метод `search()` определяет, содержится ли объект в стеке, и возвращает количество вызовов метода `pop()`, требующихся для перемещения объекта на вершину стека. Ниже приведен пример программы, в которой создается стек и в нем размещается несколько объектов типа `Integer`, а затем они извлекаются из стека.

```
// Продемонстрировать применение класса Stack
import java.util.*;

class StackDemo {
    static void showpush(Stack<Integer> st, int a) {
        st.push(a);
        System.out.println("push(" + a + ")");
        System.out.println("стек: " + st);
    }

    static void showpop(Stack<Integer> st) {
        System.out.print("pop -> ");
        Integer a = st.pop();
        System.out.println(a);
        System.out.println("стек: " + st);
    }

    public static void main(String args[]) {
        Stack<Integer> st = new Stack<Integer>();

        System.out.println("стек: " + st);

        showpush(st, 42);
        showpush(st, 66);
        showpush(st, 99);
        showpop(st);
        showpop(st);
        showpop(st);

        try {
            showpop(st);
        } catch (EmptyStackException e) {
            System.out.println("стек пуст");
        }
    }
}
```

Ниже приведен результат, выводимый данной программой. Обратите внимание на вызов обработчика исключений типа `EmptyStackException`, позволяющего изящно выйти из положения, когда стек незагружен.

```
стек: []
push(42)
стек: [42]
push(66)
стек: [42, 66]
push(99)
стек: [42, 66, 99]
pop -> 99
стек: [42, 66]
pop -> 66
стек: [42]
pop -> 42
стек: []
pop -> стек пуст
```

Следует также заметить, что класс `Stack` по-прежнему рекомендуется для употребления. Тем не менее вместо него лучше выбрать класс `ArrayDeque`.

Класс Dictionary

Класс `Dictionary` является абстрактным, предоставляет хранилище для пар “ключ–значение” и действует аналогично отображению типа `Map`. Задав ключ и значение, можно сохранить значение в объекте класса `Dictionary`. Как только значение будет сохранено, его можно извлечь по связанному с ним ключу. Аналогично отображению, словарь типа `Dictionary` можно рассматривать как список пар “ключ–значение”. И хотя класс `Dictionary` до сих пор рекомендуется для употребления, его можно считать устаревшим, поскольку его функцией полностью заменяет отображение. Тем не менее класс `Dictionary` по-прежнему применяется, и поэтому он рассматривается здесь.

В версии JDK 5 класс `Dictionary` был сделан обобщенным. Он объявляется следующим образом:

```
class Dictionary<K, V>
```

где **K** обозначает тип ключей, а **V** — тип значений. Абстрактные методы, определенные в классе `Dictionary`, перечислены в табл. 18.22.

Чтобы ввести ключ и его значение в словарь, достаточно вызвать метод `put()`, а для того чтобы извлечь значение из словаря по заданному ключу — метод `get()`. Ключи и значения могут быть возвращены в виде перечисления методами `keys()` и `elements()` соответственно. Метод `size()` возвращает количество пар “ключ–значение”, хранящихся в словаре, а метод `isEmpty()` — логическое значение `true`, если словарь пуст. Для удаления любой пары “ключ–значение” из словаря следует вызвать метод `remove()`.

Помните! Класс `Dictionary` считается устаревшим. Для получения функциональных возможностей, требующихся для хранения пар “ключ–значение”, следует реализовать интерфейс `Map`.

Таблица 18.22. Абстрактные методы из класса Dictionary

Метод	Описание
Enumeration<V> elements()	Возвращает перечисление значений, хранящихся в словаре
V get(Object <i>ключ</i>)	Возвращает объект, содержащий значение, связанное с заданным <i>ключом</i> . Если заданный <i>ключ</i> не содержится в словаре, возвращается пустое значение null
boolean isEmpty()	Возвращает логическое значение true , если словарь пуст, а если он содержит хотя бы один ключ, то логическое значение false
Enumeration<K> keys()	Возвращает перечисление ключей, хранящихся в словаре
V put(K <i>ключ</i>, V <i>значение</i>)	Вводит ключ и значение в словарь. Возвращает пустое значение null , если заданный <i>ключ</i> отсутствует в словаре, а иначе — предыдущее значение, связанное с заданным <i>ключом</i>
V remove(Object <i>ключ</i>)	Удаляет заданный <i>ключ</i> и его значение из словаря. Возвращает значение, связанное с заданным <i>ключом</i> . Если заданный <i>ключ</i> отсутствует в словаре, возвращается пустое значение null
int size()	Возвращает количество элементов в словаре

Класс Hashtable

Класс `Hashtable` входил еще в исходный пакет `java.util` и является конкретной реализацией класса `Dictionary`. Но с появлением коллекций класс `Hashtable` был переделан таким образом, чтобы реализовать также интерфейс `Map`. Следовательно, класс `Hashtable` интегрирован в каркас коллекций `Collections Framework`. Он подобен классу `HashMap`, но синхронизирован.

Подобно классу `HashMap`, класс `Hashtable` служит для хранения пар “ключ-значение” в хеш-таблице. Но ни ключи, ни значения не могут быть пустыми. Используя класс `Hashtable`, следует указать объект, который служит ключом, а также значение, которое требуется связать с этим ключом. Ключ затем хешируется, а результирующий хеш-код используется в качестве индекса, по которому значение сохраняется в таблице.

Класс `Hashtable` был сделан обобщенным в версии JDK 5. Он объявляется приведенным ниже образом, где **K** обозначает тип ключей, а **V** — тип значений.

```
class Hashtable<K, V>
```

В хеш-таблице могут храниться объекты только тех классов, в которых переопределяются методы `hashCode()` и `equals()`, определенные в классе `Object`. Метод `hashCode()` должен вычислять и возвращать хеш-код объекта, а метод `equals()` — сравнивать два объекта. Правда, во многих классах, встроенных в Java, метод `hashCode()` уже реализован. Так, в наиболее распространенной хеш-таблице типа `Hashtable` в качестве ключа используется объект типа `String`. В классе `String` реализуются также методы `hashCode()` и `equals()`.

Ниже приведены конструкторы класса `Hashtable`.

```
Hashtable()
Hashtable(int размер)
Hashtable(int размер, float коэффициент_заполнения)
Hashtable(Map<? extends K, ? extends V> m)
```

В первой форме объявляется конструктор по умолчанию. Во второй форме конструктора создается хеш-таблица, имеющая заданный начальный *размер*, по умолчанию равный 11. В третьей форме создается хеш-таблица, имеющая заданный начальный *размер* и *коэффициент заполнения*. Этот коэффициент находится в пределах от 0,0 до 1,0 и определяет, насколько полной должна быть хеш-таблица, прежде чем она будет расширена. Точнее говоря, когда число элементов превышает емкость хеш-таблицы, умноженную на коэффициент заполнения, хеш-таблица расширяется. Если коэффициент заполнения не указывается, то используется его значение по умолчанию, равное 0,75. И наконец, в четвертой форме конструктора создается хеш-таблица, инициализируемая элементами из указанной коллекции *it*. По умолчанию выбирается коэффициент заполнения, равный 0,75.

Помимо методов, определенных в интерфейсе Map, который теперь реализуется в классе Hashtable, в этом классе определяются также унаследованные методы, перечисленные в табл. 18.23. При попытке использовать пустой ключ некоторые из этих методов генерируют исключение типа NullPointerException.

Таблица 18.23. Унаследованные методы из класса Hashtable

Метод	Описание
<code>void clear()</code>	Устанавливает в исходное состояние и очищает хеш-таблицу
<code>Object clone()</code>	Возвращает дубликат вызывающего объекта
<code>boolean contains(Object значение)</code>	Возвращает логическое значение true , если заданное <i>значение</i> равно некоторому значению в хеш-таблице. Если же заданное <i>значение</i> не найдено, то возвращается логическое значение false
<code>boolean containsKey(Object ключ)</code>	Возвращает логическое значение true , если заданный <i>ключ</i> равен некоторому ключу в хеш-таблице. Если же заданный <i>ключ</i> не найден, то возвращается логическое значение false
<code>boolean containsValue(Object значение)</code>	Возвращает логическое значение true , если заданное <i>значение</i> равно некоторому значению в хеш-таблице. Если же заданное <i>значение</i> не найдено, то возвращается логическое значение false
<code>Enumeration<V> elements()</code>	Возвращает перечисление значений, содержащихся в хеш-таблице
<code>V get(Object ключ)</code>	Возвращает объект, который содержит значение, связанное с заданным <i>ключом</i> . Если заданный <i>ключ</i> не найден в хеш-таблице, то возвращается пустое значение null
<code>boolean isEmpty()</code>	Возвращает логическое значение true , если хеш-таблица пуста, а если содержит хотя бы один ключ — логическое значение false
<code>Enumeration<K> keys()</code>	Возвращает перечисление ключей, содержащихся в хеш-таблице
<code>V put(K ключ, V значение)</code>	Вводит в хеш-таблицу заданные <i>ключ</i> и <i>значение</i> . Возвращает пустое значение null , если заданный <i>ключ</i> отсутствует в хеш-таблице, а иначе — предыдущее значение, связанное с этим ключом

Метод	Описание
<code>void rehash()</code>	Увеличивает размер хеш-таблицы и повторно хеширует все ее ключи
<code>V remove(Object ключ)</code>	Удаляет заданный <i>ключ</i> из хеш-таблицы. Возвращает значение, связанное с заданным <i>ключом</i> . Если заданный <i>ключ</i> отсутствует в хеш-таблице, то возвращается пустое значение <code>null</code>
<code>int size()</code>	Возвращает количество элементов в хеш-таблице
<code>String toString()</code>	Возвращает строковый эквивалент хеш-таблицы

Приведенный ниже пример является переделанным вариантом представленной ранее программы ведения банковских счетов вкладчиков. В данном варианте класс `Hashtable` применяется для хранения Ф.И.О. вкладчиков и остатков на их текущих банковских счетах.

// Продемонстрировать применения класса `Hashtable`

```
import java.util.*;
```

```
class HTDemo {
    public static void main(String args[]) {
        Hashtable<String, Double> balance =
            new Hashtable<String, Double>();
        Enumeration<String> names;

        String str;
        double bal;

        balance.put("Джон Доу", 3434.34);
        balance.put("Том Смит", 123.22);
        balance.put("Джейн Бейкер", 1378.00);
        balance.put("Тод Холл", 99.22);
        balance.put("Ральф Смит", -19.08);

        // показать все счета в хеш-таблице
        names = balance.keys();
        while(names.hasMoreElements()) {
            str = names.nextElement();
            System.out.println(str + ": " + balance.get(str));
        }
        System.out.println();
        // внести сумму 1000 на счет Джона Доу
        bal = balance.get("Джон Доу");
        balance.put("Джон Доу", bal+1000);
        System.out.println("Новый остаток на счете Джона Доу: " +
            balance.get("Джон Доу"));
    }
}
```

Эта программа выводит следующий результат:

```
Тод Холл: 99.22
Ральф Смит: -19.08
Джон Доу: 3434.34
Джейн Бейкер: 1378.0
Том Смит: 123.22
```

```
Новый остаток на счете Джона Доу: 4434.34
```

Следует особо заметить, что, подобно отображению, класс `Hashtable` не поддерживает итераторы непосредственно. Так, в приведенной выше программе для отображения содержимого объекта `balance` используется перечисление. Тем не менее можно получить представления хеш-таблицы в виде множеств, которые допускают применение итераторов. Для этого достаточно воспользоваться одним из методов представления в виде коллекций, определенных в интерфейсе `Map`, например, методом `entrySet()` или `keySet()`. Так, можно получить представление всех ключей в виде множества и перебрать его, используя итератор или организовав усовершенствованный цикл `for` в стиле `for each`. Такой прием демонстрируется в приведенном ниже переделанном варианте предыдущей программы.

```
// Применить итераторы вместе с классом Hashtable
import java.util.*;

class HTDemo2 {
    public static void main(String args[]) {
        Hashtable<String, Double> balance =
            new Hashtable<String, Double>();

        String str;
        double bal;

        balance.put("Джон Доу", 3434.34);
        balance.put("Том Смит", 123.22);
        balance.put("Джейн Бейкер", 1378.00);
        balance.put("Тод Холл", 99.22);
        balance.put("Ральф Смит", -19.08);

        // Вывести все счета в хеш-таблице. Сначала получить
        // представление всех ключей в виде множества
        Set<String> set = balance.keySet();

        // получить итератор
        Iterator<String> itr = set.iterator();

        while(itr.hasNext()) {
            str = itr.next();
            System.out.println(str + ": " +
                balance.get(str));
        }

        System.out.println();

        // внести сумму 1000 на счет Джона Доу
        bal = balance.get("Джон Доу");
        balance.put("Джон Доу", bal+1000);
        System.out.println("Новый остаток на счете Джона Доу: " +
            balance.get("Джон Доу"));
    }
}
```

Класс Properties

Класс `Properties` является производным от класса `Hashtable`. Он служит для поддержки списков значений, в которых ключами и значениями являются объекты типа `String`. Класс `Properties` используется в ряде других классов Java.

Так, при получении значений переменных окружения вызывается метод `System.getProperties()`, который возвращает тип объекта. Несмотря на то что сам класс `Properties` не является обобщенным, некоторые его методы объявляются как обобщенные.

В классе `Properties` определяется следующая переменная экземпляра:

```
Properties defaults;
```

Эта переменная содержит список свойств по умолчанию, связанных с объектом типа `Properties`. В классе `Properties` определяются следующие конструкторы:

```
Properties()  
Properties(Properties свойство_по_умолчанию)
```

В первой форме конструктора создается объект типа `Properties`, не имеющий значений по умолчанию. А во второй форме создается объект, использующий заданное `свойство_по_умолчанию` для установки своих значений по умолчанию. В обоих случаях список свойств пуст.

Помимо методов, наследуемых классом `Properties` от класса `Hashtable`, в этом классе определяются свои методы, перечисленные в табл. 18.24. В классе `Properties` имеется также один не рекомендованный к употреблению метод `save()`. Этот метод был заменен методом `store()`, поскольку он не обрабатывал ошибки должным образом.

Таблица 18.24. Методы из класса `Properties`

Метод	Описание
<code>String getProperty(String ключ)</code>	Возвращает значение, связанное с заданным <i>ключом</i> . Если же заданный <i>ключ</i> отсутствует как в текущем списке, так и в списке свойств по умолчанию, то возвращается пустое значение <code>null</code>
<code>String getProperty(String ключ, String свойство_по_умолчанию)</code>	Возвращает значение, связанное с заданным <i>ключом</i> . Если же заданный <i>ключ</i> отсутствует как в текущем списке, так и в списке свойств по умолчанию, то возвращается указанное <i>свойство_по_умолчанию</i>
<code>void list(PrintStream поток_вывода)</code>	Направляет список свойств в поток вывода, с которым связан указанный <i>поток_вывода</i>
<code>void list(PrintWriter поток_вывода)</code>	Направляет список свойств в поток вывода, с которым связан указанный <i>поток_вывода</i>
<code>void load(InputStream поток_ввода) throws IOException</code>	Вводит список свойств из потока ввода, с которым связан указанный <i>поток_ввода</i>
<code>void load(Reader поток_ввода) throws IOException</code>	Вводит список свойств из потока ввода, с которым связан указанный <i>поток_ввода</i>
<code>void loadFromXML(InputStream поток_ввода) throws IOException, InvalidPropertiesFormatException</code>	Загружает список свойств из XML-документа, с которым связан указанный <i>поток_ввода</i>
<code>Enumeration<?> propertyNames()</code>	Возвращает перечисление ключей. В него включаются также ключи, находящиеся в списке свойств по умолчанию

Окончание табл. 18.24

Метод	Описание
Object setProperty (String <i>ключ</i> , String <i>значение</i>)	Связывает заданные <i>значение</i> и <i>ключ</i> . Возвращает предыдущее значение, связанное с заданным <i>ключом</i> , а если такая связь отсутствует — пустое значение null
void store (OutputStream <i>поток_вывода</i> , String <i>описание</i>) throws IOException	После вывода символьной строки, указанной как <i>описание</i> , список свойств выводится в поток, с которым связан указанный <i>поток_вывода</i>
void store (Writer <i>поток_вывода</i> , String <i>описание</i>) throws IOException	После вывода символьной строки, указанной как <i>описание</i> , список свойств выводится в поток, с которым связан указанный <i>поток_вывода</i>
void storeToXML (OutputStream <i>поток_вывода</i> , String <i>описание</i>) throws IOException	После вывода символьной строки, указанной как <i>описание</i> , список свойств выводится в XML-документ, с которым связан указанный <i>поток_вывода</i>
void storeToXML (OutputStream <i>поток_вывода</i> , String <i>описание</i> , String <i>кодировка</i>)	Список свойств и строка, указанная как <i>описание</i> , выводятся в документ XML, с которым связан указанный <i>поток_вывода</i> , для чего применяется заданная <i>кодировка</i> символов
Set<String> stringPropertyNames ()	Возвращает множество ключей

Класс `Properties` удобен, в частности, тем, что он позволяет указывать значения по умолчанию, которые возвращаются, если ни одно из значений не связано с определенным ключом. Например, значение по умолчанию может быть указано вместе с ключом в методе `getProperty()` следующим образом: `getProperty("имя", "значение_по_умолчанию")`. Если ключ "имя" не найден, то возвращается "значение_по_умолчанию". Создавая объект класса `Properties`, можно передать ему другой экземпляр класса `Properties` в качестве списка свойств по умолчанию для нового экземпляра. Так, если метод `getProperty("foo")` вызывается для заданного объекта типа `Properties` и ключ "foo" не существует, то его поиск осуществляется в его объекте типа `Properties` по умолчанию. Это допускает наличие произвольного числа уровней вложения свойств по умолчанию.

В приведенном ниже примере программы демонстрируется применение класса `Properties`. В этой программе создается список свойств, где ключами являются названия штатов, а значениями — названия их столиц. Обратите внимание на то, что в попытку найти столицу штата Флорида включается значение по умолчанию.

```
// Продемонстрировать применение списка свойств
import java.util.*;
```

```
class PropDemo {
    public static void main(String args[]) {
        Properties capitals = new Properties();
        capitals.put("Иллинойс", "Спрингфилд");
        capitals.put("Миссури", "Джефферсон-Сити");
        capitals.put("Вашингтон", "Олимпия");
        capitals.put("Калифорния", "Сакраменто");
        capitals.put("Индиана", "Индианаполис");
```

```

// получить множество ключей
Set<?> states = capitals.keySet();

// показать все штаты и их столицы
for(Object name : states)
    System.out.println("Столица штата " + name + " - " +
        capitals.getProperty((String)name) + ".");
System.out.println();

// найти штат, отсутствующий в списке, указав значения,
// выбираемые по умолчанию
String str = capitals.getProperty("Флорида", "не найдена");
System.out.println("Столица штата Флорида " + str + ".");
}
}

```

Ниже приведен результат, выводимый данной программой. Штат Флорида отсутствует в списке, поэтому выбирается значение по умолчанию (в данном случае — символьная строка "не найдена").

```

Столица штата Миссури — Джефферсон-Сити.
Столица штата Иллинойс — Спрингфилд.
Столица штата Индиана — Индианаполис.
Столица штата Калифорния — Сакраменто.
Столица штата Вашингтон — Олимпия.

```

```

Столица штата Флорида не найдена.

```

Указывать значение по умолчанию при вызове метода `getProperty()` вполне допустимо, как показано в предыдущем примере, тем не менее для большинства приложений списков свойств имеется лучший способ обращения со значениями по умолчанию. Список свойств по умолчанию удобнее задавать при создании объекта класса `Properties`. Если требуемый ключ не найден в главном списке, его поиск осуществляется в списке по умолчанию. В качестве примера ниже приведен немного измененный вариант предыдущей программы, где применяется список штатов по умолчанию. Теперь, когда запрашивается столица штата Флорида, она обнаруживается в списке по умолчанию.

```

// Использовать список свойств по умолчанию
import java.util.*;
class PropDemoDef {
    public static void main(String args[]) {
        Properties defList = new Properties();
        defList.put("Флорида", "Тэлесси");
        defList.put("Висконсин", "Мэдисон");
        Properties capitals = new Properties(defList);
        capitals.put("Иллинойс", "Спрингфилд");
        capitals.put("Миссури", "Джефферсон-Сити");
        capitals.put("Вашингтон", "Олимпия");
        capitals.put("Калифорния", "Сакраменто");
        capitals.put("Индиана", "Индианаполис");

        // получить множество ключей
        Set<?> states = capitals.keySet();

        // вывести все штаты и их столицы
        for(Object name : states)
            System.out.println("Столица штата " + name + " - " +
                capitals.getProperty((String)name) + ".");
    }
}

```

```

System.out.println();

// Теперь штат Флорида будет найден в списке по умолчанию
String str = capitals.getProperty("Флорида");
System.out.println("Столица Флориды - " + str + ".");
}
}

```

Применение методов store() и load()

Одна из наиболее удобных особенностей класса `Properties` состоит в том, что данные, содержащиеся в объекте класса `Properties`, могут быть легко сохранены и загружены с диска методами `store()` и `load()`. В любой момент объект класса `Properties` можно вывести в поток или ввести его обратно из потока. Это делает списки свойств особенно удобными для реализации простых баз данных. Так, в приведенном ниже примере программы список свойств используется для создания простого телефонного справочника, хранящего имена и номера телефонов абонентов. Чтобы пайти номер абонента, следует ввести его имя. В данной программе методы `store()` и `load()` применяются для сохранения и получения списка обратно. При выполнении этой программы сначала предпринимается попытка загрузить список из файла `phonebook.dat`. Если этот файл существует, он загружается. Затем в список могут быть добавлены новые элементы. В этом случае новый список сохраняется по завершении программы. Обратите внимание, насколько компактный код требуется для реализации небольшой, но вполне работоспособной автоматизированной телефонной книги.

```

/* Простая база данных телефонных номеров,
   построенная на основе списков свойств. */
import java.io.*;
import java.util.*;

class Phonebook {
    public static void main(String args[]) throws IOException {
        Properties ht = new Properties();
        BufferedReader br = new BufferedReader(
            new InputStreamReader(System.in));

        String name, number;
        FileInputStream fin = null;
        boolean changed = false;

        // Попытаться открыть файл phonebook.dat
        try {
            fin = new FileInputStream("phonebook.dat");
        } catch (FileNotFoundException e) {
            // игнорировать отсутствующий файл
        }

        /* Если телефонная книга уже существует, загрузить
           существующие телефонные номера. */
        try {
            if (fin != null) {
                ht.load(fin);
                fin.close();
            }
        } catch (IOException e) {

```



```

        System.out.println("Ошибка чтения файла.");
    }
    // разрешить пользователю вводить новые имена и
    // номера телефонов абонентов
    do {
        System.out.println("Введите имя" +
            " ('выход' для завершения): ");
        name = br.readLine();
        if(name.equals("выход")) continue;
        System.out.println("Введите номер: ");
        number = br.readLine();
        ht.put(name, number);
        changed = true;
    } while(!name.equals("выход"));
    // сохранить телефонную книгу, если она изменилась
    if(changed) {
        FileOutputStream fout =
            new FileOutputStream("phonebook.dat");
        ht.store(fout, "Телефонная книга");
        fout.close();
    }
    // искать номер по имени абонента
    do {
        System.out.println("Введите имя для поиска" +
            " ('выход' для завершения): ");
        name = br.readLine();
        if(name.equals("выход")) continue;
        number = (String) ht.get(name);
        System.out.println(number);
    } while(!name.equals("выход"));
}
}

```

Заключительные соображения по поводу коллекций

Каркас коллекций Collections Framework предоставляет эффективный ряд тщательно спроектированных решений для некоторых наиболее часто встречающихся задач программирования, связанных с сохранением и извлечением данных. Не следует, однако, забывать, что коллекции не предназначены только для “крупных задач” вроде корпоративных баз данных, списков почтовых рассылок или систем учета запасов. Они не менее эффективны и для решения мелких задач. Например, коллекция типа TreeMap может отлично подойти для хранения структуры каталогов или ряда файлов. А класс TreeSet может оказаться очень удобным для хранения информации по управлению проектом. В общем, виды задач, решая которые средствами коллекций можно получить существенный выигрыш, ограничиваются только вашим воображением. И наконец, работая с коллекциями, не следует забывать и о потоках данных, которые теперь интегрированы с коллекциями. Подробнее о новом прикладном программном интерфейсе API потоков ввода-вывода речь пойдет в главе 29.

Пакет `java.util`, часть II. Прочие служебные классы

В этой главе обсуждение пакета `java.util` продолжается рассмотрением классов и интерфейсов, не входящих в состав каркаса коллекций `Collections Framework`. К их числу относятся классы, разбивающие символьные строки на лексемы, оперирующие датами, генерирующие случайные числа, связывающие ресурсы и наблюдающие за событиями. Кроме того, описываются классы `Formatter` и `Scanner`, упрощающие чтение и запись форматированных данных, а также новый класс `Optional`, предоставляющий изящный выход из положения, когда значение просто отсутствует. И наконец, вкратце упоминаются подпакеты, входящие в состав пакета `java.util`. Особый интерес представляет подпакет `java.util.function`, в котором определяется ряд функциональных интерфейсов.

Класс `StringTokenizer`

Обработка текста зачастую предполагает синтаксический анализ или разбор форматированной входной строки. *Синтаксический анализ* подразумевает разделение текста на ряд отдельных частей, так называемых *лексем*, которые способны передать в определенной последовательности некоторое семантическое значение. Класс `StringTokenizer` обеспечивает первую стадию процесса синтаксического анализа, и поэтому его зачастую называют *лексическим анализатором* или просто *сканером*. Этот класс реализует интерфейс `Enumeration`. Таким образом, задав входную строку, средствами класса `StringTokenizer` можно перечислить содержащиеся в ней отдельные лексемы.

Чтобы воспользоваться классом `StringTokenizer`, следует указать входную и символьную строку, содержащую разделители. *Разделители* — это символы, разделяющие лексемы. Каждый символ в символьной строке разделителей рассматривается как допустимый разделитель. Например, символьная строка `" , ; "` устанавливает в качестве разделителей запятую, точку с запятой и двоеточие. Набор разделителей, выбираемых по умолчанию, состоит из пробельных символов: пробела, знака табуляции, перевода строки и возврата каретки.

Ниже приведены конструкторы класса `StringTokenizer`.

```
StringTokenizer(String строка)
StringTokenizer(String строка, String разделители)
StringTokenizer(String строка, String разделители,
                boolean разделитель_как_лексема)
```

Во всех трех формах конструктора параметр *строка* обозначает разделяемую на лексемы символьную строку. В первой форме используются разделители по умолчанию, во второй и в третьей формах параметр *разделители* обозначает символьную строку, задающую разделители. В третьей форме разделители возвращаются в качестве отдельных лексем при синтаксическом анализе символьной строки, если параметр *разделитель_как_лексема* принимает логическое значение `true`. В противном случае разделители не возвращаются. Разделители не возвращаются и в первых двух формах конструктора данного класса.

Как только объект класса `StringTokenizer` будет создан, можно вызвать его метод `nextToken()`, чтобы извлечь последовательные лексемы. Метод `hasMoreTokens()` возвращает логическое значение `true` до тех пор, пока для извлечения еще имеются лексемы. А поскольку класс `StringTokenizer` реализует интерфейс `Enumeration`, то его методы `hasMoreElements()` и `nextElement()` также реализованы и действуют точно так же, как и методы `hasMoreTokens()` и `nextToken()` соответственно. Методы из класса `StringTokenizer` перечислены в табл. 19.1.

Таблица 19.1. Методы из класса `StringTokenizer`

Метод	Описание
<code>int countTokens()</code>	Используя текущий набор разделителей, определяет количество лексем, которые осталось разобрать и вернуть в качестве результата
<code>boolean hasMoreElements()</code>	Возвращает логическое значение <code>true</code> , если в символьной строке остается одна лексема или больше, а иначе — логическое значение <code>false</code>
<code>boolean hasMoreTokens()</code>	Возвращает логическое значение <code>true</code> , если в символьной строке остается одна лексема или больше, а иначе — логическое значение <code>false</code>
<code>Object nextElement()</code>	Возвращает следующую лексему в виде объекта типа <code>Object</code>
<code>String nextToken()</code>	Возвращает следующую лексему в виде объекта типа <code>String</code>
<code>String nextToken(String <i>разделители</i>)</code>	Возвращает следующую лексему в виде объекта типа <code>Object</code> и задает символьную строку разделителей в соответствии со значением параметра <i>разделители</i>

Ниже приведен пример программы, в которой объект класса `StringTokenizer` создается для синтаксического анализа пар “ключ–значение”. Последовательный ряд пар “ключ–значение” разделяется точкой с запятой.

```
// Продемонстрировать применение класса StringTokenizer
import java.util.StringTokenizer;
class STDemo {
    static String in = "Название=Java. Полное руководство;" +
                      "Автор=Шилдт;" +
                      "Издательство=McGraw-Hill;" +
                      "Авторское право=2014";
    public static void main(String args[]) {
        StringTokenizer st = new StringTokenizer(in, ";");
        while(st.hasMoreTokens()) {
            String key = st.nextToken();
            String val = st.nextToken();
```

```

        System.out.println(key + "\t" + val);
    }
}

```

Эта программа выводит следующий результат:

```

Название Java. Полное руководство
Автор Шилдт
Издательство McGraw-Hill
Авторское право 2014

```

Класс BitSet

Этот класс служит для создания специального типа массива, содержащего битовые значения в виде логических значений. Размер массива типа `BitSet` можно, если требуется, увеличивать. Благодаря этому он становится похожим на битовый вектор. Ниже приведены конструкторы класса `BitSet`.

```

BitSet()
BitSet(int размер)

```

В первой форме конструктора создается объект по умолчанию. А вторая форма позволяет указать начальный размер (т.е. количество битов, которые можно сохранить). Все биты иницируются логическими значениями `false`.

В классе `BitSet` определяются методы, перечисленные в табл. 19.2.

Таблица 19.2. Методы из класса `BitSet`

Метод	Описание
<code>void and(BitSet множество_битов)</code>	Выполняет логическую операцию И над содержащим вызывающего объекта типа <code>BitSet</code> и заданного <code>множество_битов</code> . Результат размещается в вызывающем объекте
<code>void andNot(BitSet множество_битов)</code>	Для каждого бита, установленного в заданном <code>множество_битов</code> , сбрасывается соответствующий бит в вызывающем объекте типа <code>BitSet</code>
<code>int cardinality()</code>	Возвращает количество установленных битов в вызывающем объекте
<code>void clear()</code>	Устанавливает все биты в нуль
<code>void clear(int индекс)</code>	Устанавливает в нуль бит по указанному <code>индексу</code>
<code>void clear(int начальный_индекс, int конечный_индекс)</code>	Устанавливает в нуль биты от позиции <code>начальный_индекс</code> до позиции <code>конечный_индекс-1</code>
<code>Object clone()</code>	Дублирует вызывающий объект
<code>boolean equals(Object множество_битов)</code>	Возвращает логическое значение <code>true</code> , если вызывающее множество битов равнозначно заданному <code>множество_битов</code> , а иначе — логическое значение <code>false</code>
<code>void flip(int индекс)</code>	Обращает бит по указанному <code>индексу</code>

Метод	Описание
<code>void flip(int <i>начальный_индекс</i>, int <i>конечный_индекс</i>)</code>	Обращает биты от позиции <i>начальный_индекс</i> до позиции <i>конечный_индекс</i> -1
<code>boolean get(int <i>индекс</i>)</code>	Возвращает текущее значение бита по указанному <i>индексу</i>
<code>BitSet get(int <i>начальный_индекс</i>, int <i>конечный_индекс</i>)</code>	Возвращает объект типа BitSet , состоящий из бит от позиции <i>начальный_индекс</i> до позиции <i>конечный_индекс</i> -1. Вызывающий объект не изменяется
<code>int hashCode()</code>	Возвращает хеш-код вызывающего объекта
<code>boolean intersects(BitSet <i>множество_битов</i>)</code>	Возвращает логическое значение true , если установлена хотя бы одна пара соответствующих битов в вызывающем объекте и заданном <i>множестве_битов</i>
<code>boolean isEmpty()</code>	Возвращает логическое значение true , если все биты вызывающего объекта сброшены
<code>int length()</code>	Возвращает количество битов, требующихся для хранения содержимого вызывающего объекта типа BitSet . Это значение определяется по положению последнего установленного бита
<code>int nextClearBit(int <i>начальный_индекс</i>)</code>	Возвращает позицию следующего сброшенного бита (имеющего логическое значение false), начиная с указанной позиции <i>начальный_индекс</i>
<code>int nextSetBit(int <i>начальный_индекс</i>)</code>	Возвращает позицию следующего установленного бита (имеющего логическое значение true), начиная с указанной позиции <i>начальный_индекс</i> . Если ни один из битов не установлен, возвращается значение -1
<code>void or(BitSet <i>множество_битов</i>)</code>	Выполняет логическую операцию ИЛИ над содержимым вызывающего объекта класса BitSet и заданного <i>множества_битов</i> . Результат размещается в вызывающем объекте
<code>int previousClearBit(int <i>начальный_индекс</i>)</code>	Возвращает позицию следующего сброшенного бита (имеющего логическое значение false), расположенную до заданной позиции <i>начальный_индекс</i> включительно. Если сброшенный бит не найден, возвращается значение -1
<code>int previousSetBit(int <i>начальный_индекс</i>)</code>	Возвращает позицию следующего установленного бита (имеющего логическое значение true), расположенную до заданной позиции <i>начальный_индекс</i> включительно. Если установленный бит не найден, возвращается значение -1
<code>void set(int <i>индекс</i>)</code>	Устанавливает бит по указанному <i>индексу</i>
<code>void set(int <i>индекс</i>, boolean <i>v</i>)</code>	Устанавливает бит по указанному <i>индексу</i> равным заданному значению параметра <i>v</i> . Если этот параметр принимает логическое значение true , то бит устанавливается, а если он принимает логическое значение false , то бит сбрасывается

Окончание табл. 19.2

Метод	Описание
<code>void set(int <i>начальный_индекс</i>, int <i>конечный_индекс</i>)</code>	Устанавливает биты от позиции <i>начальный_индекс</i> до позиции <i>конечный_индекс</i> -1
<code>void set(int <i>начальный_индекс</i>, int <i>конечный_индекс</i>, boolean <i>v</i>)</code>	Устанавливает биты от позиции <i>начальный_индекс</i> до позиции <i>конечный_индекс</i> -1 равными заданному значению параметра <i>v</i> . Если этот параметр принимает логическое значение true , то биты устанавливаются, а если он принимает логическое значение false , то биты сбрасываются
<code>int size()</code>	Возвращает количество битов в вызывающем объекте типа BitSet
<code>IntStream stream()</code>	Возвращает поток ввода-вывода, содержащий позиции всех установленных битов: от младшего до старшего (добавлен в версии JDK 8)
<code>byte[] toByteArray()</code>	Возвращает массив типа byte , который содержит вызывающий объект типа BitSet
<code>long[] toLongArray()</code>	Возвращает массив типа long , который содержит вызывающий объект типа BitSet
<code>String toString()</code>	Возвращает строковый эквивалент вызывающего объекта типа BitSet
<code>static BitSet valueOf(byte[] <i>v</i>)</code>	Возвращает объект типа BitSet , содержащий биты из указанного массива <i>v</i>
<code>static BitSet valueOf(ByteBuffer <i>v</i>)</code>	Возвращает объект типа BitSet , содержащий биты из указанного буфера <i>v</i>
<code>static BitSet valueOf(long[] <i>v</i>)</code>	Возвращает объект типа BitSet , содержащий биты из указанного массива <i>v</i>
<code>static BitSet valueOf(LongBuffer <i>v</i>)</code>	Возвращает объект типа BitSet , содержащий биты из указанного буфера <i>v</i>
<code>void xor(BitSet <i>множество_битов</i>)</code>	Выполняет логическую операцию исключающее ИЛИ над содержимым вызывающего объекта типа BitSet и заданного <i>множество_битов</i> . Результат размещается в вызывающем объекте

Ниже приведен пример программы, демонстрирующий применение класса **BitSet**.

```
// Продемонстрировать применение класса BitSet
import java.util.BitSet;

class BitSetDemo {
    public static void main(String args[]) {
        BitSet bits1 = new BitSet(16);
        BitSet bits2 = new BitSet(16);

        // установить некоторые биты
        for(int i=0; i<16; i++) {
            if((i%2) == 0) bits1.set(i);
            if((i%5) != 0) bits2.set(i);
        }
    }
}
```

```

System.out.println(
    "Начальная комбинация битов в объекте bits1: ");
System.out.println(bits1);
System.out.println(
    * "\nНачальная комбинация битов в объекте bits2: ");
System.out.println(bits2);

// выполнить логическую операцию И над битами
bits2.and(bits1);
System.out.println("\nbits2 AND bits1: ");
System.out.println(bits2);

// выполнить логическую операцию ИЛИ над битами
bits2.or(bits1);
System.out.println("\nbits2 OR bits1: ");
System.out.println(bits2);

// выполнить логическую операцию исключающее ИЛИ над битами
bits2.xor(bits1);
System.out.println("\nbits2 XOR bits1: ");
System.out.println(bits2);
    }
}

```

Результат выполнения этой программы приведен ниже. Когда метод `toString()` преобразует объект типа `BitSet` в его строковый эквивалент, каждый установленный бит представлен номером своей позиции. Сброшенные биты не показаны.

Начальная комбинация битов в объекте `bits1`:
{0, 2, 4, 6, 8, 10, 12, 14}

Начальная комбинация битов в объекте `bits2`:
{1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14}

`bits2 AND bits1`:
{2, 4, 6, 8, 12, 14}

`bits2 OR bits1`:
{0, 2, 4, 6, 8, 10, 12, 14}

`bits2 XOR bits1`:
{}

Классы `Optional`, `OptionalDouble`, `OptionalInt` и `OptionalLong`

В версии JDK 8 внедрены классы `OptionalDouble`, `OptionalInt` и `OptionalLong`, предоставляющие изящный выход из положения в тех случаях, когда значение может отсутствовать. В прошлом для того чтобы обозначить отсутствие значения, обычно использовалось пустое значение `null`, но это могло привести к исключениям в связи с пустыми указателями при попытке разыменовать пустую ссылку. Поэтому во избежание подобных исключений требовались частые проверки на наличие пустого значения `null`. Классы, рассматриваемые в этом разделе, предоставляют более удобный способ разрешить данное затруднение.

Первым и самым общим из них является класс `Optional`. Поэтому именно ему и уделяется основное внимание в этом разделе. Ниже приведена общая форма его объявления.

```
class Optional<T>
```

Здесь `T` обозначает тип сохраняемого значения. Следует иметь в виду, что экземпляр класса `Optional` может содержать значение типа `T` или быть пустым. Иными словами, объект типа `Optional` совсем не обязательно должен содержать значение. У класса `Optional` вообще отсутствуют конструкторы, но в нем определяется ряд методов для обращения с объектами данного класса. С их помощью можно, например, определить наличие значения, получить значение, если оно присутствует, а если оно отсутствует — значение по умолчанию, и даже создать значение типа `Optional`. Методы, определенные в классе `Optional`, приведены в табл. 19.3.

Таблица 19.3. Методы из класса `Optional`

Метод	Описание
<code>static <T> Optional<T> empty()</code>	Возвращает объект, для которого метод <code>isPresent()</code> возвращает логическое значение false
<code>boolean equals(Object <i>необязательный</i>)</code>	Возвращает логическое значение true , если вызывающий объект равен объекту, определяемому параметром <i>необязательный</i> , а иначе — логическое значение false
<code>Optional<T> filter(Predicate<? super T> <i>условие</i>)</code>	Возвращает экземпляр класса <code>Optional</code> , содержащий такое же значение, как и у вызывающего объекта, если это значение удовлетворяет заданному <i>условию</i> , а иначе — пустой объект
<code>U Optional<U> flatMap(Function<? super T, Optional<U>> <i>отображающая функция</i>)</code>	Применяет заданную <i>отображающую функцию</i> к вызываемому объекту, если этот объект содержит значение, и возвращает полученный результат, а иначе — пустой объект
<code>T get()</code>	Возвращает значение в вызывающем объекте. Но если в нем отсутствует значение, то генерируется исключение типа <code>NoSuchElementException</code>
<code>int hashCode()</code>	Возвращает хеш-код вызывающего объекта
<code>void ifPresent(Consumer<? super T> <i>функция</i>)</code>	Вызывает заданную <i>функцию</i> , если в вызывающем объекте присутствует значение, передавая этот объект вызываемой <i>функции</i> . В отсутствие значения никаких действий не производится
<code>boolean isPresent()</code>	Возвращает логическое значение true , если вызывающий объект содержит значение, а в отсутствие значения — логическое значение false
<code>U Optional<U> map(Function<? super T, ? extends U>> <i>отображающая функция</i>)</code>	Применяет заданную <i>отображающую функцию</i> к вызываемому объекту, если этот объект содержит значение, и возвращает полученный результат, а иначе — пустой объект

Метод	Описание
static <T> Optional <T> of (T <i>значение</i>)	Создает экземпляр класса Optional , содержащий заданное <i>значение</i> и возвращает полученный результат. Заданное <i>значение</i> не должно быть пустым (null)
static <T> Optional <T> ofNullable (T <i>значение</i>)	Создает экземпляр класса Optional , содержащий заданное <i>значение</i> и возвращает полученный результат. Но если <i>значение</i> задано пустым, то возвращается пустой экземпляр класса Optional
T orElse (T <i>определяющее_значение</i>)	Если вызывающий объект содержит значение, то возвращается именно оно, а иначе — заданное <i>определяющее_значение</i>
T orElseGet (Supplier<? extends T> <i>функция_получения</i>)	Если вызывающий объект содержит значение, то возвращается именно оно, а иначе — значение, получаемое из заданной <i>функции_получения</i>
<X extends Throwable> T orElseThrow (Supplier<? extends X> <i>функция_исключения</i>) throws X extends Throwable	Возвращает значение, содержащееся в вызывающем объекте. Но если такое значение отсутствует, то заданная <i>функция_исключения</i> генерирует исключение
String toString ()	Возвращает строковое представление вызывающего объекта

Понять принцип действия класса **Optional** лучше всего на конкретном примере, в котором демонстрируется применение основных методов этого класса. Основными в классе **Optional** являются методы **isPresent()** и **get()**. Для того чтобы выяснить, присутствует ли в экземпляре класса **Optional** значение, достаточно вызвать метод **isPresent()**. Если значение присутствует, этот метод возвращает логическое значение **true**, а иначе — логическое значение **false**. Итак, если значение присутствует в экземпляре класса **Optional**, его можно получить, вызвав метод **get()**. Но если вызвать метод **get()** для объекта, не содержащего значение, то генерируется исключение типа **NoSuchElementException**. Именно поэтому следует сначала убедиться в наличии значения, а затем вызывать метод **get()** для объекта типа **Optional**.

Безусловно, необходимость вызывать два метода для получения искомого значения влечет за собой дополнительные издержки на доступ к этому значению. Правда, в классе **Optional** определяются методы, сочетающие в себе функции проверки и извлечения искомого значения. Одним из них является метод **orElse()**. Так, если объект, для которого этот метод вызывается, содержит значение, то возвращается именно оно, а иначе — значение по умолчанию.

Как упоминалось выше, в классе **Optional** конструкторы не определены. Вместо этого для создания экземпляра данного класса вызываются его методы. Например, создать экземпляр класса **Optional**, содержащий конкретное значение, можно, вызвав метод **of()**. А для того чтобы создать экземпляр класса **Optional**, не содержащий конкретное значение, достаточно вызвать метод **empty()**.

В следующем примере программы демонстрируется применение упомянутых выше методов:

```
// Продемонстрировать применение нескольких
// методов из обобщенного класса Optional<T>

import java.util.*;

class OptionalDemo {
    public static void main(String args[]) {

        Optional<String> noVal = Optional.empty();

        Optional<String> hasVal = Optional.of("ABCDEFGH");

        if(noVal.isPresent()) System.out.println("Не подлежит выводу");
        else System.out.println("Объект noVal не содержит значение");

        if(hasVal.isPresent()) System.out.println(
            "Объект hasVal содержит следующую строку: " + hasVal.get());

        String defStr = noVal.orElse("Строка по умолчанию");
        System.out.println(defStr);
    }
}
```

Ниже приведен результат, выводимый данной программой.

```
Объект noVal не содержит значение
Объект hasVal содержит следующую строку: ABCDEFGH
Строка по умолчанию
```

Как следует из приведенного выше результата выполнения данной программы, значение может быть получено из объекта типа `Optional` только в том случае, если он присутствует в нем. Этот элементарный механизм позволяет предотвратить исключения, возникающие в классе `Optional` в связи с пустыми указателями.

Аналогичным образом действуют классы `OptionalDouble`, `OptionalInt` и `OptionalLong`, за исключением того что они специально предназначены для обработки значений типа `double`, `int` и `long` соответственно. И для этой цели вместо метода `get()` в них определены методы `getAsDouble()`, `getAsInt()` и `getAsLong()` соответственно. Но в то же время в них не поддерживаются методы `filter()`, `ofNullable()`, `map()` и `flatMap()`.

Класс Date

Класс `Date` инкапсулирует текущую дату и время. Прежде чем рассматривать класс `Date`, следует заметить, что этот класс ощутимо изменился по сравнению с его первоначальным вариантом, определенным в версии Java 1.0. Когда была выпущена версия Java 1.1, многие функции исходного класса `Date` были перемещены в классы `Calendar` и `DateFormat` и, как следствие, многие исходные методы класса `Date` из версии Java 1.0 стали не рекомендованными к употреблению в новом коде, поэтому они здесь и не обсуждаются. В классе `Date` поддерживаются следующие конструкторы:

```
Date()
Date(long миллисекунд)
```

Первый конструктор инициализирует объект текущими датой и временем. А второй конструктор принимает один аргумент, обозначающий количество миллисекунд, прошедших с полуночи 1 января 1970 г. Методы из класса `Date`, рекомендованные к употреблению, перечислены в табл. 19.4. Класс `Date` реализует также интерфейс `Comparable`.

Таблица 19.4. Методы из класса `Date`, рекомендованные к употреблению

Метод	Описание
<code>boolean after(Date data)</code>	Возвращает логическое значение true , если вызывающий объект типа Date содержит более позднюю дату, чем указанная <i>data</i> , а иначе — логическое значение false
<code>boolean before(Date data)</code>	Возвращает логическое значение true , если вызывающий объект типа Date содержит более раннюю дату, чем указанная <i>data</i> , а иначе — логическое значение false
<code>Object clone()</code>	Дублирует вызывающий объект типа Date
<code>int compareTo(Date data)</code>	Сравнивает значение вызывающего объекта с указанной датой. Возвращает нулевое значение, если сравниваемые значения равны; отрицательное значение, если вызывающий объект содержит более раннюю дату, чем указанная <i>data</i> ; или положительное значение, если вызывающий объект содержит более позднюю дату
<code>boolean equals(Object data)</code>	Возвращает логическое значение true , если вызывающий объект типа Date содержит те же самые время и дату, что и указанная <i>data</i> , а иначе — логическое значение false
<code>long getTime()</code>	Возвращает количество миллисекунд, прошедших с полуночи 1 января 1970 г.
<code>int hashCode()</code>	Возвращает хеш-код вызывающего объекта
<code>void setTime(long время)</code>	Устанавливает время и дату в соответствии с параметром <i>время</i> , который обозначает количество миллисекунд, прошедших с полуночи 1 января 1970 г.
<code>String toString()</code>	Преобразует вызывающий объект типа Date в символьную строку и возвращает результат

Как следует из табл. 19.4, новые средства класса `Date` не позволяют получать составляющие даты и времени по отдельности. И как демонстрируется в приведенном ниже примере программы, получить можно только дату и время в миллисекундах или в строковом представлении по умолчанию, возвращаемом методом `toString()`. А для того чтобы получить более подробные сведения о времени и дате, следует воспользоваться средствами описываемого далее класса `Calendar`.

```
// Вывести дату и время, используя только методы из класса Date
import java.util.Date;
```

```
class DateDemo {
    public static void main(String args[]) {

        // создать объект типа Date
        Date date = new Date();

        // вывести дату и время методом toString()
        System.out.println(date);

        // вывести количество миллисекунд, прошедших
        // с 1 января 1970 г. по Гринвичу
        long msec = date.getTime();
        System.out.println("Количество миллисекунд, прошедших с " +
                           "1 января 1970 г. по Гринвичу = " + msec);
    }
}
```

Ниже приведен пример выполнения данной программы.

```
Sat Jan 01 10:27:33 CST 2011
Количество миллисекунд, прошедших с 1 января 1970 г.
по Гринвичу = 1293899253417
```

Класс Calendar

Абстрактный класс `Calendar` предоставляет ряд методов, позволяющих преобразовывать время в миллисекундах в целый ряд удобных составляющих. К примерам видов предоставляемой информации о дате и времени относятся год, месяц, день, часы, минуты и секунды. Назначение класса `Calendar` — обеспечить своим подклассам конкретные функциональные возможности для интерпретации информации о дате и времени по их собственным правилам. Это еще одна особенность библиотеки классов Java, которая позволяет писать программы, способные работать в разных интернациональных средах. Примером такого подкласса может служить класс `GregorianCalendar`.

На заметку! Для интерпретации даты и времени в версии JDK 8 определен новый прикладной программный интерфейс API, входящий в состав пакета `java.time` и рассматриваемый в главе 30. Им можно пользоваться при разработке новых прикладных программ на Java.

У класса `Calendar` отсутствуют открытые конструкторы. В этом классе определяется ряд защищенных переменных экземпляра. В частности, переменная `areFieldsSet` содержит значение типа `boolean`, обозначающее, были ли установлены составляющие времени. Переменная `fields` содержит массив целочисленных значений, обозначающих составляющие времени, а переменная `isSet` — массив значений типа `boolean`, обозначающих, была ли установлена конкретная составляющая времени. Переменная `time` типа `long` содержит текущее время для данного объекта. И наконец, переменная `isTimeSet` содержит значение типа `boolean`, указывающее на то, что было установлено текущее время.

Некоторые наиболее употребительные методы из класса `Calendar` приведены в табл. 19.5.

Таблица 19.5. Наиболее употребительные методы из класса **Calendar**

Метод	Описание
abstract void add(int составляющая, int значение)	Складывает заданное <i>значение</i> с указанной <i>составляющей</i> даты или времени. Чтобы отнять заданное <i>значение</i> , его следует указать отрицательным. В качестве параметра <i>составляющая</i> может быть указано одно из полей, определенных в классе Calendar , например Calendar.HOUR
boolean after(Object календарный_объект)	Возвращает логическое значение true , если вызывающий объект типа Calendar содержит более позднюю дату, чем указанный <i>календарный_объект</i> , а иначе — логическое значение false
boolean before(Object календарный_объект)	Возвращает логическое значение true , если вызывающий объект типа Calendar содержит более раннюю дату, чем указанный <i>календарный_объект</i> , а иначе — логическое значение false
final void clear()	Обнуляет все составляющие времени в вызываемом объекте
final void clear(int составляющая)	Обнуляет указанную <i>составляющую</i> времени в вызываемом объекте
Object clone()	Возвращает дубликат вызываемого объекта
boolean equals(Object календарный_объект)	Возвращает логическое значение true , если вызывающий объект типа Calendar содержит такую же дату, как и заданный <i>календарный_объект</i> , а иначе — логическое значение false
int get(int календарное_поле)	Возвращает значение одной из составляющих вызываемого объекта. Эта составляющая обозначается параметром <i>календарное_поле</i> . К примерам составляющих, которые можно запросить, относятся значения в полях Calendar.YEAR , Calendar.MONTH , Calendar.MINUTE и т.п.
static Locale[] getAvailableLocales()	Возвращает массив объектов типа Locale , содержащий региональные настройки, для которых в системе доступны календари
static Calendar getInstance()	Возвращает объект типа Calendar для региональных настроек и часового пояса по умолчанию
static Calendar getInstance(TimeZone часовой_пояс)	Возвращает объект типа Calendar для региональных настроек по умолчанию и указанного <i>часового_пояса</i>
static Calendar getInstance(Locale региональные_настройки)	Возвращает объект типа Calendar для указанных <i>региональных_настроек</i> и часового пояса по умолчанию
static Calendar getInstance(TimeZone часовой_пояс, Locale региональные_настройки)	Возвращает объект типа Calendar для указанных <i>региональных_настроек</i> и <i>часового_пояса</i>
final Date getTime()	Возвращает объект типа Date , содержащий такое же время, как и время вызываемого объекта

Окончание табл. 19.5

Метод	Описание
TimeZone getTimeZone()	Возвращает часовой пояс вызывающего объекта
final boolean isSet(int <i>составляющая</i>)	Возвращает логическое значение true , если указанная <i>составляющая</i> времени установлена, а иначе — логическое значение false
void set(int <i>составляющая</i> , int <i>значение</i>)	Устанавливает заданное <i>значение</i> в указанной <i>составляющей</i> даты или времени вызывающего объекта. Параметр <i>составляющая</i> должен иметь значение одного из полей класса Calendar , например Calendar.HOUR
final void set(int <i>год</i> , int <i>месяц</i> , int <i>день_месяца</i>)	Устанавливает в вызывающем объекте различные составляющие даты и времени
final void set(int <i>год</i> , int <i>месяц</i> , int <i>день_месяца</i> , int <i>час</i> , int <i>минута</i>)	Устанавливает в вызывающем объекте различные составляющие даты и времени
final void set(int <i>год</i> , int <i>месяц</i> , int <i>день_месяца</i> , int <i>час</i> , int <i>минута</i> , int <i>секунда</i>)	Устанавливает в вызывающем объекте различные составляющие даты и времени
final void setTime(Date <i>d</i>)	Устанавливает в вызывающем объекте различные составляющие даты и времени. Нужные сведения получаются из заданного объекта <i>d</i> типа Date
void setTimeZone(TimeZone <i>часовой_пояс</i>)	Устанавливает указанный <i>часовой_пояс</i> для вызывающего объекта
final Instant toInstant()	Возвращает объект типа Instant , соответствующий вызывающему объекту типа Calendar (добавлен в версии JDK 8)

В классе **Calendar** определены перечисленные ниже целочисленные константы, применяемые при получении или установке составляющих календаря. (Константы с суффиксом **FORMAT** или **STANDALONE** были внедрены в версии JDK 8.)

ALL STYLES	HOURL OF DAY	PM
AM	JANUARY	SATURDAY
AM PM	JULY	SECOND
APRIL	JUNE	SEPTEMBER
AUGUST	LONG	SHORT
DATE	LONG FORMAT	SHORT FORMAT
DAY OF MONTH	LONG STANDALONE	SHORT STANDALONE
DAY OF WEEK	MARCH	SUNDAY
DAY OF WEEK IN MONTH	MAY	THURSDAY
DAY OF YEAR	MILLISECOND	TUESDAY
DECEMBER	MINUTE	UNDECIMBER
DST OFFSET	MONDAY	WEDNESDAY
ERA	MONTH	WEEK OF MONTH
FEBRUARY	NARROW FORMAT	WEEK OF YEAR
FIELD COUNT	NARROW STANDALONE	YEAR
FRIDAY	NOVEMBER	ZONE OFFSET
HOURL	OCTOBER	

В следующем примере программы демонстрируется применение некоторых методов из класса `Calendar`:

```
// Продемонстрировать применение класса Calendar
import java.util.Calendar;

class CalendarDemo {
    public static void main(String args[]) {
        String months[] = {
            "Jan", "Feb", "Mar", "Apr",
            "May", "Jun", "Jul", "Aug",
            "Sep", "Oct", "Nov", "Dec"};

        // создать календарь, инициализируемый
        // текущими датой и временем с учетом региональных
        // настроек и часового пояса по умолчанию
        Calendar calendar = Calendar.getInstance();

        // вывести текущие дату и время
        System.out.print("Дата: ");
        System.out.print(months[calendar.get(Calendar.MONTH)]);
        System.out.print(" " + calendar.get(Calendar.DATE) + " ");
        System.out.println(calendar.get(Calendar.YEAR));
        System.out.print("Время: ");
        System.out.print(calendar.get(Calendar.HOUR) + ":");
        System.out.print(calendar.get(Calendar.MINUTE) + ":");
        System.out.println(calendar.get(Calendar.SECOND));

        // установить дату и время и вывести их
        calendar.set(Calendar.HOUR, 10);
        calendar.set(Calendar.MINUTE, 29);
        calendar.set(Calendar.SECOND, 22);

        System.out.print("Измененное время: ");
        System.out.print(calendar.get(Calendar.HOUR) + ":");
        System.out.print(calendar.get(Calendar.MINUTE) + ":");
        System.out.println(calendar.get(Calendar.SECOND));
    }
}
```

Ниже приведен пример выполнения данной программы.

```
Дата: Jan 1 2014
Время: 11:29:39
Измененное время: 10:29:22
```

Класс `GregorianCalendar`

Класс `GregorianCalendar` служит в качестве конкретной реализации привычного григорианского календаря средствами класса `Calendar`. Метод `getInstance()` из класса `Calendar` обычно возвращает объект типа `GregorianCalendar`, инициализируемый текущими датой и временем с учетом региональных настроек и часового пояса по умолчанию.

В классе `GregorianCalendar` определяются два поля, `AD` и `BC`, обозначающих две эры, определенные в григорианском календаре. Имеется также ряд конструкторов для построения объектов класса `GregorianCalendar`. В частности, кон-

конструктор по умолчанию `GregorianCalendar()` инициализирует объект данного класса текущими временем и датой с учетом региональных настроек и часового пояса по умолчанию. Другие конструкторы класса `GregorianCalendar` перечислены ниже по мере увеличения уровня их специализации.

```
GregorianCalendar(int год, int месяц, int день_месяца)
GregorianCalendar(int год, int месяц, int день_месяца, int час,
                  int минута)
GregorianCalendar(int год, int месяц, int день_месяца, int час,
                  int минута, int секунда)
```

Во всех трех формах конструкторов устанавливаются параметры `день_месяца`, `месяц` и `год`, причем нулевое значение параметра `месяц` обозначает январь. В первой форме конструктора время устанавливается в полночь, во второй форме — в часах и минутах, а в третьей форме — еще и в секундах.

Имеется также возможность создать объект типа `GregorianCalendar`, указав региональные настройки и/или часовой пояс. Следующие конструкторы создают объекты, инициализируемые текущими временем и датой с учетом заданных региональных настроек и часового пояса.

```
GregorianCalendar(Locale региональные_настройки)
GregorianCalendar(TimeZone часовой_пояс)
GregorianCalendar(TimeZone часовой_пояс, Locale региональные_настройки)
```

В классе `GregorianCalendar` предоставляется реализация абстрактных методов из класса `Calendar`. Кроме того, в нем определены некоторые дополнительные методы. Самым интересным из них, вероятно, является метод `isLeapYear()`, проверяющий, является ли год високосным. Ниже приведена его общая форма.

```
boolean isLeapYear(int год)
```

Этот метод возвращает логическое значение `true`, если указанный год является високосным, а иначе — логическое значение `false`. В версии JDK 8 класс `GregorianCalendar` дополнен следующими методами: `from()` и `toZonedDateTime()`, поддерживающими новый прикладной программный интерфейс API даты и времени, а также метод `getCalendarType()`, возвращающий тип календаря в виде символьной строки (в данном случае — "gregory", т.е. григорианский календарь).

В следующем примере программы демонстрируется применение класса `GregorianCalendar`.

```
// Продемонстрировать применение класса типа GregorianCalendar
import java.util.*;
```

```
class GregorianCalendarDemo {
    public static void main(String args[]) {
        String months[] = {
            "Jan", "Feb", "Mar", "Apr",
            "May", "Jun", "Jul", "Aug",
            "Sep", "Oct", "Nov", "Dec"};
        int year;

        // создать григорианский календарь, инициализируемый
        // текущими датой и временем с учетом региональных
        // настроек и часового пояса по умолчанию
        GregorianCalendar gcalendar = new GregorianCalendar();
```



```
// вывести текущие время и дату
System.out.print("Дата: ");
System.out.print(months[gcalendar.get(Calendar.MONTH)]);
System.out.print(" " + gcalendar.get(Calendar.DATE) + " ");
System.out.println(year = gcalendar.get(Calendar.YEAR));

System.out.print("Время: ");
System.out.print(gcalendar.get(Calendar.HOUR) + ":");
System.out.print(gcalendar.get(Calendar.MINUTE) + ":");
System.out.println(gcalendar.get(Calendar.SECOND));

// проверить, является ли текущий год високосным
if(gcalendar.isLeapYear(year)) {
    System.out.println("Текущий год високосный");
}
else {
    System.out.println("Текущий год не високосный");
}
}
```

Ниже приведен пример выполнения данной программы.

Дата: Jan 1 2014

Время: 1:45:5

Текущий год не високосный

Класс `TimeZone`

Еще одним, имеющим отношение ко времени является класс `TimeZone`. Он позволяет оперировать временем с учетом часовых поясов, смещенных относительно среднего времени по Гринвичу (GMT), называемого также *универсальным скоординированным временем* (UCT). В классе `TimeZone` учитывается также летнее время. В нем поддерживается только конструктор по умолчанию. Избранные методы, определенные в классе `TimeZone`, перечислены в табл. 19.6.

Таблица 19.6. Избранные методы из класса `TimeZone`

Метод	Описание
<code>Object clone()</code>	Возвращает конкретный вариант метода <code>clone()</code> для класса <code>TimeZone</code>
<code>static String[] getAvailableIDs()</code>	Возвращает массив символьных строк, представляющих названия всех часовых поясов
<code>static String[] getAvailableIDs(int <i>разница_во_времени</i>)</code>	Возвращает массив символьных строк, представляющих названия всех часовых поясов, имеющих заданную <i>разницу_во_времени</i> относительно среднего времени по Гринвичу
<code>static TimeZone getDefault()</code>	Возвращает объект типа <code>TimeZone</code> , представляющий часовой пояс по умолчанию, используемый на главном компьютере
<code>String getID()</code>	Возвращает имя вызывающего объекта типа <code>TimeZone</code>

Окончание табл. 19.6

Метод	Описание
abstract int getOffset(int <i>эра</i>, int <i>год</i>, int <i>месяц</i>, int <i>день_месяца</i>, int <i>день_недели</i>, int <i>миллисекунд</i>)	Возвращает смещение, которое должно быть добавлено к среднему времени по Гринвичу, чтобы рассчитать местное время. Рассчитанное время корректируется с учетом летнего времени. Параметры данного метода задают составляющие даты и времени
abstract int getRawOffset()	Возвращает исходное смещение (в миллисекундах), которое должно быть добавлено к среднему времени по Гринвичу, чтобы рассчитать местное время. Рассчитанное время не корректируется с учетом летнего времени
static TimeZone getTimeZone(String <i>название_часового_пояса</i>)	Возвращает объект типа TimeZone для заданного <i>названия_часового_пояса</i>
abstract boolean inDaylightTime(Date <i>d</i>)	Возвращает логическое значение true , если заданная дата <i>d</i> представлена в вызывающем объекте с учетом летнего времени, а иначе — логическое значение false
static void setDefault(TimeZone <i>часовой_пояс</i>)	Устанавливает <i>часовой_пояс</i> по умолчанию для данного главного компьютера, где <i>часовой_пояс</i> — ссылка на используемый объект типа TimeZone
void setID(String <i>название_часового_пояса</i>)	Устанавливает <i>название_часового_пояса</i> (т.е. его идентификатор)ы
abstract void setRawOffset(int <i>миллисекунд</i>)	Устанавливает смещение относительно среднего времени по Гринвичу в <i>миллисекундах</i>
ZoneId toZoneId()	Преобразует вызывающий объект в объект класса ZoneId и возвращает результат. Класс ZoneId входит в состав пакета java.time (добавлен в версии JDK 8)
abstract boolean useDaylight- Time()	Возвращает логическое значение true , если в вызывающем объекте учитывается летнее время, а иначе — логическое значение false

Класс SimpleTimeZone

Класс SimpleTimeZone является служебным подклассом, производным от класса TimeZone. Он реализует абстрактные методы из класса TimeZone и позволяет работать с часовыми поясами для григорианского календаря. В этом классе учитывается также летнее время.

В классе SimpleTimeZone определяются четыре конструктора. Общая форма объявления первого из них приведена ниже.

```
SimpleTimeZone(int разница_во_времени, String название_часового_пояса)
```

Этот конструктор создает объект типа SimpleTimeZone. Здесь параметр *разница_во_времени* обозначает смещение относительно среднего времени по Гринвичу, а па-

параметр *название_часового_пояса* — конкретное название часового пояса. Второй конструктор данного класса объявляется следующим образом:

```
SimpleTimeZone(int разница_во_времени,
               String идентификатор_часового_пояса,
               int месяц0, int дни_месяца0,
               int день0, int время0, int месяц1,
               int дни_месяца1, int, день1, int время1)
```

Здесь смещение среднего времени по Гринвичу задается параметром *разница_во_времени*, а конкретное название часового пояса — параметром *идентификатор_часового_пояса*. Начало действия летнего времени определяется параметрами *месяц0*, *дни_месяца0*, *день0* и *время0*, а окончание действия летнего времени — параметрами *месяц1*, *дни_месяца1*, *день1* и *время1*.

Третий конструктор класса SimpleTimeZone объявляется таким образом:

```
SimpleTimeZone(int разница_во_времени,
               String идентификатор_часового_пояса,
               int месяц0, int дни_месяца0, int день0,
               int время0, int месяц1, int дни_месяца1,
               int день1, int время1, int разница_в_летнее_время)
```

где параметр *разница_в_летнее_время* обозначает количество миллисекунд, сэкономленных в течение летнего времени. И наконец, четвертый конструктор класса SimpleTimeZone объявляется следующим образом:

```
SimpleTimeZone(int разница_во_времени,
               String идентификатор_часового_пояса,
               int месяц0, int дни_месяца0, int день0,
               int время0, int режим_времени0, int месяц1,
               int дни_месяца1, int день1, int время1,
               int режим_времени1, int разница_в_летнее_время)
```

где параметр *режим_времени0* обозначает режим начального времени, а параметр *режим_времени1* — режим конечного времени. Ниже перечислены допустимые значения этих режимов в виде констант.

STANDARD_TIME	WALL_TIME	UTC_TIME
---------------	-----------	----------

Режим времени определяет, каким образом интерпретируются величины времени. По умолчанию во всех остальных конструкторах используется режим времени, обозначаемый константой WALL_TIME.

Класс Locale

Класс Locale предназначен для создания объектов, каждый из которых описывает географический или культурный регион. Это один из нескольких классов, обеспечивающих возможность написания программ для выполнения в интернациональных средах. Например, форматы, применяемые для отображения даты, времени и чисел, в разных регионах отличаются.

Вопросы интернационализации программ на Java выходят за рамки рассмотрения данной книги. Но для интернационализации большинства прикладных программ требуются только самые элементарные действия, включая установку текущих региональных настроек.

В классе `Locale` определяются приведенные ниже константы, удобные для обращения с наиболее часто употребляемыми региональными настройками. Например, выражение `Locale.CANADA` обозначает объект типа `Locale` для Канады.

CANADA	GERMAN	KOREAN
CANADA_FRENCH	GERMANY	PRC
CHINA	ITALIAN	SIMPLIFIED_CHINESE
CHINESE	ITALY	TAIWAN
ENGLISH	JAPAN	TRADITIONAL_CHINESE
FRANCE	JAPANESE	UK
FRENCH	KOREA	US

Ниже перечислены конструкторы класса `Locale`.

```
Locale(String язык)
Locale(String язык, String страна)
Locale(String язык, String страна, String вариант)
```

Эти конструкторы создают объект типа `Locale` для представления конкретного языка, а два последних конструктора — еще и страны. Эти значения должны содержать стандартные коды стран и языков. В качестве параметра *вариант* могут быть предоставлены различные вспомогательные сведения.

В классе `Locale` определяется ряд методов. К числу самых важных относится метод `setDefault()`, общая форма которого приведена ниже.

```
static void setDefault(Locale объект_региональных_настроек)
```

Этот метод устанавливает региональные настройки, используемые по умолчанию в виртуальной машине JVM и обозначаемые параметром *объект_региональных_настроек*. Ниже приведены общие формы объявления других интересных методов из данного класса.

```
final String getDisplayCountry()
final String getDisplayLanguage()
final String getDisplayName()
```

Эти методы возвращают удобочитаемые символьные строки, которые можно использовать для отображения названий стран, языков и полного описания региональных настроек.

Региональные настройки по умолчанию можно получить, вызвав приведенный ниже метод `getDefault()`.

```
static Locale getDefault()
```

В версии JDK 7 были внесены существенные изменения в класс `Locale`, который с тех пор поддерживает стандарт Internet Engineering Task Force (IETF) BCP 47, определяющий дескрипторы для идентификации языков, а также стандарт Unicode Technical Standard (UTS) 35, определяющий язык разметки региональных данных (LSML). Для поддержки стандартов BCP 47 и UTS 35 в класс `Locale` пришлось ввести ряд средств, в том числе несколько новых методов и класс `Locale.Builder`. К их числу относится новый метод `getScript()`, получающий

сценарий региональных настроек, а также метод `toLanguageTag()`, получающий символьную строку с дескриптором языка в региональных настройках. В классе `Locale.Builder` создаются экземпляры класса `Locale`. Этим гарантируется, что спецификация региональных настроек будет сформирована правильно по стандарту ВСР 47. (Конструкторы класса `Locale` не обеспечивают такую проверку.) Ряд новых методов был введен в класс `Locale` и в версии JDK 8. К их числу относятся методы, поддерживающие фильтрацию, обработку исключений и поиск.

Классы `Calendar` и `GregorianCalendar` служат примерами классов, действующих с учетом региональных настроек. Классы `DateFormat` и `SimpleDateFormat` также зависят от региональных настроек.

Класс Random

Класс `Random` служит в качестве генератора псевдослучайных чисел. Они называются псевдослучайными, поскольку представляют собой сложные распределенные последовательности. В классе `Random` определяются следующие конструкторы:

```
Random()  
Random(long начальное_число)
```

В первой форме конструктора создается генератор псевдослучайных чисел, использующий однозначное начальное число. А во второй форме это число можно указать вручную.

Инициализировав объект типа `Random` начальным числом, можно определить начальную точку для генерирования последовательности случайных чисел. Если использовать то же самое начальное число для инициализации другого объекта типа `Random`, то получится та же самая последовательность случайных чисел. Если же требуется получить разные последовательности случайных чисел, объекты типа `Random` следует инициализировать разными начальными числами. С этой целью можно, в частности, воспользоваться текущим временем, чтобы инициализировать им объект типа `Random`. Благодаря этому уменьшается вероятность получения повторяющихся последовательностей случайных чисел.

Основные открытые методы из класса `Random` перечислены в табл. 19.7. Эти методы уже давно доступны в классе `Random`, начиная с версии Java 1.0, и поэтому они нашли широкое применение в прикладном коде.

Как видите, имеется семь типов случайных чисел, которые можно извлечь из объекта типа `Random`. Логические случайные значения можно получить с помощью метода `nextBoolean()`, случайные байты — с помощью метода `nextBytes()`, а целые случайные числа — с помощью метода `nextInt()`. Случайные длинные целочисленные значения, равномерно распределенные по диапазону допустимых значений, выдает метод `nextLong()`, тогда как методы `nextFloat()` и `nextDouble()` возвращают случайные значения типа `float` и `double`, равномерно распределенные в пределах от 0,0 до 1,0. И наконец, метод `nextGaussian()` возвращает значение типа `double` со стандартным отклонением на 1,0 от центральной точки 0,0. Это так называемая *кривая нормального распределения*.

Таблица 19.7. Основные методы из класса Random

Метод	Описание
<code>boolean nextBoolean()</code>	Возвращает следующее случайное числовое значение типа <code>boolean</code>
<code>void nextBytes(byte значения[])</code>	Заполняет массив <i>значения</i> случайно сгенерированными числовыми значениями
<code>double nextDouble()</code>	Возвращает следующее случайное числовое значение типа <code>double</code>
<code>float nextFloat()</code>	Возвращает следующее случайное числовое значение типа <code>float</code>
<code>double nextGaussian()</code>	Возвращает следующее случайное числовое значение из нормального распределения
<code>int nextInt()</code>	Возвращает следующее случайное числовое значение типа <code>int</code>
<code>int nextInt(int n)</code>	Возвращает следующее случайное числовое значение типа <code>int</code> в пределах от 0 до <i>n</i>
<code>long nextLong()</code>	Возвращает следующее случайное числовое значение типа <code>long</code>
<code>void setSeed(long начальное_число)</code>	Устанавливает <i>начальное_число</i> (т.е. начальную точку для генерирования случайных чисел)

Ниже приведен пример программы, демонстрирующий случайную последовательность, формируемую методом `nextGaussian()`, получающим 100 случайных значений из нормального распределения и усредняющим их. В данной программе подсчитывается также количество значений, попадающих в пределы двух стандартных отклонений с шагом 0,5 в положительную или отрицательную сторону в каждом случае. Получаемый результат графически отображается боком на экране.

```
// Продемонстрировать генерирование случайных значений
// с нормальным распределением
import java.util.Random;
class RandDemo {
    public static void main(String args[]) {
        Random r = new Random();
        double val;
        double sum = 0;
        int bell[] = new int[10];

        for(int i=0; i<100; i++) {
            val = r.nextGaussian();
            sum += val;
            double t = -2;

            for(int x=0; x<10; x++, t += 0.5)
                if(val < t) {
                    bell[x]++;
                    break;
                }
        }
        System.out.println("Среднее всех значений: " + (sum/100));

        // вывести кривую распределения
        for(int i=0; i<10; i++) {
            for(int x=bell[i]; x>0; x--)
                System.out.print("*");
        }
    }
}
```

```

        System.out.println();
    }
}

```

Ниже приведен пример выполнения данной программы. Как видите, получается колоколоподобное распределение чисел.

```

Среднее всех значений: 0.0702235271133344
**
*****
*****
*****
*****
*****
*****
*****
*****
*****
***

```

В версии JDK 8 класс `Random` был дополнен тремя новыми методами `doubles()`, `ints()` и `longs()`, поддерживающими новый прикладной программный интерфейс API потоков данных, рассматриваемый в главе 29. Каждый из них возвращает ссылку на поток данных, содержащий последовательность псевдослучайных числовых значений указанного типа. Для каждого из этих методов определяется несколько перегружаемых вариантов. Ниже приведены простейшие формы объявления этих методов.

```

DoubleStream doubles()
IntStream ints()
LongStream longs()

```

Метод `doubles()` возвращает поток данных, содержащий псевдослучайные числовые значения типа `double` в пределах от 0,0 до 1,0. Метод `ints()` возвращает поток данных, содержащий псевдослучайные числовые значения типа `int`, а метод `longs()` — поток данных с псевдослучайными числовыми значениями типа `long`. Но в любом случае поток данных оказывается, по существу, бесконечным. Для каждого из этих методов определено несколько перегружаемых вариантов, позволяющих задавать размер потока данных, начало отсчета и верхнюю границу.

Класс `Observable`

Класс `Observable` служит для создания подклассов, за которыми могут наблюдать остальные части прикладной программы. Когда в объекте такого подкласса происходят изменения, об этом извещаются наблюдающие классы. Наблюдающие классы должны реализовать интерфейс `Observer`, в котором определен метод `update()`. Этот метод вызывается, когда наблюдатель получает извещение об изменении наблюдаемого объекта.

В классе `Observable` определяются методы, приведенные в табл. 19.8. Наблюдаемый объект должен следовать двум простым правилам. Во-первых, если он изменяется, то должен вызывать метод `setChanged()`. Во-вторых, когда он

готов известить наблюдателей об этом изменении, то должен вызвать метод `notifyObservers()`. Это вынуждает наблюдающий объект (или объекты) вызывать метод `update()`. Следует, однако, иметь в виду, что если объект обращается к методу `notifyObservers()`, не вызвав предварительно метод `setChanged()`, то никакого действия не последует. Наблюдаемый объект должен вызвать оба метода, `setChanged()` и `notifyObservers()`, прежде чем будет вызван метод `update()`.

Таблица 19.8. Методы из класса `Observable`

Метод	Описание
<code>void addObserver(Observer объект)</code>	Вводит заданный объект в список объектов, наблюдающих за вызывающим объектом
<code>protected void clearChanged()</code>	Возвращает неизменное состояние вызывающего объекта
<code>int countObservers()</code>	Возвращает количество объектов, наблюдающих за текущим объектом
<code>void deleteObserver(Observer объект)</code>	Удаляет заданный объект из списка объектов, наблюдающих за вызывающим объектом
<code>void deleteObservers()</code>	Удаляет все наблюдатели вызывающего объекта
<code>boolean hasChanged()</code>	Возвращает логическое значение true , если вызывающий объект был изменен, а иначе — логическое значение false
<code>void notifyObservers()</code>	Извещает наблюдателей, что вызывающий объект был изменен методом <code>update()</code> . В качестве второго параметра методу <code>update()</code> передается пустое значение null
<code>void notifyObservers(Object объект)</code>	Извещает наблюдателей, что вызывающий объект был изменен методом <code>update()</code> . В качестве второго параметра методу <code>update()</code> передается заданный объект
<code>protected void setChanged()</code>	Вызывается при изменении вызывающего объекта

Следует заметить, что у метода `notifyObservers()` имеются две формы: с аргументом и без такового. Если вызвать метод `notifyObservers()` с объектом в качестве аргумента, этот объект передается методу `update()` наблюдателя в качестве второго параметра. В противном случае методу `update()` передается пустое значение **null**. В качестве второго параметра можно передать объект любого типа, подходящий для прикладной программы.

Интерфейс `Observer`

Чтобы организовать наблюдение за объектом, следует реализовать интерфейс `Observer`. В этом интерфейсе определяется следующий единственный метод:

`void update(Observable наблюдаемый_объект, Object аргумент)`

где параметр *наблюдаемый_объект* обозначает объект для наблюдения, тогда как *аргумент* — значение, передаваемое методом `notifyObservers()`. Метод `update()` вызывается при изменении наблюдаемого объекта.

Пример наблюдения за объектами

Ниже приведен пример программы, демонстрирующий обращение с наблюдаемым объектом. В этой программе создается класс наблюдателя `Watcher`, реализующий интерфейс `Observer`, а также наблюдаемый класс `BeingWatched`, расширяющий класс `Observable` и содержащий метод `counter()`, который просто выполняет обратный отсчет от заданного значения, вызывая метод `sleep()` для ожидания в течение 10 секунд в промежутках между последовательными отсчетами. Каждый раз, когда счетчик изменяется, вызывается метод `notifyObservers()`, которому в качестве аргумента передается текущее значение счетчика. Это вынуждает вызывать метод `update()` из класса `Watcher`, который выводит текущее значение счетчика. В теле метода `main()` сначала создаются наблюдающий и наблюдаемый объекты классов `Watcher` и `BeingWatched`, называемые `observing` и `observed` соответственно. Затем объект `observing` вводится в список наблюдателей за объектами `observed`. Это означает, что метод `observing.update()` будет вызываться всякий раз, когда метод `counter()` вызывает метод `notifyObservers()`.

```
/* Продемонстрировать применение класса Observable
   и интерфейса Observer
*/
import java.util.*;

// Это наблюдающий класс
class Watcher implements Observer {
    public void update(Observable obj, Object arg) {
        System.out.println(
            "Метод update() вызван, отсчет count равен " +
            ((Integer)arg).intValue());
    }
}

// А это наблюдаемый класс
class BeingWatched extends Observable {
    void counter(int period) {
        for( ; period >=0; period--) {
            setChanged();
            notifyObservers(new Integer(period));
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                System.out.println("Ожидание прервано");
            }
        }
    }
}

class ObserverDemo {
    public static void main(String args[]) {
        BeingWatched observed = new BeingWatched();
        Watcher observing = new Watcher();
        /* Ввести наблюдающий объект в список наблюдателей
           за наблюдаемым объектом */
        observed.addObserver(observing);
        observed.counter(10);
    }
}
```

Ниже приведен результат, выводимый данной программой.

```
Метод update() вызван, отсчет count равен 10
Метод update() вызван, отсчет count равен 9
Метод update() вызван, отсчет count равен 8
Метод update() вызван, отсчет count равен 7
Метод update() вызван, отсчет count равен 6
Метод update() вызван, отсчет count равен 5
Метод update() вызван, отсчет count равен 4
Метод update() вызван, отсчет count равен 3
Метод update() вызван, отсчет count равен 2
Метод update() вызван, отсчет count равен 1
Метод update() вызван, отсчет count равен 0
```

Наблюдателями могут быть несколько объектов. Так, в следующем примере программы реализуются два класса наблюдателей, и объекты каждого из них вводятся в список наблюдателей за объектом класса BeingWatched. Второй наблюдатель ожидает до тех пор, пока счетчик достигнет нулевого значения, после чего подается звуковой сигнал.

```
/* За одним объектом могут наблюдать
   несколько наблюдателей.
*/
import java.util.*;

// Класс первого наблюдателя
class Watcher1 implements Observer {
    public void update(Observable obj, Object arg) {
        System.out.println(
            "Метод update() вызван, отсчет count равен " +
            ((Integer)arg).intValue());
    }
}

// Класс второго наблюдателя
class Watcher2 implements Observer {
    public void update(Observable obj, Object arg) {
        // по окончании выдать звуковой сигнал
        if(((Integer)arg).intValue() == 0)
            System.out.println("Готово" + '\n');
    }
}

// Наблюдаемый класс
class BeingWatched extends Observable {
    void counter(int period) {
        for( ; period >=0; period--) {
            setChanged();
            notifyObservers(new Integer(period));

            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                System.out.println("Ожидание прервано");
            }
        }
    }
}

class TwoObservers {
    public static void main(String args[]) {
        BeingWatched observed = new BeingWatched();
```

```

    Watcher1 observing1 = new Watcher1();
    Watcher2 observing2 = new Watcher2();

    // ввести в список оба наблюдателя
    observed.addObserver(observing1);
    observed.addObserver(observing2);
    observed.counter(10);
}
}

```

Класс `Observable` и интерфейс `Observer` позволяют реализовывать изощренные программные архитектуры, основанные на методике “документ–представление”.

Классы `Timer` и `TimerTask`

В пакете `java.util` предоставляется интересная и удобная возможность планировать запуск задания в определенный момент времени в будущем. Такую возможность предоставляют классы `Timer` и `TimerTask`. Используя эти классы, можно создать поток, исполняющийся в фоновом режиме и ожидающий в течение заданного времени. По истечении заданного времени запускается задание, связанное с этим потоком исполнения. Различные параметры позволяют запланировать запуск задания на повторное исполнение или на определенную дату. И хотя с помощью класса `Thread` можно всегда запланировать задание вручную на запуск в определенный момент времени, тем не менее классы `Timer` и `TimerTask` значительно упрощают этот процесс.

Классы `Timer` и `TimerTask` действуют совместно. В частности, класс `Timer` служит для планирования выполняемого задания. Планируемое задание должно быть экземпляром класса `TimerTask`. Следовательно, чтобы запланировать задание, следует создать сначала объект класса `TimerTask`, а затем запланировать его запуск с помощью экземпляра класса `Timer`.

Класс `TimerTask` реализует интерфейс `Runnable`. Это означает, что он может быть использован для создания потока исполнения. Ниже приведен его конструктор.

`TimerTask()`

В классе `TimerTask` определены методы, перечисленные в табл. 19.9. Обратите внимание на то, что метод `run()` является абстрактным, а это означает, что он должен быть переопределен. Метод `run()`, определенный в интерфейсе `Runnable`, содержит исполняемый код. Следовательно, простейший способ запланировать запуск задания по таймеру — расширить класс `TimerTask` и переопределить метод `run()`.

Как только задание будет сформировано, его выполнение планируется с помощью объекта класса `Timer`. Ниже приведены конструкторы класса `Timer`.

```

Timer()
Timer(boolean потоковый_демон)
Timer(String имя_потока)
Timer(String имя_потока, boolean потоковый_демон)

```

В первой форме конструктора сначала создается объект типа `Timer`, а затем он запускается как обычный поток исполнения. Во второй форме используется потоковый демон, если параметр `потоковый_демон` принимает логическое значение

true. Поточковый демон будет исполняться только до тех пор, пока выполняется остальная часть программы. Третья и четвертая формы конструкторов позволяют указывать имя объекта типа `Timer`. Методы, определенные в классе `Timer`, перечислены в табл. 19.10.

Таблица 19.9. Методы из класса `TimerTask`

Метод	Описание
<code>boolean cancel()</code>	Прерывает задание. Возвращает логическое значение true , если выполнение задания прервано, а иначе — логическое значение false
<code>abstract void run()</code>	Содержит код задания, запускаемого таймером
<code>long scheduledExecutionTime()</code>	Возвращает момент времени, на который запуск задания планировался в последний раз

Таблица 19.10. Методы из класса `Timer`

Метод	Описание
<code>void cancel()</code>	Прерывает поток исполнения таймера
<code>int purge()</code>	Удаляет прерванные задания из очереди таймера
<code>void schedule(TimerTask задание_no_таймеру, long ожидание)</code>	Планирует указанное <i>задание_no_таймеру</i> для выполнения через промежуток времени в миллисекундах, определяемый параметром <i>ожидание</i>
<code>void schedule(TimerTask задание_no_таймеру, long ожидание, long повтор)</code>	Планирует указанное <i>задание_no_таймеру</i> для выполнения через промежуток времени в миллисекундах, определяемый параметром <i>ожидание</i> . Выполнение задания затем повторяется через промежуток времени в миллисекундах, определяемый параметром <i>повтор</i>
<code>void schedule(TimerTask задание_no_таймеру, Date заданное_время)</code>	Планирует указанное <i>задание_no_таймеру</i> для выполнения на <i>заданное_время</i>
<code>void schedule(TimerTask задание_no_таймеру, Date заданное_время, long повтор)</code>	Планирует указанное <i>задание_no_таймеру</i> для выполнения на <i>заданное_время</i> . Выполнение задания затем повторяется через промежуток времени в миллисекундах, определяемый параметром <i>повтор</i>
<code>void scheduleAtFixedRate(TimerTask задание_no_таймеру, long ожидание, long повтор)</code>	Планирует указанное <i>задание_no_таймеру</i> для выполнения через промежуток времени в миллисекундах, определяемый параметром <i>ожидание</i> . Выполнение задания затем повторяется через промежуток времени в миллисекундах, определяемый параметром <i>повтор</i> . Время каждого повтора задается относительно первого запуска, а не предыдущего. Следовательно, общая частота повторов остается фиксированной

Метод	Описание
void scheduleAtFixedRate(TimerTask задание_по_таймеру, Date заданное_время, long повтор)	Планирует указанное <i>задание_по_таймеру</i> для выполнения на <i>заданное_время</i> . Выполнение задания затем повторяется через промежуток времени в миллисекундах, определяемый параметром <i>повтор</i> . Время каждого повтора задается относительно первого запуска, а не предыдущего. Следовательно, общая частота повторов остается фиксированной

Как только объект класса `Timer` будет создан, запуск задания можно запланировать, вызвав метод `schedule()` из этого класса. Как следует из табл. 19.10, имеется несколько форм метода `schedule()`, позволяющих запланировать задание разными способами.

Если задание не формируется как демон, то по завершении программы придется вызвать метод `cancel()`, чтобы прервать это задание. И если не сделать этого, то программа может на некоторое время “зависнуть”.

В приведенном ниже примере программы демонстрируется применение классов `Timer` и `TimerTask`. В этой программе определяется задание, запускаемое по таймеру, а при его выполнении метод `run()` выводит сообщение “Задание по таймеру выполняется”. Это задание планируется для запуска каждые полсекунды после первоначальной паузы в течение одной секунды.

```
// Продемонстрировать применение классов Timer и TimerTask
import java.util.*;

class MyTimerTask extends TimerTask {
    public void run() {
        System.out.println("Задание по таймеру выполняется.");
    }
}

class TTest {
    public static void main(String args[]) {
        MyTimerTask myTask = new MyTimerTask();
        Timer myTimer = new Timer();
        /* Установить первоначальную паузу в течение одной
           секунды, а затем повторять задание каждые полсекунды
        */
        myTimer.schedule(myTask, 1000, 500);
        try {
            Thread.sleep(5000);
        } catch (InterruptedException exc) {}
        myTimer.cancel();
    }
}
```

Класс Currency

Класс `Currency` инкапсулирует сведения о денежной единице. В этом классе конструкторы не определяются. Методы из класса `Currency` перечислены в табл. 19.11. В следующем примере программы демонстрируется применение класса `Currency`:

```
// Продемонстрировать применение класса Currency
import java.util.*;

class CurDemo {
    public static void main(String args[]) {
        Currency c;

        c = Currency.getInstance(Locale.US);

        System.out.println("Символ: " + c.getSymbol());
        System.out.println(
            "Количество цифр в дробной части числа по умолчанию: "+
            c.getDefaultFractionDigits());
    }
}
```

Ниже приведен результат, выводимый данной программой.

```
Символ: $
Количество цифр в дробной части числа по умолчанию: 2
```

Таблица 19.11. Методы из класса Currency

Метод	Описание
static Set<Currency> getAvailableCurrencies()	Возвращает ряд поддерживаемых денежных единиц
String getCurrencyCode()	Возвращает код денежной единицы по стандарту ISO 4217 для вызывающего объекта
int getDefaultFractionDigits()	Возвращает количество цифр после десятичной точки, которые обычно указываются в денежной единице для вызывающего объекта. Например, для сумм в долларах после десятичной точки указываются две цифры
String getDisplayName()	Возвращает название денежной единицы из региональных настроек по умолчанию для вызывающего объекта
String getDisplayName(Locale <i>региональные_настройки</i>)	Возвращает название денежной единицы из указанных <i>региональных_настроек</i> для вызывающего объекта
static Currency getInstance(Locale <i>объект_региональных_настроек</i>)	Возвращает объект типа Currency для региональных настроек, обозначаемых параметром <i>объект_региональных_настроек</i>
static Currency getInstance(String <i>код</i>)	Возвращает объект типа Currency , связанный с указанным <i>кодом</i> денежной единицы
int getNumericCode()	Возвращает числовой код денежной единицы по стандарту ISO 4217 для вызывающего объекта
String getSymbol()	Возвращает знак денежной единицы (например, \$) для вызывающего объекта
String getSymbol(Locale <i>объект_региональных_настроек</i>)	Возвращает знак денежной единицы (например, \$) для региональных настроек, обозначаемых параметром <i>объект_региональных_настроек</i>
String toString()	Возвращает код денежной единицы для вызывающего объекта

Класс `Formatter`

В основу системы поддержки форматированного вывода положен класс `Formatter`. Он выполняет *преобразование формата*, позволяющее выводить числа, строки, время и даты практически в любом виде. Этот класс действует подобно функции `printf()` в C/C++, и если у вас имеется некоторый опыт программирования на C/C++, то освоить класс `Formatter` вам не составит большого труда, а также упростит перенос кода из C/C++ на Java. Но даже если у вас нет опыта программирования на C/C++, вам все равно будет нетрудно научиться форматировать данные средствами класса `Formatter`.

На заметку! Несмотря на то что класс `Formatter` очень похож на функцию `printf()` в C/C++, он все же имеет некоторые отличия и новые средства форматирования данных. Поэтому, если у вас имеется некоторый опыт программирования на C/C++, рекомендуется внимательно прочитать этот раздел.

Конструкторы класса `Formatter`

Прежде чем воспользоваться классом `Formatter` для форматирования выводимых данных, следует создать его объект. В общем, объект класса `Formatter` преобразует в форматированный текст двоичную форму данных, используемых в прикладной программе. Он явно сохраняет форматированный текст в буфере, содержимое которого может быть доступно из прикладной программы в любой удобный момент. При этом имеются следующие возможности: разрешить объекту типа `Formatter` предоставить такой буфер автоматически, указать его явно при создании объекта типа `Formatter` или позволить объекту класса `Formatter` вывести содержимое его буфера в файл.

В классе `Formatter` определяется немало конструкторов, позволяющих создавать его объекты самыми разными способами. Ниже перечислены лишь некоторые конструкторы этого класса.

```

Formatter()
Formatter(Appendable буфер)
Formatter(Appendable буфер, Locale региональные_настройки)
Formatter(String имя_файла) throws FileNotFoundException
Formatter(String имя_файла, String набор_символов)
throws FileNotFoundException, UnsupportedEncodingException
Formatter(File файл_вывода) throws FileNotFoundException
Formatter(OutputStream поток_вывода)

```

Здесь параметр *буфер* обозначает конкретный буфер для хранения отформатированных выводимых данных. Если указанный *буфер* пуст, то объект типа `Formatter` автоматически выделяет объект типа `StringBuffer` для хранения отформатированных выводимых данных. Параметр *региональные_настройки* обозначает используемые региональные настройки. Если региональные настройки не заданы, используются региональные настройки по умолчанию. Параметр *имя_файла* обозначает имя того файла, который будет принимать отформатированные выводимые данные. Параметр *набор_символов* обозначает используемый набор символов. Если

конкретный набор символов не указан, используется набор символов по умолчанию. Параметр *файл_вывода* обозначает ссылку на открытый файл, который должен принимать выводимые данные. И наконец, параметр *поток_вывода* обозначает ссылку на тот поток вывода, куда будут направлены выводимые данные. Если для вывода используется файл, то выводимые данные также записываются в этот файл.

Чаще всего применяется первый из перечисленных выше конструкторов рассматриваемого здесь класса, поскольку у него отсутствуют параметры. Он автоматически использует региональные настройки по умолчанию и выделяет объект типа `StringBuffer` для хранения отформатированных выводимых данных.

Методы из класса `Formatter`

В классе `Formatter` определяются методы, перечисленные в табл. 19.12.

Таблица 19.12. Методы из класса `Formatter`

Метод	Описание
<code>void close()</code>	Закрывает вызываемый объект типа Formatter . В итоге освобождаются все ресурсы, используемые этим объектом. После закрытия объекта типа Formatter его больше нельзя использовать. Любая попытка использовать закрытый объект типа Formatter приведет к исключению типа FormatterClosedException
<code>void flush()</code>	Очищает буфер отформатированных данных. В итоге все данные, находящиеся в буфере, выводятся по месту назначения. Этот метод вызывается в основном для объекта типа Formatter , связанного с файлом
<code>Formatter format(String <i>форматирующая_строка</i>, Object ... <i>аргументы</i>)</code>	Форматирует указанные <i>аргументы</i> в соответствии со спецификаторами формата, которые содержит заданная <i>форматирующая_строка</i> . Возвращает вызываемый объект
<code>Formatter format(Locale <i>региональные_настройки</i>, String <i>форматирующая_строка</i>, Object ... <i>аргументы</i>)</code>	Форматирует указанные <i>аргументы</i> в соответствии со спецификаторами формата, которые содержит заданная <i>форматирующая_строка</i> . При форматировании используются заданные <i>региональные_настройки</i> . Возвращает вызываемый объект
<code>IOException ioException()</code>	Если базовый объект, который служит в качестве адресата, генерирует исключение типа IOException , то возвращает именно оно, а иначе — пустое значение null
<code>Locale locale()</code>	Возвращает региональные настройки для вызывающего объекта
<code>Appendable out()</code>	Возвращает ссылку на базовый объект, который служит в качестве адресата для выводимых данных
<code>String toString()</code>	Возвращает объект типа String , содержащий отформатированные выводимые данные

Основы форматирования

Как только объект класса `Formatter` будет создан, его можно применять для создания форматированных строк. Для этой цели служит метод `format()`. Ниже приведена его наиболее часто используемая форма.

`Formatter format(String форматирующая_строка, Object ... аргументы)`

Здесь параметр *форматирующая_строка* состоит из элементов двух типов. К первому типу относятся символы, которые просто копируются в буфер вывода, а ко второму типу — *спецификаторы формата*, определяющие способ, которым должны выводиться указываемые далее *аргументы*.

В простейшей форме спецификатор формата начинается со знака процента с последующим *спецификатором преобразования*. Все спецификаторы преобразования формата состоят из единственного знака. Например, спецификатор формата чисел с плавающей точкой обозначается следующим образом: `%f`. В общем, должно быть указано столько аргументов, сколько задано спецификаторов формата, причем соответствие аргументов устанавливается слева направо. Рассмотрим в качестве примера следующий фрагмент кода:

```
Formatter fmt = new Formatter();
fmt.format("Форматировать %s очень просто: %d %f",
    "средствами Java", 10, 98.6);
```

В этом фрагменте кода создается объект типа `Formatter`, содержащий следующую отформатированную строку:

Форматировать средствами Java очень просто: 10 98.600000

В данном примере спецификаторы формата `%s`, `%d` и `%f` замещаются аргументами, следующими за строкой формата. В частности, спецификатор формата `%s` заменяется символьной строкой "средствами Java", спецификатор формата `%d` — числовым значением 10, а спецификатор формата `%f` — числовым значением 98.6. Все остальные символы в формирующей строке используются без изменения. Как и следовало ожидать, спецификатор формата `%s` обозначает символьную строку, а спецификатор формата `%d` — целочисленное значение. Как упоминалось выше, спецификатор формата `%f` означает числовое значение с плавающей точкой.

Метод `format()` принимает широкое разнообразие спецификаторов формата, перечисленных в табл. 19.13. Обратите внимание на то, что многие спецификаторы формата имеют как прописную, так и строчную форму. Когда используется прописная форма, буквы отображаются в верхнем регистре, а в остальном обе формы равнозначны. Следует иметь в виду, что каждый спецификатор формата проверяется в Java на соответствие типу аргумента. Если такое соответствие отсутствует, генерируется исключение типа `IllegalFormatException`.

Как только символьная строка будет отформатирована, ее можно получить, вызвав метод `toString()`. Например, в следующей строке кода получается отформатированная символьная строка, содержащаяся в объекте `fmt`:

```
String str = fmt.toString();
```

Таблица 19.13. Спецификаторы формата

Спецификатор формата	Применяемое преобразование
<code>%a</code>	Шестнадцатеричное значение с плавающей точкой
<code>%A</code>	
<code>%b</code>	Логическое значение
<code>%B</code>	
<code>%c</code>	Символ
<code>%d</code>	Десятичное целое значение
<code>%h</code>	Хеш-код аргумента
<code>%H</code>	
<code>%e</code>	Экспоненциальное представление числа
<code>%E</code>	
<code>%f</code>	Десятичное число с плавающей точкой
<code>%g</code>	Использует спецификатор формата <code>%e</code> или <code>%f</code> в зависимости от формируемого значения и заданной точности
<code>%G</code>	
<code>%o</code>	Восьмеричное целое число
<code>%n</code>	Вставляет знак перевода строки
<code>%s</code>	Символьная строка
<code>%S</code>	
<code>%t</code>	Время и дата
<code>%T</code>	
<code>%x</code>	Шестнадцатеричное целое число
<code>%X</code>	
<code>%%</code>	Вставляет знак %

Безусловно, если требуется лишь вывести отформатированную строку, то для этого ее совсем не обязательно присваивать ее сначала объекту типа `String`. Например, когда объект типа `Formatter` передается методу `println()`, его метод `toString()` вызывается автоматически.

Ниже приведен краткий пример программы, где демонстрируется порядок составления и вывода отформатированной строки.

```
// Очень простой пример применения класса Formatter
import java.util.*;
```

```
class FormatDemo {
    public static void main(String args[]) {
        Formatter fmt = new Formatter();

        fmt.format(
            "Форматировать %s просто %d %f", "средствами Java", 10, 98.6);

        System.out.println(fmt);
        fmt.close();
    }
}
```

Следует также иметь в виду, что ссылку на буфер вывода можно получить, вызвав метод `out()`. Этот метод возвращает ссылку на объект типа `Appendable`.

Теперь, когда разъяснен общий механизм создания форматированных строк, рассмотрим подробнее каждый из перечисленных выше видов преобразования форматлируемых данных в отдельности. Попутно поясним такие параметры, как выравнивание, минимальная ширина поля и точность.

Форматирование строк и символов

Для форматирования отдельных символов служит спецификатор формата `%c`. В итоге соответствующий символьный аргумент будет выводиться без всяких изменений. А для форматирования символьных строк служит спецификатор формата `%s`.

Форматирование чисел

Для форматирования целых чисел в десятичном формате служит спецификатор `%d`, для форматирования чисел с плавающей точкой в десятичном формате – спецификатор `%f`, а для форматирования чисел с плавающей точкой в экспоненциальном представлении – спецификатор `%e`. Числа в экспоненциальном представлении указываются в следующей общей форме:

`x.dddddde+/-yy`

Спецификатор формата `%g` предписывает классу `Formatter` использовать спецификатор `%f` или `%e` в зависимости от форматлируемого значения и заданной точности, которая по умолчанию составляет 6 цифр. В следующем примере программы демонстрируется применение спецификаторов формата `%f` или `%e`:

```
// Продемонстрировать применение спецификаторов %f и %e
import java.util.*;

class FormatDemo2 {
    public static void main(String args[]) {
        Formatter fmt = new Formatter();

        for(double i=1.23; i < 1.0e+6; i    100) {
            fmt.format("%f %e", i, i);
            System.out.println(fmt);
        }
        fmt.close();
    }
}
```

Эта программа выводит следующий результат:

```
1.230000 1.230000e+00
1.230000 1.230000e+00 123.000000 1.230000e+02
1.230000 1.230000e+00 123.000000 1.230000e+02 12300.000000 1.230000e+04
```

Для вывода целых чисел в восьмеричном или шестнадцатеричном формате служит спецификатор `%o` или `%x` соответственно. Например, выполнение следующего фрагмента кода:

```
fmt.format(
    "Шестнадцатеричное число: %x, восьмеричное число: %o", 196, 196);
```

приводит к выводу такого результата:

Шестнадцатеричное число: c4, восьмеричное число: 304

Для вывода чисел с плавающей точкой в шестнадцатеричном формате служит спецификатор **%a**. На первый взгляд, форматирование с помощью спецификатора **%a** может показаться не совсем обычным. Дело в том, что для такого представления чисел используется форма, подобная экспоненциальному представлению и состоящая из шестнадцатеричной мантиссы и десятичного показателя в степени 2. Ниже приведен общий формат представления чисел с плавающей точкой в шестнадцатеричном виде.

0x1.*sig**exp*

Здесь *sig* содержит дробную часть мантиссы, *exp* — показатель степени, а символ *p* обозначает начало показателя степени. Например, следующий вызов метода:

```
fmt.format("%a", 512.0);
```

приводит к выводу такого результата:

0x1.0p9

Форматирование времени и даты

Одно из наиболее эффективных преобразований формируемых данных задается с помощью спецификатора формата **%t**, который позволяет форматировать сведения о дате и времени. Спецификатор формата **%t** действует несколько иначе, чем другие спецификаторы, поскольку он требует указывать суффиксы для описания части или точности формата времени и даты. Суффиксы перечислены в табл. 19.14. Например, чтобы вывести время в минутах, следует использовать спецификатор **%tm**, где **M** обозначает минуты в поле из двух символов. Аргументы, соответствующие спецификатору **%t**, должны иметь тип `Calendar`, `Date`, `Long` или `long`.

Таблица 19.14. Суффиксы формата времени и даты

Суффикс	Заменяется на
a	Сокращенное название дня недели
A	Полное название дня недели
b	Сокращенное название месяца
B	Полное название месяца
c	Стандартная строка даты и времени в таком формате: <i>день месяц дата чч:мм:сс часовой пояс год</i>
C	Первые две цифры года
d	День месяца в десятичном представлении (01–31)
D	месяц/день/год
e	День месяца в десятичном представлении (1–31)
F	Формат «год-месяц-день»
h	Сокращенное название месяца
H	Часы (от 00 до 23)
I	Часы (от 01 до 12)

Суффикс	Заменяется на
j	День года в десятичном представлении (от 001 до 366)
k	Часы (от 0 до 23)
l	Часы (от 1 до 12)
L	Миллисекунды (от 000 до 999)
m	Месяц в десятичном представлении (от 01 до 13)
M	Минуты в десятичном представлении (от 00 до 59)
N	Наносекунды (от 000000000 до 999999999)
P	Региональный эквивалент времени до полудня (AM) или после полудня (PM) в нижнем регистре
Q	Количество миллисекунд, прошедших с даты 01/01/1970
r	чч:мм:сс (12-часовой формат)
R	чч:мм (24-часовой формат)
S	Секунды (от 00 до 60)
s	Количество миллисекунд, прошедших с даты 01/01/1970 в формате времени UTC
T	чч:мм:сс (24-часовой формат)
Y	Год в десятичном представлении без указания столетия (от 00 до 99)
y	Год в десятичном представлении, включая столетие (от 0001 до 9999)
z	Смещение относительно времени UTC
Z	Наименование часового пояса

Ниже приведена программа, демонстрирующая применение некоторых форматов даты и времени.

```
// Форматирование времени и даты
import java.util.*;

class TimeDateFormat {
    public static void main(String args[]) {
        Formatter fmt = new Formatter();
        Calendar cal = Calendar.getInstance();

        // вывести время в стандартном 12-часовом формате
        fmt.format("%tr", cal);
        System.out.println(fmt);

        // вывести все сведения о дате и времени
        fmt = new Formatter();
        fmt.format("%tc", cal);
        System.out.println(fmt);

        // вывести только часы и минуты
        fmt = new Formatter();
        fmt.format("%tl:%tM", cal, cal);
        System.out.println(fmt);

        // вывести название и номер месяца
        fmt = new Formatter();
        fmt.format("%tB %tb %tm", cal, cal, cal);
        System.out.println(fmt);
    }
}
```

```
        fmt.close();
    }
}
```

Ниже приведен пример выполнения данной программы.

```
03:15:34 PM
Wed Jan 01 15:15:34 CST 2014
3:15
January Jan 01
```

Спецификаторы формата %n и %%

Спецификаторы формата %n и %% отличаются от других тем, что они не сопоставляются с аргументом, а представляют управляющие последовательности, вставляющие символ в выводимую последовательность. В частности, спецификатор %n вставляет знак перевода строки, а спецификатор %% — знак процента. Ни один из этих символов не может быть введен непосредственно в форматирующую строку. Безусловно, чтобы вставить, например, знак перевода строки, можно также воспользоваться стандартной управляющей последовательностью \n.

В следующем примере программы демонстрируется применение спецификаторов формата %n и %%:

```
// Продемонстрировать применение спецификаторов формата %n и %%
import java.util.*;
class FormatDemo3 {
    public static void main(String args[]) {
        Formatter fmt = new Formatter();

        fmt.format(
            "Копирование файла%nПередача завершена на %d%%", 88);
        System.out.println(fmt);
        fmt.close();
    }
}
```

Эта программа выводит следующий результат:

```
Копирование файла
Передача завершена на 88%
```

Указание минимальной ширины поля

Целое число, указанное между знаком % и кодом преобразования формата, действует в качестве *спецификатора минимальной ширины поля*, дополняющего выводимые данные пробелами, чтобы обеспечить определенную минимальную длину. Если символьная строка или число получается длиннее этого указанного минимума, они будут выведены полностью. По умолчанию дополнение осуществляется пробелами. Если же выводимые данные требуются дополнить нулями, перед спецификатором ширины поля следует указать 0. Например, формат %05d означает дополнение нулями числа, состоящее менее чем из 5 цифр, чтобы его общая ширина была равна пяти знакам. Спецификатор ширины поля можно применять вместе с остальными спецификаторами формата, кроме %n.

В следующем примере программы демонстрируется применение спецификатора минимальной ширины поля вместе со спецификатором преобразования `%f`:

```
// Продемонстрировать применение спецификатора ширины поля
import java.util.*;

class FormatDemo4 {
    public static void main(String args[]) {
        Formatter fmt = new Formatter();

        fmt.format("|%f|\n|%12f|\n|%012f|",
                    10.12345, 10.12345, 10.12345);

        System.out.println(fmt);
        fmt.close();
    }
}
```

Эта программа выводит следующий результат:

```
|10.123450|
| 10.123450|
|00010.123450|
```

В первой строке приведенного выше результата выводится число `10,12345` с шириной по умолчанию. Во второй строке выводится это же число в поле из 12 символов. А в третьей строке оно выводится в поле из 12 символов, дополняемом начальными нулями.

Минимальный модификатор ширины поля часто используется для составления таблиц, состоящих из строк и столбцов. В следующем примере программы выводится таблица квадратов и кубов чисел от 1 до 10.

```
// Составить таблицу квадратов и кубов заданных чисел
import java.util.*;

class FieldWidthDemo {
    public static void main(String args[]) {
        Formatter fmt;

        for(int i=1; i <= 10; i++) {
            fmt = new Formatter();
            fmt.format("%4d %4d %4d", i, i*i, i*i*i);
            System.out.println(fmt);
            fmt.close();
        }
    }
}
```

Ниже приведен результат, выводимый данной программой.

```
1      1      1
2      4      8
3      9     27
4     16     64
5     25    125
6     36    216
7     49    343
8     64    512
9     81    729
10    100   1000
```

Указание точности

Спецификатор точности можно применять вместе со спецификаторами формата `%f`, `%e`, `%g` и `%s`. Он указывается после спецификатора минимальной ширины поля (если таковой имеется) и состоит из точки и целого числа. Его конкретное значение зависит от типа данных, к которому он применяется.

Если спецификатор точности применяется к данным с плавающей точкой вместе со спецификатором формата `%f` или `%e`, он определяет количество отображаемых десятичных цифр. Например, формат `%10.4f` означает вывод числа шириной не меньше 10 знаков с четырьмя цифрами после запятой. Если же применяется спецификатор `%g`, точность определяет количество значащих десятичных цифр. Точность по умолчанию составляет 6 цифр после запятой.

Применительно к символьным строкам спецификатор точности задает максимальную ширину поля. Например, формат `%5.7s` означает вывод строки длиной минимум пять символов, но не больше семи символов. Если же строка оказывается длиннее максимальной ширины, конечные символы отбрасываются.

В следующем примере программы демонстрируется применение спецификатора точности:

```
// Продемонстрировать применение спецификатора точности
import java.util.*;
class PrecisionDemo {
    public static void main(String args[]) {
        Formatter fmt = new Formatter();

        // Формат с четырьмя цифрами после десятичной точки
        fmt.format("%.4f", 123.1234567);
        System.out.println(fmt);
        fmt.close();

        // Формат с двумя цифрами после десятичной точки
        // в поле из 16 символов
        fmt = new Formatter();
        fmt.format("%16.2e", 123.1234567);
        System.out.println(fmt);
        fmt.close();

        // вывести максимум 15 символов из строки
        fmt = new Formatter();
        fmt.format("%.15s",
            "Форматировать в Java теперь очень просто.");
        System.out.println(fmt);
        fmt.close();
    }
}
```

Ниже приведен результат, выводимый данной программой.

```
123.1235
1.23e+02
форматировать в
```

Применение признаков формата

В классе `Formatter` распознается ряд *признаков* формата, с помощью которых можно управлять различными аспектами преобразования форматируемых данных.

Все признаки формата обозначаются одним символом, который указывается после знака % в спецификаторе формата. Признаки формата перечислены в табл. 19.15.

Таблица 19.15. Признаки формата

Признак	Действие
-	Выравнивание по левому краю
#	Альтернативный формат преобразования
0	Выводимые данные дополняются нулями вместо пробелов
пробел	Положительные числа выводятся с предшествующим пробелом
+	Положительные числа выводятся с предшествующим знаком +
,	Числовые значения содержат групповые разделители
(Отрицательные числовые значения заключаются в круглые скобки

Признаки формата применяются не ко всем спецификаторам формата. В следующих разделах они поясняются более подробно.

Выравнивание выводимых данных

По умолчанию все выводимые данные выравниваются по правому краю. Иными словами, если ширина поля больше, чем выводимые данные, то эти данные будут размещены по правому краю поля. Но выводимые данные можно выровнять и по левому краю, указав знак “минус” сразу после знака %. Например, формат %-10.2f обозначает выравнивание по левому краю числа с плавающей точкой и двумя цифрами после десятичной точки в пределах поля из 10 символов. Рассмотрим в качестве примера следующую программу:

```
// Продемонстрировать выравнивание по левому краю
import java.util.*;
class LeftJustify {
    public static void main(String args[]) {
        Formatter fmt = new Formatter();

        // выровнять по правому краю (по умолчанию)
        fmt.format("|%10.2f|", 123.123);
        System.out.println(fmt);
        fmt.close();

        // а теперь выровнять по левому краю
        fmt = new Formatter();
        fmt.format("|%-10.2f|", 123.123);
        System.out.println(fmt);
        fmt.close();
    }
}
```

Ниже приведен результат, выводимый данной программой. Как видите, вторая строка результата выровнена по левому краю в пределах поля из 10 символов.

```
|      123.12|
|123.12      |
```

Признаки пробела, +, 0 и (

Чтобы выводить знак + перед положительными числовыми значениями, достаточно указать признак +. Например, в результате выполнения следующей строки кода:

```
fmt.format("%+d", 100);
```

выводится такая строка:

```
+100
```

При составлении столбцов из чисел иногда удобно выводить пробел перед положительными числами, чтобы положительные и отрицательные числовые значения выводились в одном столбце. Чтобы добиться этого, достаточно указать признак пробела, как показано в приведенном ниже примере программы.

```
// Продемонстрировать применение пробела в качестве
// спецификатора формата
import java.util.*;
```

```
class FormatDemo5 {
    public static void main(String args[]) {
        Formatter fmt = new Formatter();

        fmt.format("% d", -100);
        System.out.println(fmt);
        fmt.close();

        fmt = new Formatter();
        fmt.format("% d", 100);
        System.out.println(fmt);
        fmt.close();

        fmt = new Formatter();
        fmt.format("% d", -200);
        System.out.println(fmt);
        fmt.close();

        fmt = new Formatter();
        fmt.format("% d", 200);
        System.out.println(fmt);
        fmt.close();
    }
}
```

Ниже приведен результат, выводимый данной программой. Обратите внимание на то, что положительные значения в данном результате имеют начальный пробел, что обеспечивает ровное расположение разрядов в столбце.

```
-100
 100
-200
 200
```

Чтобы вывести отрицательные числовые значения в скобках вместо начального минуса, достаточно указать признак (. Например, в результате выполнения следующей строки кода:

```
fmt.format("(%(d", -100);
```

выводится такая строка:

```
(100)
```

А если указать признак 0, то выводимые данные дополняются нулями вместо пробелов.

Признак запятой

При выводе больших чисел зачастую удобно указывать разделители групп, которыми в англоязычной среде являются запятые. Например, числовое значение 1234567 легче читается, когда оно отформатировано в следующем виде: 1,234,567. Для указания спецификаторов группирования служит признак запятой (,). Например, в результате выполнения следующей строки кода:

```
fmt.format("%,.2f", 4356783497.34);
```

выводится такая строка:

```
4,356,783,497.34
```

Признак

Этот признак может применяться вместе со спецификаторами формата %o, %x, %e и %f. Для спецификаторов формата %e и %f признак # обеспечивает наличие десятичной точки даже в том случае, если отсутствует дробная часть числа. Так, если перед спецификатором формата %x указать признак #, то шестнадцатеричное число будет выведено с префиксом 0x. Если же указать признак # перед спецификатором формата %o, восьмеричное число будет выводиться с начальным нулем.

Прописные формы спецификаторов формата

Как упоминалось ранее, некоторые спецификаторы формата имеют прописные формы, которые предписывают употреблять прописные буквы при преобразовании там, где это возможно. В табл. 19.16 описывается действие прописных форм спецификаторов формата.

Таблица 19.16. Прописные формы спецификаторов формата

Спецификатор	Действие
%A	Шестнадцатеричные цифры от a до f выводятся прописными буквами, т.е. от A до F . Кроме того, префикс 0x отображается как 0X , а показатель степени p — как P
%B	Логические значения true и false выводятся прописными буквами
%E	Знак экспоненты e выводится прописной буквой
%G	Знак экспоненты e выводится прописной буквой
%H	Шестнадцатеричные цифры от a до f выводятся прописными буквами, т.е. от A до F
%S	Соответствующая символьная строка выводится прописными буквами
%T	Весь алфавит выводится прописными буквами
%X	Шестнадцатеричные цифры от a до f выводятся прописными буквами, т.е. от A до F . Кроме того, префикс 0x отображается как 0X , если он присутствует

Например, в результате выполнения следующей строки кода:

```
fmt.format("%X", 250);
```

выводится такая строка:

```
PA
```

А выполнение следующей строки кода:

```
fmt.format("%E", 123.1234);
```

приводит к выводу такой строки:

```
1.231234E+02
```

Применение индекса аргумента

В состав класса `Formatter` входит очень удобное средство, позволяющее указать аргумент, к которому должен применяться конкретный спецификатор формата. Обычно порядок следования аргументов и спецификаторов формата сопоставляется слева направо. Это означает, что первый спецификатор формата относится к первому аргументу, второй — ко второму аргументу и т.д. Но, используя *индекс аргумента*, можно явно управлять сопоставлением аргументов со спецификаторами формата.

Индекс аргумента указывается после знака `%` в спецификаторе формата. Он имеет следующий вид:

`n$`

где *n* обозначает индекс нужного аргумента, начиная с 1. Рассмотрим следующий пример кода:

```
fmt.format("%3$d %1$d %2$d", 10, 20, 30);
```

В результате выполнения этого кода выводится такая строка:

```
30 10 20
```

В данном примере первый спецификатор формата соответствует 30, второй — 10, а третий — 20. Следовательно, аргументы используются в порядке, отличающемся от строгого порядка следования слева направо.

Одно из преимуществ индексирования аргументов заключается в том, что оно позволяет повторно использовать аргумент, не указывая его дважды. Например, в результате выполнения следующей строки кода:

```
fmt.format("Число %d в шестнадцатеричном формате равно %1$x", 255);
```

выводится приведенный ниже результат. Как видите, аргумент 255 используется с обоими спецификаторами формата.

Число 255 в шестнадцатеричном формате равно ff

Существует удобное сокращение, которое называется *относительным индексом* и позволяет повторно использовать аргументы, совпадающие с предшествующим спецификатором формата. Для этого достаточно указать знак `<` вместо индекса

аргумента. Например, следующий вызов метода `format()` приводит к такому же результату, что и в предыдущем примере кода:

```
fmt.format("%d в шестнадцатеричном формате равно %x", 255);
```

Относительные индексы особенно удобны при создании пользовательских форматов времени и даты. Рассмотрим следующий пример программы:

```
// Использовать относительные индексы, чтобы упростить
// создание пользовательских форматов даты и времени
import java.util.*;

class FormatDemo6 {
    public static void main(String args[]) {
        Formatter fmt = new Formatter();
        Calendar cal = Calendar.getInstance();

        fmt.format("Today is day %te of %<tB, %<tY", cal);
        System.out.println(fmt);
        fmt.close();
    }
}
```

Ниже приведен пример выполнения данной программы. Благодаря относительной индексации аргумент `cal` может быть передан только один раз вместо трех.

```
Today is day 1 of January, 2011
```

Заккрытие объекта типа `Formatter`

Обычно объект типа `Formatter` следует закрывать, когда завершается его использование. Благодаря этому освобождаются все используемые им ресурсы, что очень важно не только при форматировании данных, выводимых в файл, но и в других случаях. В приведенных выше примерах демонстрировался один из способов закрыть объект типа `Formatter`, который состоит в том, чтобы вызвать явным образом метод `close()`. Но начиная с версии JDK 7 класс `Formatter` реализует интерфейс `AutoCloseable`. Это означает, что он поддерживает оператор `try` с ресурсами. Применяя такой подход, можно автоматически закрывать объект типа `Formatter`, когда он больше не нужен.

Применение оператора `try` с ресурсами для обращения с файлами описывается в главе 13, поскольку файлы относятся к одним из наиболее часто используемых ресурсов, которые следует закрывать. Но те же основные принципы распространяются и на закрытие объекта типа `Formatter`. Ниже приведен первый пример применения объекта типа `Formatter`, переделанный с учетом автоматического управления ресурсами.

```
// Использование автоматического управления ресурсами
// для закрытия объекта типа Formatter
import java.util.*;

class FormatDemo {
    public static void main(String args[]) {

        try (Formatter fmt = new Formatter())
```

```
{
    fmt.format(
        "Форматировать %s просто %d %f", "средствами Java", 10, 98.6);
    System.out.println(fmt);
}
}
```

Эта версия программы выводит такой же результат, как и прежняя ее версия.

Аналог функции printf() в Java

Хотя в непосредственном применении класса `Formatter`, как это делалось в предыдущих примерах, нет ничего формально неверного, для форматирования данных, выводимых на консоль, имеется более удобная альтернатива в виде метода `printf()`. Этот метод автоматически использует класс `Formatter` для создания отформатированной строки. Затем он отправляет эту строку в стандартный поток вывода `System.out`, который по умолчанию представляет консоль. Метод `printf()` определен в обоих классах, `PrintStream` и `PrintWriter`, и подробно рассматривается в главе 20.

Класс Scanner

Класс `Scanner` служит дополнением класса `Formatter`. Он читает отформатированные вводимые данные и преобразует их в двоичную форму. Класс `Scanner` можно применять для чтения данных, вводимых с консоли, из файла, символьной строки или из другого источника, реализующего интерфейс `Readable` или `ReadableByteChannel`. Например, класс `Scanner` можно использовать для чтения числа с клавиатуры и присвоения его значения переменной. Как будет показано далее, класс `Scanner` очень прост в употреблении, несмотря на то, что он обладает развитыми функциональными возможностями.

Конструкторы класса Scanner

В классе `Scanner` определяются конструкторы, перечисленные в табл. 19.17. В общем, объект класса `Scanner` может быть создан для объекта класса `String`, `InputStream`, `File` или объекта любого другого класса, реализующего интерфейс `Readable` либо `ReadableByteChannel`. Ниже приведены некоторые тому примеры.

В следующем фрагменте кода создается объект типа `Scanner` для чтения данных из файла `Test.txt`:

```
FileReader fin = new FileReader("Test.txt");
Scanner src = new Scanner(fin);
```

Этот код оказывается работоспособным потому, что класс `FileReader` реализует интерфейс `Readable`. Следовательно, вызов конструктора интерпретируется как `Scanner(Readable)`.

В следующей строке кода создается объект типа `Scanner` для чтения данных из стандартного потока ввода, которым по умолчанию является клавиатура:

```
Scanner conin = new Scanner(System.in);
```

И этот код оказывается работоспособным потому, что стандартный поток ввода `System.in` относится к типу `InputStream`. Следовательно, вызов конструктора преобразуется в вызов `Scanner(InputStream)`.

В приведенном ниже фрагменте кода создается объект типа `Scanner` для чтения данных из символьной строки.

```
String instr = "10 99.88 сканировать очень просто.";
Scanner conin = new Scanner(instr);
```

Таблица 19.17. Конструкторы класса `Scanner`

Конструктор	Описание
<code>Scanner(File <i>откуда</i>) throws FileNotFoundException</code>	Создает объект типа <code>Scanner</code> , использующий указанный файл <i>откуда</i> в качестве источника для ввода данных
<code>Scanner(File <i>откуда</i>, String <i>набор_символов</i>) throws FileNotFoundException</code>	Создает объект типа <code>Scanner</code> , использующий указанный файл <i>откуда</i> с заданной кодировкой <i>набор_символов</i> в качестве источника для ввода данных
<code>Scanner(InputStream <i>откуда</i>)</code>	Создает объект типа <code>Scanner</code> , использующий указанный поток ввода <i>откуда</i> в качестве источника для ввода данных
<code>Scanner(InputStream <i>откуда</i>, String <i>набор_символов</i>)</code>	Создает объект типа <code>Scanner</code> , использующий указанный поток ввода <i>откуда</i> с заданной кодировкой <i>набор_символов</i> в качестве источника для ввода данных
<code>Scanner(Path <i>откуда</i>) throws IOException</code>	Создает объект типа <code>Scanner</code> , использующий указанный файл <i>откуда</i> в качестве источника для ввода данных
<code>Scanner(Path <i>откуда</i>, String <i>набор_символов</i>) throws IOException</code>	Создает объект типа <code>Scanner</code> , использующий указанный файл <i>откуда</i> с заданной кодировкой <i>набор_символов</i> в качестве источника для ввода данных
<code>Scanner(Readable <i>откуда</i>)</code>	Создает объект типа <code>Scanner</code> , использующий указанный объект <i>откуда</i> типа <code>Readable</code> в качестве источника для ввода данных
<code>Scanner(ReadableByteChannel <i>откуда</i>)</code>	Создает объект типа <code>Scanner</code> , использующий указанный объект <i>откуда</i> типа <code>ReadableByteChannel</code> в качестве источника для ввода данных
<code>Scanner(ReadableByteChannel <i>откуда</i>, String <i>набор_символов</i>)</code>	Создает объект типа <code>Scanner</code> , использующий указанный объект <i>откуда</i> типа <code>ReadableByteChannel</code> с заданной кодировкой <i>набор_символов</i> в качестве источника для ввода данных
<code>Scanner(String <i>откуда</i>)</code>	Создает объект типа <code>Scanner</code> , использующий указанную символьную строку <i>откуда</i> в качестве источника для ввода данных

Основы сканирования

Как только объект типа `Scanner` будет создан, его очень просто использовать для чтения отформатированных вводимых данных. В общем, объект типа `Scanner`

читает лексемы из некоторого базового источника, указываемого при создании этого объекта. С точки зрения класса `Scanner` *лексема* — это порция вводимых данных, разграничиваемая рядом разделителей, которыми по умолчанию являются пробелы. Лексема читается по совпадению с конкретным *регулярным выражением*, задающим формат данных. И хотя класс `Scanner` позволяет определить конкретный тип регулярного выражения для совпадения в следующей операции ввода, в состав этого класса входит немало предопределенных шаблонов для сопоставления с элементарными типами вроде `int` и `double`, а также символьными строками. Иными словами, указывать шаблоны для сопоставления, как правило, не требуется.

Таким образом, для применения класса `Scanner` необходимо придерживаться следующей процедуры:

- Определить, доступен ли конкретный тип вводимых данных, вызвав одним из методов типа `Scanner hasNextX`, где *X* — требуемый тип данных.
- Если вводимые данные доступны, прочитать их одним из методов типа `Scanner nextX`.
- Повторить пп. 1 и 2 вплоть до исчерпания вводимых данных.
- Закрыть объект типа `Scanner`, вызвав метод `close()`.

Как упоминалось выше, в классе `Scanner` определяются два ряда методов, которые позволяют читать вводимые данные. К первому ряду относятся методы типа `hasNextX`, перечисленные в табл. 19.18. Эти методы определяют, доступен ли указанный тип ввода. Например, в результате вызова метода `hasNextInt()` возвращается логическое значение `true` только в том случае, если следующая читаемая лексема оказывается целым числом. Если же требуемые данные доступны, они читаются одним из методов типа `Scanner nextX`, перечисленных в табл. 19.19. Например, чтобы прочитать следующее целое число, следует вызвать метод `nextInt()`. В приведенном ниже фрагменте кода показано, каким образом организуется чтение списка целых чисел, вводимых с клавиатуры.

```
Scanner conin = new Scanner(System.in);
int i;
// читать список целых значений
while(conin.hasNextInt()) {
    i = conin.nextInt();
    // ...
}
```

В этом фрагменте кода цикл `while` прерывается, как только следующая лексема не оказывается целым числом. Следовательно, чтение целых чисел в цикле прекращается, как только в потоке ввода обнаруживается значение, отличное от целочисленного типа данных.

Если метод типа `next` не в состоянии найти тип данных, который он ожидает, то генерируется исключение типа `InputMismatchException`. А исключение типа `NoSuchElementException` передается в том случае, если доступные для ввода данные исчерпаны. Поэтому сначала лучше проверить с помощью метода типа `hasNext`, что данные требуемого типа доступны, прежде чем вызывать соответствующий ему метод типа `next`.

Таблица 19.18. Методы типа `hasNext` из класса `Scanner`

Метод	Описание
<code>boolean hasNext()</code>	Возвращает логическое значение true , если для чтения доступна следующая лексема любого типа, а иначе — логическое значение false
<code>boolean hasNext(Pattern шаблон)</code>	Возвращает логическое значение true , если для чтения доступна лексема, совпадающая с заданным <i>шаблоном</i> , а иначе — логическое значение false
<code>boolean hasNext(String шаблон)</code>	Возвращает логическое значение true , если для чтения доступна лексема, совпадающая с заданным <i>шаблоном</i> , а иначе — логическое значение false
<code>boolean hasNextBigDecimal()</code>	Возвращает логическое значение true , если для чтения доступно значение, которое можно разместить в объекте типа BigDecimal , а иначе — логическое значение false
<code>boolean hasNextBigInteger()</code>	Возвращает логическое значение true , если для чтения доступно значение, которое можно разместить в объекте типа BigInteger , а иначе — логическое значение false . (По умолчанию используется основание 10.)
<code>boolean hasNextBigInteger(int основание)</code>	Возвращает логическое значение true , если для чтения доступно значение по указанному <i>основанию</i> , которое можно разместить в объекте типа BigInteger , а иначе — логическое значение false
<code>boolean hasNextBoolean()</code>	Возвращает логическое значение true , если для чтения доступно значение типа boolean , а иначе — логическое значение false
<code>boolean hasNextByte()</code>	Возвращает логическое значение true , если для чтения доступно значение типа byte , а иначе — логическое значение false
<code>boolean hasNextByte(int основание)</code>	Возвращает логическое значение true , если для чтения доступно значение типа byte по указанному <i>основанию</i> , а иначе — логическое значение false
<code>boolean hasNextDouble()</code>	Возвращает логическое значение true , если для чтения доступно значение типа double , а иначе — логическое значение false
<code>boolean hasNextFloat()</code>	Возвращает логическое значение true , если для чтения доступно значение типа float , а иначе — логическое значение false
<code>boolean hasNextInt()</code>	Возвращает логическое значение true , если для чтения доступно значение типа int , а иначе — логическое значение false (По умолчанию используется основание 10.)
<code>boolean hasNextInt(int основание)</code>	Возвращает логическое значение true , если для чтения доступно значение типа int по указанному <i>основанию</i> , а иначе — логическое значение false
<code>boolean hasNextLine()</code>	Возвращает логическое значение true , если для чтения доступна строка вводимых данных

Окончание табл. 19.18

Метод	Описание
<code>boolean hasNextLong()</code>	Возвращает логическое значение true , если для чтения доступно значение типа long , а иначе — логическое значение false (По умолчанию используется основание 10.)
<code>boolean hasNextLong(int long <i>основание</i>)</code>	Возвращает логическое значение true , если для чтения доступно значение типа long по указанному <i>основанию</i> , а иначе — логическое значение false
<code>boolean hasNextShort()</code>	Возвращает логическое значение true , если для чтения доступно значение типа short , а иначе — логическое значение false (По умолчанию используется основание 10.)
<code>boolean hasNextShort(int <i>основание</i>)</code>	Возвращает логическое значение true , если для чтения доступно значение типа short по указанному <i>основанию</i> , а иначе — логическое значение false

Таблица 19.19. Методы типа `next` из класса `Scanner`

Метод	Описание
<code>String next()</code>	Возвращает следующую лексему любого типа из источника ввода данных
<code>String next(Pattern <i>шаблон</i>)</code>	Возвращает следующую лексему, совпадающую с указанным <i>шаблоном</i> , из источника ввода данных
<code>String next(String <i>шаблон</i>)</code>	Возвращает следующую лексему, совпадающую с указанным <i>шаблоном</i> , из источника ввода данных
<code>BigDecimal nextBigDecimal()</code>	Возвращает следующую лексему в виде объекта типа BigDecimal
<code>BigInteger nextBigInteger()</code>	Возвращает следующую лексему в виде объекта типа BigInteger . (По умолчанию используется основание 10.)
<code>BigInteger nextBigInteger(int <i>основание</i>)</code>	Возвращает следующую лексему в виде объекта типа BigInteger по указанному <i>основанию</i>
<code>boolean nextBoolean()</code>	Возвращает следующую лексему в виде значения типа boolean
<code>byte nextByte()</code>	Возвращает следующую лексему в виде значения типа byte . (По умолчанию используется основание 10.)
<code>byte nextByte(int <i>основание</i>)</code>	Возвращает следующую лексему в виде значения типа byte по указанному <i>основанию</i>
<code>double nextDouble()</code>	Возвращает следующую лексему в виде значения типа double
<code>float nextFloat()</code>	Возвращает следующую лексему в виде значения типа float
<code>int nextInt()</code>	Возвращает следующую лексему в виде значения типа int . (По умолчанию используется основание 10.)
<code>int nextInt(int <i>основание</i>)</code>	Возвращает следующую лексему в виде значения типа int по указанному <i>основанию</i>
<code>String nextLine()</code>	Возвращает следующую строку вводимых данных

Метод	Описание
<code>long nextLong()</code>	Возвращает следующую лексему в виде значения типа long . (По умолчанию используется основание 10.)
<code>long nextLong(int основание)</code>	Возвращает следующую лексему в виде значения типа long по указанному <i>основанию</i>
<code>short nextShort()</code>	Возвращает следующую лексему в виде значения типа short . (По умолчанию используется основание 10.)
<code>short nextShort(int основание)</code>	Возвращает следующую лексему в виде значения типа short по указанному <i>основанию</i>

Некоторые примеры применения класса Scanner

Класс Scanner позволяет существенно упростить решение задачи, которая в противном случае оказалась бы трудоемкой. Рассмотрим несколько примеров его применения. В следующей программе подсчитывается среднее из списка чисел, введенных с клавиатуры:

```
// Использовать класс Scanner для вычисления среднего
// из списка введенных числовых значений
import java.util.*;
class AvgNums {
    public static void main(String args[]) {
        Scanner conin = new Scanner(System.in);

        int count = 0;
        double sum = 0.0;

        System.out.println("Введите числа для подсчета среднего.");

        // читать и суммировать числовые значения
        while(conin.hasNext()) {
            if(conin.hasNextDouble()) {
                sum += conin.nextDouble();
                count++;
            }
            else {
                String str = conin.next();
                if(str.equals("готово")) break;
                else {
                    System.out.println("Ошибка формата данных.");
                    return;
                }
            }
        }
        conin.close();
        System.out.println("Среднее равно " + sum / count);
    }
}
```

Эта программа читает числа с клавиатуры и суммирует их до тех пор, пока пользователь не введет символьную строку "готово". В таком случае она прекращает ввод и выводит среднее значение введенных чисел. Ниже приведен пример выполнения данной программы.

Введите числа для подсчета среднего.

1.2

2

3.4

4

готово

Среднее равно 2.65

Рассматриваемая здесь программа читает числа до тех пор, пока не получит лексему, которую нельзя интерпретировать как достоверное значение типа `double`. Когда это происходит, она проверяет, соответствует ли введенная лексема символьной строке "готово". Если она соответствует этой строке, то программа завершается нормально. В противном случае она выводит сообщение об ошибке.

Обратите внимание на то, что числа читаются путем вызова метода `nextDouble()`. Этот метод читает любые числа, которые могут быть приведены к типу `double`, включая целые значения вроде 2 и значения с плавающей точкой, подобные 3.4. Следовательно, число, прочитанное методом `nextDouble()`, не требует наличия десятичной точки. Тот же общий принцип действует во всех остальных методах типа `next`. Они обнаружат совпадение и прочитают данные в любом формате, который может представлять данные запрашиваемого типа.

Еще одна примечательная особенность класса `Scanner` состоит в том, что одну и ту же методику можно применять для чтения данных из разных источников. В качестве примера ниже представлена версия предыдущей программы, переделанная для вычисления среднего из списка чисел, вводимых из файла.

```
// Использовать класс Scanner для вычисления среднего
// из списка чисел, вводимых из файла
import java.util.*;
import java.io.*;
class AvgFile {
    public static void main(String args[]) throws IOException {
        int count = 0;
        double sum = 0.0;

        // вывести данные в файл
        FileWriter fout = new FileWriter("test.txt");
        fout.write("2 3.4 5 6 7.4 9.1 10.5 готово");
        fout.close();
        FileReader fin = new FileReader("Test.txt");
        Scanner src = new Scanner(fin);

        // читать и суммировать числовые значения
        while(src.hasNext()) {
            if(src.hasNextDouble()) {
                sum += src.nextDouble();
                count++;
            }
            else {
                String str = src.next();
                if(str.equals("готово")) break;
                else {
                    System.out.println("Ошибка формата файла.");
                    return;
                }
            }
        }
    }
}
```

```

        src.close();
        System.out.println("Среднее равно " + sum / count);
    }
}

```

Ниже приведен результат, выводимый данной программой.

Среднее равно 6.2

Данный пример программы иллюстрирует другую важную особенность класса `Scanner`. Обратите внимание на то, что поток чтения из файла, доступный по ссылке `fin`, не закрывается непосредственно. Вместо этого он автоматически закрывается, когда объект `src` вызывает метод `close()`. Когда закрывается объект типа `Scanner`, связанный с ним объект типа `Readable` также закрывается (если он реализует интерфейс `Closeable`). Поэтому в данном случае файл, доступный по ссылке `fin`, автоматически закрывается вместе с объектом `src`.

Начиная с версии `JDK 7` класс `Scanner` реализует также интерфейс `AutoCloseable`. Это означает, что объектом этого класса, представляющим сканер, можно управлять в блоке оператора `try` с ресурсами. Как пояснялось в главе 13, когда используется оператор `try` с ресурсами, сканер автоматически закрывается в конце блока этого оператора. Так, в приведенном выше примере программы можно было бы организовать управление объектом `src` следующим образом:

```

try (Scanner src = new Scanner(fin))
{
    // читать и суммировать числа
    while(src.hasNext()) {
        if(src.hasNextDouble()) {
            sum += src.nextDouble();
            count++;
        }
        else {
            String str = src.next();
            if(str.equals("готово")) break;
            else {
                System.out.println("Ошибка формата файла.");
                return;
            }
        }
    }
}
}

```

С целью продемонстрировать закрытие объекта типа `Scanner` в приведенных далее примерах программ метод `close()` вызывается явным образом. Это позволяет также скомпилировать их в версиях Java, предшествующих `JDK 7`. Но оператор `try` с ресурсами упрощает прикладной код и позволяет предотвратить возможные ошибки, поэтому его рекомендуется применять при написании нового кода.

Следует также заметить, что ради краткости примеров программ, представленных в этом разделе, исключения, возникающие при вводе-выводе, просто генерируются в теле метода `main()`. Но в реальном коде они, как правило, обрабатываются непосредственно.

Класс `Scanner` можно использовать для чтения разнотипных вводимых данных, даже если порядок их следования заранее неизвестен. Для этого достаточно выяснить, какого типа данные доступны, прежде чем их читать. Рассмотрим в качестве примера следующую программу:

```
// Использовать класс Scanner для чтения разнотипных данных из файла
import java.util.*;
import java.io.*;

class ScanMixed {
    public static void main(String args[]) throws IOException {
        int i;
        double d;
        boolean b;
        String str;

        // вывести данные в файл
        FileWriter fout = new FileWriter("test.txt");
        fout.write("Тестирование Scanner 10 12.2 один true два false");
        fout.close();

        FileReader fin = new FileReader("Test.txt");

        Scanner src = new Scanner(fin);

        // читать данные до конца файла
        while(src.hasNext()) {
            if(src.hasNextInt()) {
                i = src.nextInt();
                System.out.println("int: " + i);
            }
            else if(src.hasNextDouble()) {
                d = src.nextDouble();
                System.out.println("double: " + d);
            }
            else if(src.hasNextBoolean()) {
                b = src.nextBoolean();
                System.out.println("boolean: " + b);
            }
            else {
                str = src.next();
                System.out.println("String: " + str);
            }
        }

        src.close();
    }
}
```

Ниже приведен результат, выводимый данной программой.

```
String: Тестирование
String: Scanner
int: 10
double: 12.2
String: один
boolean: true
String: два
boolean: false
```

При чтении разнотипных данных, как в данном примере программы, следует внимательнее следить за порядком, в котором вызываются методы `next`. Так, если поменять в цикле порядок вызова методов `nextInt()` и `nextDouble()`, то оба числовых значения будут прочитаны как относящиеся к типу `double`, поскольку метод `nextDouble()` обнаруживает совпадение с любой символьной строкой, содержащей число, которое может быть представлено типом `double`.

Установка разделителей

Для того чтобы определить, где начинаются и оканчиваются лексемы, в классе `Scanner` применяются *разделители*. По умолчанию в качестве разделителей выбираются пробельные символы, как было показано в предыдущих примерах. Но тип разделителей можно изменить, вызвав метод `useDelimiters()`, общие формы которого приведены ниже.

```
Scanner useDelimiter(String шаблон)
Scanner useDelimiter(Pattern шаблон)
```

Здесь параметр *шаблон* обозначает регулярное выражение, определяющее набор разделителей. В качестве примера ниже приведена переделанная версия представленной ранее программы для чтения из списка чисел, разделяемых запятыми и любым количеством пробелов.

```
// Использовать класс Scanner для вычисления среднего
// из списка чисел, разделяемых запятыми
import java.util.*;
import java.io.*;
class SetDelimiters {
    public static void main(String args[]) throws IOException {
        int count = 0;
        double sum = 0.0;

        // вывести данные в файл
        FileWriter fout = new FileWriter("test.txt");

        // а теперь сохранить данные в списке, разделив их запятыми
        fout.write("2, 3.4, 5,6, 7.4, 9.1, 10.5, готово");
        fout.close();
        FileReader fin = new FileReader("Test.txt");
        Scanner src = new Scanner(fin);

        // установить в качестве разделителей запятые и пробелы
        src.useDelimiter(", *");

        // читать и суммировать числовые значения
        while(src.hasNext()) {
            if(src.hasNextDouble()) {
                sum += src.nextDouble();
                count++;
            }
            else {
                String str = src.next();
                if(str.equals("готово")) break;
                else {
                    System.out.println("Ошибка формата файла.");
                    return;
                }
            }
        }
        src.close();
        System.out.println("Среднее равно " + sum / count);
    }
}
```

В этой версии программы числа, записанные в файл `test.txt`, разделяются запятыми и пробелами. Указанный шаблон разделителей `", *` предписывает обь-

екту типа `Scanner` воспринимать наличие запятой, отсутствие или наличие пробелов во вводимых данных как разделители. Эта версия программы выводит такой же результат, как и предыдущая ее версия.

Текущий шаблон разделителей можно получить, вызвав метод `delimiter()`. Ниже приведена общая форма этого метода.

```
Pattern delimiter()
```

Прочие средства класса `Scanner`

В классе `Scanner` определяется ряд других методов, помимо упомянутых ранее. В частности, одним из наиболее полезных в некоторых случаях является метод `findInLine()`. Его общие формы представлены ниже.

```
String findInLine(Pattern шаблон)
String findInLine(String шаблон)
```

Этот метод осуществляет поиск на совпадение с указанным шаблоном в следующей строке текста. Если совпадение обнаружено, то соответствующая этому шаблону лексема употребляется и возвращается. В противном случае возвращается пустое значение `null`. Это метод действует независимо от установленного набора разделителей. Он удобен, если требуется обнаружить совпадение с конкретным шаблоном. Так, в следующем примере программы сначала обнаруживается поле возраста во введенной символьной строке, а затем выводится его содержимое.

```
// Продемонстрировать применение метода findInLine()
import java.util.*;
class FindInLineDemo {
    public static void main(String args[]) {
        String instr = "Имя: Том Возраст: 28 ID: 77";
        Scanner conin = new Scanner(instr);

        // найти поле возраста и вывести его содержимое
        conin.findInLine("Возраст:");
        if(conin.hasNext())
            System.out.println(conin.next());
        else
            System.out.println("Ошибка!");
        conin.close();
    }
}
```

В результате выполнения этой программы будет выведено значение **28**. В этой программе метод `findInLine()` вызывается для поиска совпадения с шаблоном "Возраст:". Как только совпадение будет обнаружено, читается следующая лексема со значением из поля возраста.

С методом `findInLine()` связан метод `findWithinHorizon()`. Ниже приведены общие формы его объявления.

```
String findWithinHorizon(Pattern шаблон, int количество)
String findWithinHorizon(String шаблон, int количество)
```

Этот метод пытается обнаружить совпадение с указанным шаблоном в последующем количестве символов. При удачном исходе метод `findWithinHorizon()` возвращает результат совпадения с шаблоном, а иначе — пустое значение `null`.

Если параметр *количество* принимает нулевое значение, то поиск осуществляется во всех вводимых данных до тех пор, пока не будет обнаружено совпадение или достигнут конец вводимых данных.

Чтобы пропустить шаблон, достаточно вызвать метод `skip()`. Ниже приведены общие формы его объявления.

```
Scanner skip(Pattern шаблон)
```

```
Scanner skip(String шаблон)
```

Если обнаружено совпадение с заданным шаблоном, метод `skip()` просто пропускает его и возвращает ссылку на вызывающий объект. Если же совпадение с шаблоном не обнаружено, метод `skip()` генерирует исключение типа `NoSuchElementException`.

В классе `Scanner` имеется также метод `radix()`, возвращающий основание системы счисления по умолчанию, используемое в этом классе для чтения чисел; метод `useRadix()`, задающий основание системы счисления; метод `reset()`, устанавливающий сканер в исходное состояние; а также метод `close()`, закрывающий сканер.

Классы `ResourceBundle`, `ListResourceBundle` и `PropertyResourceBundle`

В состав пакета `java.util` входят три класса, предназначенные для интернационализации прикладных программ. Первым из них является абстрактный класс `ResourceBundle`. В нем определяются методы, позволяющие управлять коллекцией ресурсов, зависящих от региональных настроек, например, символьных строк, используемых в качестве меток в элементах пользовательского интерфейса прикладных программ. Для этого можно определить два или более набора символьных строк, переведенных на разные языки (например, английский, немецкий или китайский), причем каждый набор переведенных строк будет находиться в отдельном комплекте ресурсов. Требующийся комплект ресурсов можно затем загружать в соответствии с текущими региональными настройками и использовать переведенные строки для построения пользовательского интерфейса прикладной программы.

Комплекты ресурсов обозначаются *именем семейства*, называемым иначе *базовым именем*. К имени семейства может быть добавлен двухсимвольный *код языка*, обозначающий конкретный язык. В этом случае используется данная версия комплекта ресурсов, если запрошенные региональные настройки соответствуют коду языка. Например, комплект ресурсов под именем семейства `SampleRB` может иметь немецкую версию `SampleRB_de` и русскую версию `SampleRB_ru`. (Обратите внимание на то, что знак подчеркивания связывает имя семейства с кодом языка.) Таким образом, будет применяться комплект ресурсов `SampleRB_de`, если `Locale.GERMAN` — текущие региональные настройки.

Имеется также возможность задать конкретные варианты языка, которые относятся к определенной стране, указав код страны после кода языка. *Код страны* — это двухсимвольный идентификатор прописными буквами, например, `AU` для Австрии или `IN` для Индии. Коду страны также предшествует знак подчеркивания, когда он связывается с именем комплекта ресурсов. Комплект ресурсов, име-

ющий только имя семейства, применяется по умолчанию. Он выбирается, когда недоступны комплекты ресурсов для поддержки конкретных языков.

На заметку! Коды языков определены в стандарте ISO 639, а коды стран — в стандарте ISO 3166.

Методы, определенные в классе `ResourceBundle`, перечислены в табл. 19.20. Следует, однако, иметь в виду, что пустые ключи не допускаются, и поэтому некоторые методы генерируют исключение типа `NullPointerException`, если они получают пустой ключ. Следует также обратить внимание на вложенный класс `ResourceBundle.Control`, внедренный в версии Java SE 6 и предназначенный для управления процессом загрузки комплектов ресурсов.

Таблица 19.20. Методы из класса `ResourceBundle`

Метод	Описание
<code>static final void clearCache()</code>	Удаляет из кеша все комплекты ресурсов, загружаемые по умолчанию загрузчиком классов
<code>static final void clearCache(ClassLoader загрузчик)</code>	Удаляет из кеша все комплекты ресурсов, загружаемые указанным <i>загрузчиком</i>
<code>boolean containsKey(String k)</code>	Возвращает логическое значение <code>true</code> , если заданный ключ <i>k</i> находится в вызывающем комплекте ресурсов (или его родителе)
<code>String getBaseBundleName()</code>	Возвращает базовое имя комплекта ресурсов, если таковое имеется, а иначе — пустое значение <code>null</code> (добавлен в версии JDK 8)
<code>static final ResourceBundle getBundle(String имя_семейства)</code>	Загружает комплект ресурсов, имеющий указанное <i>имя_семейства</i> , используя региональные настройки и загрузчик классов по умолчанию. Генерирует исключение типа <code>MissingResourceException</code> , если не найден комплект ресурсов, совпадающий с указанным <i>именем_семейства</i>
<code>static final ResourceBundle getBundle(String имя_семейства, Locale региональные_настройки)</code>	Загружает комплект ресурсов, имеющий указанное <i>имя_семейства</i> , используя заданные <i>региональные_настройки</i> и загрузчик классов по умолчанию. Генерирует исключение типа <code>MissingResourceException</code> , если не найден комплект ресурсов, совпадающий с указанным <i>именем_семейства</i>
<code>static ResourceBundle getBundle(String имя_семейства, Locale региональные_настройки, ClassLoader загрузчик)</code>	Загружает комплект ресурсов, имеющий указанное <i>имя_семейства</i> , используя заданные <i>региональные_настройки</i> и <i>загрузчик</i> классов. Генерирует исключение типа <code>MissingResourceException</code> , если не найден комплект ресурсов, совпадающий с указанным <i>именем_семейства</i>
<code>static final ResourceBundle getBundle(String имя_семейства, ResourceBundle.Control контроль)</code>	Загружает комплект ресурсов, имеющий указанное <i>имя_семейства</i> , используя региональные настройки и загрузчик классов по умолчанию. Процесс загрузки выполняется под управлением заданного объекта <i>контроля</i> . Генерирует исключение типа <code>MissingResourceException</code> , если не найден комплект ресурсов, совпадающий с указанным <i>именем_семейства</i>

Метод	Описание
static final ResourceBundle getBundle(String <i>имя_семейства</i> , Locale <i>региональные_настройки</i> , ResourceBundle.Control <i>контроль</i>)	Загружает комплект ресурсов, имеющий указанное <i>имя_семейства</i> , используя заданные <i>региональные_настройки</i> и загрузчик классов по умолчанию. Процесс загрузки выполняется под управлением заданного объекта <i>контроль</i> . Генерирует исключение типа MissingResourceException , если не найден комплект ресурсов, совпадающий с указанным <i>именем_семейства</i>
static ResourceBundle getBundle(String <i>имя_семейства</i> , Locale <i>региональные_настройки</i> , ClassLoader <i>загрузчик</i> , ResourceBundle.Control <i>контроль</i>)	Загружает комплект ресурсов, имеющий указанное <i>имя_семейства</i> , используя заданные <i>региональные_настройки</i> и <i>загрузчик</i> классов. Процесс загрузки выполняется под управлением заданного объекта <i>контроль</i> . Генерирует исключение типа MissingResourceException , если не найден комплект ресурсов, совпадающий с указанным <i>именем_семейства</i>
abstract Enumeration <String>getKeys()	Возвращает ключи из комплекта ресурсов в виде перечисления символьных строк. Получаются также ключи из любого родителя данного комплекта ресурсов
Locale getLocale()	Возвращает региональные настройки, поддерживаемые в комплекте ресурсов
final Object getObject(String k)	Возвращает объект, связанный с заданным ключом <i>k</i> . Генерирует исключение типа MissingResourceException , если заданный ключ <i>k</i> не найден в комплекте ресурсов
final String getString(String k)	Возвращает символьную строку, связанную с заданным ключом <i>k</i> . Генерирует исключение типа MissingResourceException , если заданный ключ <i>k</i> не найден в комплекте ресурсов. Генерирует также исключение типа ClassCastException , если объект, связанный с заданным ключом <i>k</i> , не является символьной строкой
final String[] getStringArray(String k)	Возвращает массив символьных строк, связанных с заданным ключом <i>k</i> . Генерирует исключение типа MissingResourceException , если заданный ключ <i>k</i> не найден в комплекте ресурсов. Генерирует также исключение типа ClassCastException , если объект, связанный с заданным ключом <i>k</i> , не является массивом символьных строк
protected abstract Object handleGetObject(String k)	Возвращает объект, связанный с заданным ключом <i>k</i> . Возвращает пустое значение null , если заданный ключ <i>k</i> не найден в комплекте ресурсов
protected Set<String> handleKeySet()	Возвращает ключи из комплекта ресурсов в виде множества. Из родителя данного комплекта ресурсов ключи не извлекаются. Кроме того, ключи со значениями null не возвращаются
Set<String> keySet()	Возвращает ключи из комплекта ресурсов в виде множества символьных строк. Получаются также ключи из любого родителя данного комплекта ресурсов

Окончание табл. 19.20

Метод	Описание
protected void setParent (ResourceBundle родитель)	Устанавливает <i>родитель</i> в качестве родительского комплекта для данного комплекта ресурсов. Если ключ не найден в вызывающем комплекте ресурсов, то его поиск продолжится в родительском комплекте ресурсов

У класса ResourceBundle имеются два подкласса. Первым из них является класс PropertyResourceBundle, управляющий ресурсами с помощью файлов свойств. Этот класс не вводит свои методы. Вторым является абстрактный класс ListResourceBundle, управляющий ресурсами в массиве пар “ключ–значение”. У этого класса имеется свой метод getContents(), который должны реализовать все его подклассы. Ниже показано, каким образом объявляется этот метод.

```
protected abstract Object[][] getContents()
```

Этот метод возвращает двухмерный массив, содержащий пары “ключ–значение”, представляющие ресурсы. Ключи должны быть символьными строками, а значения — как правило, символьными строками, но они могут быть также представлены объектами других типов.

Рассмотрим пример, демонстрирующий применение комплекта ресурсов под именем семейства SampleRB. Два класса комплектов ресурсов из этого семейства создаются расширением класса ListResourceBundle. Первый из них называется SampleRB и представляет комплект ресурсов по умолчанию для поддержки английского языка:

```
import java.util.*;
public class SampleRB extends ListResourceBundle {
    protected Object[][] getContents() {
        Object[][] resources = new Object[3][2];

        resources[0][0] = "title";
        resources[0][1] = "My Program";

        resources[1][0] = "S5stopText";
        resources[1][1] = "Stop";

        resources[2][0] = "StartText";
        resources[2][1] = "Start";

        return resources;
    }
}
```

Второй класс комплекта ресурсов называется SampleRB_de и содержит перевод содержимого символьных строк на немецкий язык:

```
import java.util.*;

// Версия на немецком языке
public class SampleRB_de extends ListResourceBundle {
    protected Object[][] getContents() {
        Object[][] resources = new Object[3][2];

        resources[0][0] = "title";
        resources[0][1] = "Mein Programm";
```

```

        resources[1][0] = "StopText";
        resources[1][1] = "Anschlag";

        resources[2][0] = "StartText";
        resources[2][1] = "Anfang";

        return resources;
    }
}

```

Применение обоих упомянутых выше комплектов ресурсов демонстрируется в приведенном ниже примере программы. С их помощью выводятся символьные строки, связанные с каждым ключом как для английской версии программы, выбираемой по умолчанию, так и для немецкой ее версии.

```

// Продемонстрировать применение комплектов ресурсов
import java.util.*;

class LRBDemo {
    public static void main(String args[]) {
        // загрузить комплект ресурсов по умолчанию
        ResourceBundle rd = ResourceBundle.getBundle("SampleRB");

        System.out.println("Английская версия программы: ");
        System.out.println("Строка по ключу Title: " +
            rd.getString("title"));
        System.out.println("Строка по ключу StopText: " +
            rd.getString("StopText"));
        System.out.println("Строка по ключу StartText: " +
            rd.getString("StartText"));

        // загрузить комплект ресурсов для поддержки немецкого языка
        rd = ResourceBundle.getBundle("SampleRB", Locale.GERMAN);

        System.out.println("\nНемецкая версия программы: ");
        System.out.println("Строка для ключа Title: " +
            rd.getString("title"));
        System.out.println("Строка по ключу StopText: " +
            rd.getString("StopText"));
        System.out.println("Строка по ключу StartText: " +
            rd.getString("StartText"));
    }
}

```

Эта программа выводит следующий результат:

```

Английская версия программы:
Строка по ключу Title: My Program
Строка по ключу StopText: Stop
Строка по ключу StartText: Start

Немецкая версия программы:
Строка по ключу Title: Mein Programm
Строка по ключу StopText: Anschlag
Строка по ключу StartText: Anfang

```

Прочие служебные классы и интерфейсы

Помимо описанных ранее классов, в пакет `java.util` входят классы, перечисленные в табл. 19.21.

Таблица 19.21. Дополнительные классы из пакета java.util

Класс	Описание
Base64	Поддерживает кодировку Base64. Определены также вложенные классы Encoder и Decoder (добавлен в версии JDK 8)
DoubleSummaryStatistics	Поддерживает сбор статистических данных типа double . Доступны следующие виды статистических данных: среднее, минимальное, максимальное, подсчет и итог (добавлен в версии JDK 8)
EventListenerProxy	Расширяет класс EventListener для передачи дополнительных параметров. Приемники событий рассматриваются в главе 24
EventObject	Суперкласс для всех классов событий. События рассматриваются в главе 24
FormattableFlags	Определяет признаки форматирования, используемые в интерфейсе Formattable
IntSummaryStatistics	Поддерживает сбор статистических данных типа int . Доступны следующие виды статистических данных: среднее, минимальное, максимальное, подсчет и итог (добавлен в версии JDK 8)
Objects	Определяет различные методы для обращения с объектами
PropertyPermission	Управляет правами доступа к свойствам
ServiceLoader	Предоставляет средства для поиска поставщиков услуг
UUID	Инкапсулирует универсальные уникальные идентификаторы (UUID) и управляет ими

Интерфейсы, перечисленные в табл. 19.22, также входят в состав пакета java.util.

Таблица 19.22. Дополнительные интерфейсы из пакета java.util

Интерфейс	Описание
EventListener	Обозначает, что класс является приемником событий. События рассматриваются в главе 24
Formattable	Описывает класс, обеспечивающий специальное форматирование

Подпакеты, входящие в состав пакета java.util

В Java определяются следующие подпакеты, входящие в состав пакета java.util:

- java.util.concurrent
- java.util.concurrent.atomic
- java.util.concurrent.locks

- `java.util.function`
- `java.util.jar`
- `java.util.logging`
- `java.util.prefs`
- `java.util.regex`
- `java.util.spi`
- `java.util.stream`
- `java.util.zip`

Все они вкратце описаны далее.

Пакеты `java.util.concurrent`, `java.util.concurrent.atomic`, `java.util.concurrent.locks`

Пакет `java.util.concurrent` вместе с двумя своими подпакетами `java.util.concurrent.atomic` и `java.util.concurrent.locks` служит для поддержки параллельного программирования. Все эти пакеты предоставляют высокопроизводительную альтернативу применению встроенных в Java средств синхронизации, когда требуются потокобезопасные операции. С версии JDK 7 в пакете `java.util.concurrent` поддерживается также каркас Fork/Join Framework. Подробнее эти пакеты рассматриваются в главе 28.

Пакет `java.util.function`

В этом пакете предопределен ряд функциональных интерфейсов, предназначенных для создания лямбда-выражений или ссылок на методы. Эти функциональные интерфейсы широко применяются в прикладном программном интерфейсе Java API. Все функциональные интерфейсы из пакета `java.util.function` перечислены в табл. 19.23 вместе с кратким описанием их абстрактных методов. Следует, однако, иметь в виду, что для расширения функциональных возможностей в некоторых из этих интерфейсов определяются также методы по умолчанию или статические методы. Поэтому вам придется изучить их самостоятельно. (Подробнее о функциональных интерфейсах см. в главе 15.)

Таблица 19.23. Функциональные интерфейсы из пакета `java.util.function` и их абстрактные методы

Интерфейс	Абстрактный метод
<code>BiConsumer<T, U></code>	<code>void accept(T tVal, U uVal)</code> Описание: оперирует аргументами <i>tVal</i> и <i>uVal</i>
<code>BiFunction<T, U, R></code>	<code>R apply(T tVal, U uVal)</code> Описание: оперирует аргументами <i>tVal</i> и <i>uVal</i> и возвращает полученный результат

Продолжение табл. 19.23

Интерфейс	Абстрактный метод
BinaryOperator<T>	T apply(T val1, T val2) Описание: оперирует двумя однотипными объектами и возвращает полученный результат того же самого типа
BiPredicate<T, U>	boolean test(T tVal, U uVal) Описание: возвращает логическое значение true , если оба аргумента, <i>tVal</i> и <i>uVal</i> , удовлетворяют условию, задаваемому методом test() , а иначе – логическое значение false
BooleanSupplier	boolean getAsBoolean() Описание: возвращает логическое значение
Consumer<T>	void accept(T val) Описание: оперирует аргументом <i>val</i>
DoubleBinaryOperator	double applyAsDouble(double val1, double val2) Описание: оперирует двумя аргументами, <i>val1</i> и <i>val2</i> , типа double и возвращает результат того же самого типа
DoubleConsumer	void accept(double val) Описание: оперирует аргументом <i>val</i>
DoubleFunction<R>	R apply(double val) Описание: оперирует аргументом <i>val</i> типа double и возвращает результат
DoublePredicate	boolean test(double val) Описание: возвращает логическое значение true , если аргумент <i>val</i> удовлетворяет условию, задаваемому методом test() , а иначе – логическое значение false
DoubleSupplier	double getAsDouble() Описание: возвращает результат типа double
DoubleToIntFunction	int applyAsInt(double val) Описание: оперирует аргументом <i>val</i> типа double и возвращает результат типа int
DoubleToLongFunction	long applyAsLong(double val) Описание: оперирует аргументом <i>val</i> типа double и возвращает результат типа long
DoubleUnaryOperator	double applyAsDouble(double val) Описание: оперирует аргументом <i>val</i> типа double и возвращает результат типа double
Function<T, R>	R apply(T val) Описание: оперирует аргументом <i>val</i> и возвращает результат
IntBinaryOperator	int applyAsInt(int val1, int val2) Описание: оперирует аргументами <i>val1</i> и <i>val2</i> типа int и возвращает результат того же самого типа

Интерфейс	Абстрактный метод
IntConsumer	int accept(int val) Описание: оперирует аргументом <i>val</i>
IntFunction<R>	R apply(int val) Описание: оперирует аргументом <i>val</i> типа int и возвращает результат
IntPredicate	boolean test(int val) Описание: возвращает логическое значение true , если аргумент <i>val</i> удовлетворяет условию, задаваемому методом test() , а иначе — логическое значение false
IntSupplier	int getAsInt() Описание: возвращает результат типа int
IntToDoubleFunction	double applyAsDouble(int val) Описание: оперирует аргументом <i>val</i> типа int и возвращает результат типа double
IntToLongFunction	long applyAsLong(int val) Описание: оперирует аргументом <i>val</i> типа int и возвращает результат типа long
IntUnaryOperator	int applyAsInt(int val) Описание: оперирует аргументом <i>val</i> типа int и возвращает результат типа int
LongBinaryOperator	long applyAsLong(long val1, long val2) Описание: оперирует аргументам <i>val1</i> и <i>val2</i> типа long и возвращает результат того же типа
LongConsumer	void accept(long val) Описание: оперирует аргументом <i>val</i>
LongFunction<R>	R apply(long val) Описание: оперирует аргументом <i>val</i> типа long и возвращает результат
LongPredicate	boolean test(long val) Описание: возвращает логическое значение true , если аргумент <i>val</i> удовлетворяет условию, задаваемому методом test() , а иначе — логическое значение false
LongSupplier	long getAsLong() Описание: возвращает результат типа long
LongToDoubleFunction	double applyAsDouble(long val) Описание: оперирует аргументом <i>val</i> типа long и возвращает результат типа double
LongToIntFunction	int applyAsInt(long val) Описание: оперирует аргументом <i>val</i> типа long и возвращает результат типа int

Окончание табл. 19.23

Интерфейс	Абстрактный метод
LongUnaryOperator	long applyAsLong(long val) Описание: оперирует аргументом <i>val</i> типа long и возвращает результат типа long
ObjDoubleConsumer<T>	void accept(T val1, double val2) Описание: оперирует аргументом <i>val1</i> типа T и аргументом <i>val2</i> типа double
ObjIntConsumer<T>	void accept(T val1, int val2) Описание: оперирует аргументом <i>val1</i> типа T и аргументом <i>val2</i> типа int
ObjLongConsumer<T>	void accept(T val1, long val2) Описание: оперирует аргументом <i>val1</i> типа T и аргументом <i>val2</i> типа long
Predicate<T>	boolean test(T val) Описание: возвращает логическое значение true , если аргумент <i>val</i> удовлетворяет условию, задаваемому методом test() , а иначе — логическое значение false
Supplier<T>	T get() Описание: возвращает объект типа T
ToDoubleBiFunction<T, U>	double applyAsDouble(T tVal, U uVal) Описание: оперирует аргументами <i>tVal</i> и <i>uVal</i> и возвращает результат типа double
ToDoubleFunction<T>	double applyAsDouble(T val) Описание: оперирует аргументом <i>val</i> и возвращает результат типа double
ToIntBiFunction<T, U>	int applyAsInt(T tVal, U uVal) Описание: оперирует аргументами <i>tVal</i> и <i>uVal</i> и возвращает результат типа int
ToIntFunction<T>	int applyAsInt(T val) Описание: оперирует аргументом <i>val</i> и возвращает результат типа int
ToLongBiFunction<T, U>	long applyAsLong(T tVal, U uVal) Описание: оперирует аргументами <i>tVal</i> и <i>uVal</i> и возвращает результат типа long
ToLongFunction<T>	long applyAsLong(T val) Описание: оперирует аргументом <i>val</i> и возвращает результат типа long
UnaryOperator<T>	T apply(T val) Описание: оперирует аргументом <i>val</i> и возвращает результат

Пакет `java.util.jar`

Предоставляет средства для чтения и записи архивных файлов формата Java Archive (JAR).

Пакет `java.util.logging`

Обеспечивает поддержку журналов регистрации, которые могут быть использованы для записи операций, выполняемых в программе, а также для обнаружения ошибок и отладки программ.

Пакет `java.util.prefs`

Обеспечивает поддержку глобальных параметров, настраиваемых пользователями. Как правило, применяется для поддержки разных конфигураций программ.

Пакет `java.util.regex`

Обеспечивает поддержку, требующуюся для обработки регулярных выражений. Подробнее он рассматривается в главе 30.

Пакет `java.util.spi`

Обеспечивает поддержку поставщиков услуг.

Пакет `java.util.stream`

Содержит прикладной программный интерфейс API потоков ввода-вывода в Java, внедренный в версии JDK 8. Подробнее этот прикладной программный интерфейс рассматривается в главе 29.

Пакет `java.util.zip`

Предоставляет средства для чтения и записи архивных файлов в распространенных форматах ZIP и GZIP. Доступны также потоки ввода-вывода данных, архивируемых в форматах ZIP и GZIP.

Пакет `java.io` для ввода-вывода

Эта глава посвящена пакету `java.io`, поддерживающему операции ввода-вывода. В главе 13 был сделан краткий обзор системы ввода-вывода в Java, включая основные методики чтения и записи файлов, обработки исключений ввода-вывода и закрытия файла. А в этой главе система ввода-вывода в Java рассматривается более подробно.

Как давно уже известно всем программистам, большинство программ не в состоянии выполнять свои функции, не имея доступа к внешним данным. Данные извлекаются из источника *ввода*. А результат выполнения программы направляется адресату *вывода*. В Java эти понятия определяются очень широко. Например, источником ввода или адресатом вывода может служить сетевое соединение, буфер памяти или дисковый файл, и всеми ими можно манипулировать с помощью классов ввода-вывода в Java. И хотя все устройства ввода-вывода отличаются физически, все они описываются единой абстракцией — потоком ввода-вывода. Как пояснялось главе 13, *поток ввода-вывода* — это логический объект, который предоставляет или потребляет информацию. Поток ввода-вывода присоединяется к физическому устройству системой ввода-вывода в Java. Все потоки ввода-вывода ведут себя сходным образом, несмотря на то, что физические устройства, к которым они присоединены, радикально отличаются.

На заметку! Потоковая система ввода-вывода, входящая в пакет `java.io` и рассматриваемая в этой главе, была составной частью Java, начиная с первого выпуска и широко применяется до сих пор. Но в версии 1.4 в Java была внедрена вторая система ввода-вывода. Она называется NIO (что первоначально означало New I/O, т.е. новый ввод-вывод). Система NIO входит в пакет `java.nio` и его подпакеты и подробно рассматривается в главе 21.

На заметку! Не следует путать потоки ввода-вывода из рассматриваемой здесь системы ввода-вывода с потоками данных из нового прикладного интерфейса API, внедренного в версии JDK 8. Несмотря на концептуальную связь между ними, это все-таки разные средства ввода-вывода. Следовательно, под термином *поток ввода-вывода* здесь подразумевается поток из системы ввода-вывода.

Классы и интерфейсы ввода-вывода

Ниже перечислены классы ввода-вывода, определенные в пакете `java.io`.

BufferedInputStream	FileWriter	PipedOutputStream
BufferedOutputStream	FilterInputStream	PipedReader
BufferedReader	FilterOutputStream	PipedWriter
BufferedWriter	FilterReader	PrintStream
ByteArrayInputStream	FilterWriter	PrintWriter
ByteArrayOutputStream	InputStream	PushbackInputStream
CharArrayReader	InputStreamReader	PushbackReader
CharArrayWriter	LineNumberReader	RandomAccessFile
Console	ObjectInputStream	Reader
DataInputStream	ObjectInputStream.GetField	SequenceInputStream
DataOutputStream	ObjectOutputStream	SerializablePermission
File	ObjectOutputStream.PutField	StreamTokenizer
FileDescriptor	ObjectStreamClass	StringReader
FileInputStream	ObjectStreamField	StringWriter
FileOutputStream	OutputStream	Writer
FilePermission	OutputStreamWriter	
FileReader	PipedInputStream	

Пакет `java.io` содержит также два устаревших класса, которые отсутствуют среди перечисленных выше. Это классы `LineNumberInputStream` и `StringBufferInputStream`. Их не рекомендуется употреблять в новом коде.

Кроме того, в пакете `java.io` определены следующие интерфейсы:

Closeable	FileFilter	ObjectInputValidation
DataInput	FilenameFilter	ObjectOutput
DataOutput	Flushable	ObjectStreamConstants
Externalizable	ObjectInput	Serializable

Как видите, в пакете `java.io` имеется немало классов и интерфейсов. Они реализуют потоки ввода-вывода байтов и символов, а также сериализацию объектов (их сохранение и восстановление). В этой главе рассматриваются наиболее употребительные составляющие системы ввода-вывода в Java. Начнем их обсуждение с класса `File` — одного из наиболее характерных для ввода-вывода.

Класс File

Большинство классов, определенных в пакете `java.io`, оперируют потоками ввода-вывода, чего нельзя сказать о классе `File`. Он оперирует непосредственно файлами и взаимодействует с файловой системой. Следовательно, в классе `File` не определяется, каким образом данные извлекаются и сохраняются в файлах, но описываются свойства самих файлов. Объект класса `File` служит для получения таких сведений о файле на диске, как права доступа, время, дата и путь к каталогу, или манипулирования этими сведениями, а также для перемещения по иерархиям подкаталогов.

На заметку! Интерфейс `Path` и класс `Files`, входящие в систему ввода-вывода NIO, нередко служат эффективной альтернативой классу `File`. Более подробно система ввода-вывода NIO рассматривается в главе 21.

Файлы служат первичными источниками и адресатами данных во многих программах. Несмотря на строгие ограничения, накладываемые на использование файлов в апплетах из соображений безопасности, файлы по-прежнему остаются центральным ресурсом для хранения постоянной и обмениваемой информации. Каталог в Java интерпретируется как объект класса `File` с единственным дополнительным свойством — списком имен файлов, которые могут быть получены методом `list()`.

Для создания объектов класса `File` можно воспользоваться следующими конструкторами:

```
File(String путь_к_каталогу)
File(String путь_к_каталогу, String имя_файла)
File(File объект_каталога, String имя_файла)
File(URI объект_URI)
```

где параметр `путь_к_каталогу` обозначает путь к файлу; параметр `имя_файла` — имя конкретного файла или подкаталога; параметр `объект_каталога` — объект типа `File`, задающий каталог; а параметр `объект_URI` — объект типа `URI`, описывающий файл.

В приведенном ниже примере создаются три файла в переменных `f1`, `f2` и `f3`. Первый объект типа `File` создается в каталоге, указываемом в качестве единственного аргумента конструктора; второй объект — с указанием пути и имени файла в качестве двух аргументов конструктора; третий объект — с указанием пути, присваиваемому переменной `f1`, а также имени файла, причем переменная `f3` ссылается на тот же самый объект типа `File`, обозначающий файл, что и переменная `f2`.

```
File f1 = new File("/");
File f2 = new File("/", "autoexec.bat");
File f3 = new File(f1, "autoexec.bat");
```

На заметку! В Java правильно интерпретируются разделители путей к файлам, которые отличаются в UNIX и Windows. Даже если в качестве такого разделителя используется знак косой черты (/) в версии Java для Windows, путь к файлу будет все равно сформирован правильно. Не следует, однако, забывать, что для употребления знака обратной косой черты (\) в символьных строках под Windows его следует экранировать в виде управляющей последовательности (\\).

В классе `File` определяется немало методов для получения стандартных свойств файла, представленного объектом этого класса. Например, метод `getName()` возвращает имя файла, метод `getParent()` — имя родительского каталога, а метод `exists()` — логическое значение `true`, если файл существует, а иначе — логическое значение `false`. В приведенном ниже примере программы демонстрируется применение некоторых методов из класса `File`. При этом подразумевается, что в корневом каталоге существует каталог `java` с файлом `COPYRIGHT`.

```
// Продемонстрировать применение некоторых методов из класса File
import java.io.File;
class FileDemo {
    static void p(String s) {
        System.out.println(s);
    }
    public static void main(String args[]) {
        File f1 = new File("/java/COPYRIGHT");
        p("Имя файла: " + f1.getName());
        p("Путь: " + f1.getPath());
        p("Абсолютный путь: " + f1.getAbsolutePath());
        p("Родительский каталог: " + f1.getParent());
        p(f1.exists() ? "существует" : "не существует");
        p(f1.canWrite() ? "доступен для записи" :
            "не доступен для записи");
        p(f1.canRead() ? "доступен для чтения" :
            "не доступен для чтения");
        p(f1.isDirectory() ? "является каталогом" :
            "не является каталогом");
        p(f1.isFile() ? "является обычным файлом" :
            "может быть именованным каналом");
        p(f1.isAbsolute() ? "является абсолютным" :
            "не является абсолютным");
        p("Последнее изменение в файле: " + f1.lastModified());
        p("Размер: " + f1.length() + " байт");
    }
}
```

Ниже приведен пример выполнения данной программы.

```
Имя файла: COPYRIGHT
Путь: \java\COPYRIGHT
Абсолютный путь: \java\COPYRIGHT
Родительский каталог: \java
существует
доступен для записи
доступен для чтения
не является каталогом
является обычным файлом
не является абсолютным
Последнее изменение в файле: 1282832030047
Размер: 695 байт
```

Назначение большинства методов из класса `File` самоочевидно, но методы `isFile()` и `isAbsolute()` требуют дополнительных пояснений. В частности, метод `isFile()` возвращает логическое значение `true`, если он вызывается с файлом, а если с каталогом, то логическое значение `false`. Кроме того, метод `isFile()` возвращает логическое значение `false` для некоторых специальных файлов, например, драйверов устройств и именованных каналов, и поэтому, вызывая этот метод, можно убедиться, что файл действительно ведет себя как файл. А метод `isAbsolute()` возвращает логическое значение `true`, если файл имеет абсолютный путь, а если относительный путь, то логическое значение `false`.

В классе `File` имеются также два полезных служебных метода. Первым из них является метод `renameTo()`, общая форма объявления которого показана ниже.

```
boolean renameTo(File новое_имя)
```

Здесь имя файла, указанное в качестве параметра *новое_имя*, становится новым именем вызывающего объекта типа `File`. Метод `renameTo()` возвращает логическое значение `true` при удачном исходе переименования файла, а если файл не может быть переименован, поскольку в качестве нового имени указано имя уже существующего файла, то логическое значение `false`.

Второй служебный метод, `delete()`, удаляет с диска файл по пути, который предоставляется вызывающим объектом типа `File`. Ниже приведена общая форма объявления этого метода.

```
boolean delete()
```

С помощью метода `delete()` можно также удалить каталог, если он пуст. Метод `delete()` возвращает логическое значение `true`, если ему удастся удалить файл, а иначе — логическое значение `false`. Другие полезные методы из класса `File` перечислены в табл. 20.1.

Таблица 20.1. Другие полезные методы из класса `File`

Метод	Описание
<code>void deleteOnExit()</code>	Удаляет файл, связанный с вызывающим объектом, по завершении работы виртуальной машины JVM
<code>long getFreeSpace()</code>	Возвращает количество свободных байтов, доступных в разделе запоминающего устройства, связанном с вызывающим объектом
<code>long getTotalSpace()</code>	Возвращает емкость раздела запоминающего устройства, связанного с вызывающим объектом
<code>long getUsableSpace()</code>	Возвращает количество пригодных для употребления свободных байтов, доступных в разделе запоминающего устройства, связанном с вызывающим объектом
<code>boolean isHidden()</code>	Возвращает логическое значение true , если вызывающий файл является скрытым, а иначе — логическое значение false
<code>boolean setLastModified(long миллисекунд)</code>	Устанавливает временную метку для вызываемого файла в виде заданного количества <i>миллисекунд</i> , прошедших с 1 января 1970 г., в формате времени UTC
<code>boolean setReadOnly()</code>	Делает вызывающий файл доступным только для чтения

Кроме того, имеются методы, помечающие файлы как доступные только для чтения, записи или выполнения. А поскольку класс `File` реализует интерфейс `Comparable`, то в нем поддерживается также метод `compareTo()`.

В версии JDK 7 класс `File` был дополнен методом `toPath()`, который объявляется следующим образом:

```
Path toPath()
```

Метод `toPath()` возвращает объект типа `Path`, который представляет файл, инкапсулируемый вызываемым объектом типа `File`. (Иными словами, метод `toPath()` преобразует объект типа `File` в объект типа `Path`.) Интерфейс `Path` входит в состав пакета `java.nio.file` и является составной частью системы

ввода-вывода NIO. Таким образом, метод `toPath()` наводит мост между старым классом `File` и новым интерфейсом `Path`. (Подробнее об интерфейсе `Path` речь пойдет в главе 21.)

Каталоги

Каталог является объектом типа `File`, содержащим список других файлов и каталогов. После создания объекта типа `File` как каталога его метод `isDirectory()` возвратит логическое значение `true`. В таком случае для этого объекта можно вызвать метод `list()`, чтобы извлечь список других находящихся в нем файлов и каталогов. У метода `list()` имеются две общие формы. Первая из них приведена ниже.

String[] list()

Список файлов возвращается в виде массива объектов типа `String`. В следующем примере программы демонстрируется применение метода `list()` для просмотра содержимого каталога:

```
// Использовать каталоги
import java.io.File;

class DirList {
    public static void main(String args[]) {
        String dirname = "/java";
        File f1 = new File(dirname);
        if (f1.isDirectory()) {
            System.out.println("Каталог " + dirname);
            String s[] = f1.list();

            for (int i=0; i < s.length; i++) {
                File f = new File(dirname + "/" + s[i]);
                if (f.isDirectory()) {
                    System.out.println(s[i] + " является каталогом");
                } else {
                    System.out.println(s[i] + " является файлом");
                }
            }
        } else {
            System.out.println(dirname + " не является каталогом");
        }
    }
}
```

Ниже приведен примерный результат, выводимый данной программой (у вас он может оказаться иным, в зависимости от того, что находится в каталоге).

```
Каталог /java
bin является каталогом
lib является каталогом
demo является каталогом
COPYRIGHT является файлом
README является файлом
index.html является файлом
include является каталогом
src.zip является файлом
src является каталогом
```

Применение интерфейса `FilenameFilter`

Нередко требуется ограничить количество файлов, возвращаемых методом `list()`, чтобы включить в их число только те файлы, которые соответствуют определенному образцу имен, или *фильтру*. Для этого следует использовать вторую форму метода `list()`, которая приведена ниже.

```
String[] list(FilenameFilter FFObj)
```

В этой форме параметр `FFObj` обозначает объект класса, реализующего интерфейс `FilenameFilter`. В интерфейсе `FilenameFilter` определяется единственный метод `accept()`, вызываемый один раз с каждым файлом из списка. Его общая форма такова:

```
boolean accept(File каталог, String имя_файла)
```

Метод `accept()` возвращает логическое значение `true` для файлов из указанного каталога, которые должны быть включены в список (т.е. тех файлов, которые совпадают с заданным именем_файла), а для файлов, которые следует исключить из списка, возвращается логическое значение `false`.

Приведенный ниже класс `OnlyExt` реализует интерфейс `FilenameFilter`. Он будет использован в видоизмененной версии программы из предыдущего примера, чтобы ограничить имена файлов, возвращаемых методом `list()`, только теми из них, которые оканчиваются расширением, указанным при создании объекта этого класса.

```
import java.io.*;

public class OnlyExt implements FilenameFilter {
    String ext;

    public OnlyExt(String ext) {
        this.ext = "." + ext;
    }

    public boolean accept(File dir, String name) {
        return name.endsWith(ext);
    }
}
```

Ниже приведена видоизмененная версия данной программы для просмотра файлов в каталоге. Теперь она выведет только файлы с расширением `.html`.

```
// Просмотреть каталог HTML-файлов
import java.io.*;

class DirListOnly {
    public static void main(String args[]) {
        String dirname = "/java";
        File fl = new File(dirname);
        FilenameFilter only = new OnlyExt("html");
        String s[] = fl.list(only);

        for (int i=0; i < s.length; i++) {
            System.out.println(s[i]);
        }
    }
}
```

Альтернативный метод `listFiles()`

Существует разновидность метода `list()` под названием `listFiles()`, которая может оказаться удобной. Ниже приведены сигнатуры метода `listFiles()`.

```
File[] listFiles()
File[] listFiles(FilenameFilter FFObj)
File[] listFiles(FileFilter FObj)
```

Эти формы метода `listFiles()` возвращают список файлов в виде массива объектов типа `File` вместо символьных строк. В первой форме этот метод возвращает все файлы, во второй — только те файлы, которые удовлетворяют условию, задаваемому объектом типа `FilenameFilter` в качестве параметра `FFObj`. Помимо возвращения массива объектов типа `File`, обе эти формы метода `listFiles()` действуют таким же образом, как и эквивалентные им формы метода `list()`.

Третья форма метода `listFiles()` возвращает те файлы, имена путей к которым удовлетворяют условию, задаваемому объектом типа `FileFilter` в качестве параметра `FObj`. В интерфейсе `FileFilter` определяется единственный метод `accept()`, который вызывается один раз для каждого файла из списка. Его общая форма такова:

```
boolean accept(File путь)
```

Метод `accept()` возвращает логическое значение `true` для тех файлов, которые должны быть включены в список (т.е. совпадают с аргументом `путь`), и логическое значение `false` для тех файлов, которые следует исключить из списка.

Создание каталогов

В классе `File` имеются еще два полезных служебных метода: `mkdir()` и `mkdirs()`. В частности, метод `mkdir()` создает каталог, возвращая логическое значение `true` при удачном исходе операции и логическое значение `false` при неудачном. Операция создания каталога может завершиться неудачно по разным причинам. Например, путь, указанный в объекте типа `File`, уже существует, или каталог не может быть создан, потому что полный путь к нему еще не существует. Чтобы создать каталог, путь к которому еще не создан, следует вызвать метод `mkdirs()`. Он создаст как сам каталог, так и все его родительские каталоги.

Интерфейсы `AutoCloseable`, `Closeable` и `Flushable`

Эти три интерфейса имеют большое значение для классов потоков ввода-вывода. Два из них, интерфейсы `Closeable` и `Flushable`, были определены в пакете `java.io` и внедрены в версии JDK 5. А третий, интерфейс `AutoCloseable`, был внедрен в версии JDK 7 и входит в состав пакета `java.lang`.

В версии JDK 7 интерфейс `AutoCloseable` обеспечивает поддержку нового оператора `try` с ресурсами, который автоматизирует процесс закрытия ресур-

са, как поясняется в главе 13. Только объекты классов, реализующих интерфейс `AutoCloseable`, могут управляться оператором `try` с ресурсами. Интерфейс `AutoCloseable` обсуждается в главе 17, но здесь ради удобства приведен краткий его обзор. В интерфейсе `AutoCloseable` определяется единственный метод `close()`:

```
void close() throws Exception
```

Этот метод закрывает вызывающий объект, высвобождая любые ресурсы, которыми он может пользоваться. Метод `close()` вызывается автоматически по завершении оператора `try` с ресурсами, избавляя от необходимости вызывать его явным образом. Интерфейс `AutoCloseable` реализуется всеми классами, открывающими потоки потоков ввода-вывода, и поэтому эти потоки могут быть автоматически закрыты оператором `try` с ресурсами. Автоматическое закрытие потоков ввода-вывода гарантирует правильность их освобождения, когда они больше не нужны, предотвращая тем самым утечку памяти и другие осложнения.

В интерфейсе `Closeable` также определяется метод `close()`. Объекты класса, реализующего интерфейс `Closeable`, могут быть закрыты. Начиная с версии JDK 7 интерфейс `Closeable` расширяет интерфейс `AutoCloseable`. Поэтому в версии JDK 7 любой класс, реализующий интерфейс `Closeable`, реализует также интерфейс `AutoCloseable`.

Объекты класса, реализующего интерфейс `Flushable`, могут принудительно направить буферизованные данные в тот поток вывода, к которому присоединен данный объект. В этом интерфейсе определяется единственный метод `flush()`, как показано ниже.

```
void flush() throws IOException
```

Очистка потока вывода обычно приводит к тому, что буферизованные данные физически выводятся на базовое устройство. Этот интерфейс реализуется всеми классами, способными направлять данные в поток вывода.

Исключения ввода-вывода

Для организации ввода-вывода большое значение имеют два исключения. Первое из них, исключение типа `IOException`, имеет отношение к большинству классов ввода-вывода, описанных в данной главе, поэтому при возникновении ошибки ввода-вывода генерируется исключение типа `IOException`. Если файл нельзя открыть, то, как правило, генерируется исключение типа `FileNotFoundException`. Класс исключения `FileNotFoundException` является производным от класса `IOException`, поэтому оба типа исключений могут быть перехвачены в одном блоке оператора `catch`, предназначенном для перехвата и обработки исключения типа `IOException`. Именно такой подход применяется ради краткости в большинстве примеров программ, представленных в этой главе. Но в своих прикладных программах вам, возможно, будет удобнее обрабатывать их по отдельности.

Для организации ввода-вывода иногда оказывается очень важным еще один класс исключения `SecurityException`. Как пояснялось в главе 13, в тех случаях, когда при-

существует диспетчер безопасности, некоторые классы обращения с файлами генерируют исключение типа `SecurityException` при попытке открыть файл с нарушением безопасности. По умолчанию прикладные программы, запускаемые на выполнение по команде `java`, не пользуются диспетчером безопасности. Поэтому в примерах организации ввода-вывода, представленных в данной книге, не отслеживается возможность генерирования исключения типа `SecurityException`. Но апплеты обычно пользуются диспетчером безопасности, предоставляемым браузером, и поэтому при вводе-выводе в файл из апплета может быть сгенерировано исключение типа `SecurityException`. В таком случае придется обработать и это исключение.

Два способа закрытия потока ввода-вывода

Как правило, поток ввода-вывода следует закрыть, когда он больше не нужен. Если не сделать этого, может произойти утечка памяти и истощение ресурсов. Способы закрытия потока ввода-вывода были описаны в главе 13, но из-за особого значения этих способов напомним их вкратце, прежде чем перейти к рассмотрению классов потоков ввода-вывода.

В версии JDK 7 появились два основных способа, которыми можно закрыть поток ввода-вывода. Первый способ подразумевает явный вызов метода `close()` для потока ввода-вывода. Это традиционный подход, который применялся с первого выпуска Java. При таком подходе метод `close()` обычно вызывается в блоке оператора `finally`. Ниже приведен упрощенный шаблон традиционного способа закрытия потока ввода-вывода. Эта общая методика (или ее разновидность) широко применялась в коде, написанном до появления версии JDK 7.

```
try {  
    // открыть файл и получить доступ к нему  
} catch (исключение_ввода-вывода) {  
    // ...  
} finally {  
    // закрыть файл  
}
```

Второй способ закрытия потока ввода-вывода подразумевает автоматизацию данного процесса с помощью оператора `try` с ресурсами, который был внедрен в версии JDK 7 (и, разумеется, поддерживается в версии JDK 8). Оператор `try` с ресурсами является усовершенствованной формой оператора `try`, имеющей следующий вид:

```
try (спецификация_ресурса) {  
    // использовать ресурс  
}
```

где параметр `спецификация_ресурса` обозначает один или несколько операторов, в которых объявляется и инициализируется ресурс, например, файл или другой ресурс, связанный с потоком ввода-вывода. В указанной `спецификации_ресурса` обычно объявляется переменная, которая инициализируется ссылкой на управляемый объект. По завершении блока оператора `try` ресурс освобождается автоматически. Для файла это означает его автоматическое закрытие. Следовательно, отпадает необходимость вызывать метод `close()` явным образом.

Ниже перечислены три главные особенности применения оператора `try` с ресурсами.

- Ресурсы, управляемые оператором `try` с ресурсами, должны быть объектами классов, реализующих интерфейс `AutoCloseable`.
- Ресурс, объявляемый в блоке оператора `try`, неявно считается завершенным.
- Управлять можно несколькими ресурсами, перечислив их списком через запятую при объявлении.

Не следует также забывать, что область действия объявляемого ресурса ограничивается оператором `try` с ресурсами. Главное преимущество оператора `try` с ресурсами заключается в том, что ресурс (в данном случае поток ввода-вывода) закрывается автоматически по завершении блока оператора `try`. Таким образом, исключается даже возможность забыть закрыть поток ввода-вывода по небрежности. Кроме того, способ закрытия потока ввода-вывода с помощью оператора `try` с ресурсами, как правило, приводит к более краткому и понятному исходному коду, который проще сопровождать.

Благодаря неоспоримым преимуществам оператора `try` с ресурсами можно предположить, что он будет широко применяться в новом коде. Поэтому в большинстве примеров организации ввода-вывода из этой главы в частности и всей книги вообще применяется именно этот оператор. Но поскольку все еще имеется немало унаследованного кода, программисты должны быть знакомы с традиционным подходом к закрытию потока ввода-вывода. Вполне возможно, что вам придется иметь дело с унаследованным кодом, в котором применяется этот традиционный подход, или же в среде, где используется одна из прежних версий Java. Не исключено также, что автоматизированный способ окажется непригодным из-за других особенностей прикладного кода. Поэтому в некоторых примерах организации ввода-вывода в этой книге демонстрируется традиционный способ закрытия потока ввода-вывода, чтобы показать, каким образом он применяется на практике.

И последнее замечание: примеры, в которых применяется оператор `try` с ресурсами, следует компилировать современной версией компилятора Java. Их нельзя скомпилировать прежним компилятором, выпущенным до версии Java SE 7. А примеры, в которых применяется традиционный способ закрытия потока ввода-вывода, могут быть откомпилированы прежними версиями компиляторов Java.

Помните! Оператор `try` с ресурсами упрощает процесс освобождения ресурсов, исключая даже возможность забыть освободить используемый ресурс по небрежности. Поэтому такой способ освобождения ресурсов рекомендуется применять при всякой возможности в процессе разработки нового кода.

Классы потоков ввода-вывода

Система потокового ввода-вывода в Java построена на основе следующих абстрактных классов: `InputStream`, `OutputStream`, `Reader` и `Writer`. Эти классы уже описывались вкратце в главе 13. Они служат для создания ряда конкретных

подклассов потоков ввода-вывода. Несмотря на то что операции ввода-вывода в прикладных программах выполняются с помощью конкретных подклассов, упомянутые выше классы верхнего уровня определяют основные функциональные возможности, которые являются общими для всех классов потоков ввода-вывода.

Классы `InputStream` и `OutputStream` предназначены для организации потоков ввода-вывода байтов, а абстрактные классы `Reader` и `Writer` — для потоков ввода-вывода символов. Классы потоков ввода-вывода байтов и символов образуют отдельные иерархии. В целом классы потоков ввода-вывода символов следует использовать, оперируя символами или их строками, а классы потоков ввода-вывода байтов — оперируя байтами или другими двоичными объектами. Далее в этой главе рассматриваются потоки ввода-вывода как байтов, так и символов.

Потоки ввода-вывода байтов

Классы потоков ввода-вывода байтов предоставляют богатую среду для организации байтового ввода-вывода данных. Поток ввода-вывода байтов можно использовать вместе с объектами любого типа, включая двоичные данные. Такая универсальность делает потоки ввода-вывода байтов важными для многих видов программ. Классы потоков ввода-вывода байтов происходят от классов `InputStream` и `OutputStream`, поэтому именно с них и следует начать обсуждение данной категории потоков ввода-вывода.

Класс `InputStream`

Класс `InputStream` является абстрактным и определяет в Java модель потокавого ввода байтов. Он реализует интерфейсы `AutoCloseable` и `Closeable`. При возникновении ошибок ввода-вывода большинство методов этого класса генерируют исключение типа `IOException`. (К их числу не относятся методы `mark()` и `markSupported()`.) В табл. 20.2 перечислены методы из класса `InputStream`.

Таблица 20.2. Методы из класса `InputStream`

Метод	Описание
<code>int available()</code>	Возвращает количество байтов вводимых данных, доступных в данный момент для чтения
<code>void close()</code>	Закрывает источник ввода данных. При всякой последующей попытке прочитать данные из этого источника будет сгенерировано исключение типа <code>IOException</code>
<code>void mark(int количество_байтов)</code>	Размещает на текущей позиции в потоке ввода метку, которая остается достоверной до тех пор, пока не будет прочитано заданное <i>количество_байтов</i>
<code>boolean markSupported()</code>	Возвращает логическое значение <code>true</code> , если методы <code>mark()</code> и <code>reset()</code> поддерживаются вызывающим потоком ввода

Окончание табл. 20.2

Метод	Описание
<code>int read()</code>	Возвращает целочисленное представление следующего байта, доступного в потоке ввода. По достижении конца файла возвращается значение <code>-1</code>
<code>int read(byte буфер[])</code>	Пытается прочитав в указанный <i>буфер</i> количество байтов, равное <i>буфер.длина</i> , возвращая количество успешно прочитанных байтов. По достижении конца файла возвращается значение <code>-1</code>
<code>int read(byte буфер[], int смещение, int количество_байтов)</code>	Пытается прочитав в указанный <i>буфер</i> заданное <i>количество_байтов</i> , начиная с позиции <i>буфер[смещение]</i> и возвращая количество успешно прочитанных байтов. По достижении конца файла возвращается значение <code>-1</code>
<code>void reset()</code>	Перемещает указатель ввода на установленную ранее метку
<code>long skip(long количество_байтов)</code>	Игнорирует (т.е. пропускает) заданное для ввода <i>количество_байтов</i> , возвращая количество фактически проигнорированных байтов

На заметку! Большинство методов, перечисленных в табл. 20.2, реализуются в классах, производных от класса `InputStream`. Исключением из этого правила служат методы `mark()` и `reset()`. Поэтому обратите внимание на их присутствие или отсутствие в каждом рассматриваемом далее подклассе, производном от класса `InputStream`.

Класс `OutputStream`

Класс `OutputStream` является абстрактным и определяет потоковый вывод байтов. Этот класс реализует интерфейсы `AutoCloseable`, `Closeable` и `Flushable`. Большинство методов из этого класса возвращают значение типа `void` и генерируют исключение типа `IOException` при возникновении ошибок ввода-вывода. Методы из класса `OutputStream` перечислены в табл. 20.3.

Таблица 20.3. Методы из класса `OutputStream`

Метод	Описание
<code>int close()</code>	Закрывает поток вывода. Последующие попытки вывести данные в поток приведут к исключению типа <code>IOException</code>
<code>void flush()</code>	Делает конечным состояние вывода, очищая все буфера, в том числе и буфера вывода
<code>void write(int b)</code>	Записывает единственный байт в поток вывода. Обратите внимание на то, что параметр метода <code>write()</code> относится к типу <code>int</code> , что позволяет вызывать этот метод в выражении, не приводя полученный результат обратно к типу <code>byte</code>

Метод	Описание
<code>void write(byte буфер[])</code>	Записывает весь массив байтов в поток вывода
<code>void write(byte буфер[], int смещение, int количество_байтов)</code>	Записывает часть заданного <i>количества_байтов</i> из указанного массива <i>буфер</i> , начиная с позиции <i>буфер[смещение]</i>

Класс `FileInputStream`

В классе `FileInputStream` создается объект типа `InputStream`, который можно использовать для чтения байтов из файла. Ниже приведены наиболее часто употребляемые конструкторы этого класса.

```
FileInputStream(String путь_к_файлу)
FileInputStream(File объект_файла)
```

Каждый из них может сгенерировать исключение типа `FileNotFoundException`. Здесь параметр *путь_к_файлу* обозначает полное имя пути к файлу, а параметр *объект_файла* — объект типа `File`, описывающий файл. В следующем примере кода создаются два объекта класса `FileInputStream`, использующих один и тот же файл на диске и оба конструктора данного класса.

```
FileInputStream f0 = new FileInputStream("/autoexec.bat")
File f = new File("/autoexec.bat");
FileInputStream f1 = new FileInputStream(f);
```

Хотя первый конструктор, вероятно, используется чаще, второй конструктор позволяет подробно исследовать файл с помощью методов из класса `File`, прежде чем присоединять его к потоку ввода. Когда создается объект типа `FileInputStream`, определяемый им поток ввода открывается для чтения. В классе `FileInputStream` переопределяются шесть методов из абстрактного класса `InputStream`. В то же время методы `mark()` и `reset()` не переопределяются, и поэтому все попытки использовать метод `reset()` вместе с объектом типа `FileInputStream` приводят к генерированию исключения типа `IOException`.

В приведенном ниже примере программы показано, как прочесть один байт, массив байтов и часть массива байтов, а также демонстрируется применение метода `available()` для определения оставшегося количества байтов и метода `skip()` для пропуска нежелательных байтов. Данная программа читает свой исходный файл, который должен присутствовать в текущем каталоге. Обратите внимание на то, что в данном примере используется оператор `try` с ресурсами для автоматического закрытия файла, когда он больше не нужен.

```
// Продемонстрировать применение класса FileInputStream.
// В этой программе используется оператор try с ресурсами.
// Требуется установка комплекта JDK, начиная с версии 7
```

```
import java.io.*;

class FileInputStreamDemo {
    public static void main(String args[]) {
        int size;
```

```

// Для автоматического закрытия потока ввода
// используется оператор try с ресурсами
try ( FileInputStream f =
    new FileInputStream("FileInputStreamDemo.java") ) {

    System.out.println("Общее количество доступных байтов: " +
        (size = f.available()));
    int n = size/40;
    System.out.println("Первые " + n + " байтов, " +
        "прочитанных из файла по очереди методом read()");
    for (int i=0; i < n; i++) {
        System.out.print((char) f.read());
    }

    System.out.println("\nВсе еще доступно: " + f.available());

    System.out.println("Чтение следующих " + n +
        " байтов по очереди методом read(b[])");
    byte b[] = new byte[n];
    if (f.read(b) != n) {
        System.err.println(
            "Нельзя прочитать " + n + " байтов.");
    }

    System.out.println(new String(b, 0, n));
    System.out.println("\nВсе еще доступно: " +
        (size = f.available()));
    System.out.println(
        "Пропустить половину оставшихся байтов методом skip()");
    f.skip(size/2);
    System.out.println("Все еще доступно: " + f.available());

    System.out.println("Чтение " + n/2 +
        " байтов, размещаемых в конце массива");
    if (f.read(b, n/2, n/2) != n/2) {
        System.err.println(
            "Нельзя прочитать " + n/2 + " байтов.");
    }

    System.out.println(new String(b, 0, b.length));
    System.out.println("\nВсе еще доступно: " + f.available());
} catch (IOException e) {
    System.out.println("Ошибка ввода-вывода: " + e);
}
}
}

```

Эта программа выводит следующий результат:

```

Общее количество доступных байтов: 2258
Первые 56 байтов, прочитанных из файла по очереди методом read()
// Продемонстрировать применение класса FileInputStream
Все еще доступно: 2202
Чтение следующих 56 байтов по очереди методом read(b[])
// В этой программе используется оператор try с ресурса

Все еще доступно: 2146
Пропустить половину оставшихся байтов методом skip()
Все еще доступно: 1073
Чтение 28 байтов, размещаемых в конце массива

```

```
// В этой программе использ ввод-вывода: " + e);
Все еще доступно: 1045
```

Этот несколько надуманный пример демонстрирует чтение тремя способами, пропуск вводимых байтов данных и проверку количества байтов, доступных для ввода из потока.

На заметку! В приведенном выше и других примерах из этой главы обрабатываются все исключения ввода-вывода, которые могут произойти, как описано в главе 13. (Подробнее об обработке исключений см. в главе 13.)

Класс `FileOutputStream`

В классе `FileOutputStream` создается объект типа `OutputStream`, который можно использовать для записи байтов в файл. Этот класс реализует интерфейсы `AutoCloseable`, `Closeable` и `Flushable`. Ниже приведены четыре наиболее часто употребляемых конструктора данного класса.

```
FileOutputStream(String путь_к_файлу)
FileOutputStream(File объект_файла)
FileOutputStream(String путь_к_файлу, boolean добавить)
FileOutputStream(File объект_файла, boolean добавить)
```

Все эти конструкторы могут сгенерировать исключение типа `FileNotFoundException`. Здесь параметр `путь_к_файлу` обозначает имя полного пути к файлу, а параметр `объект_файла` — объект типа `File`, описывающий файл. Если параметр `добавить` принимает логическое значение `true`, файл открывается в режиме добавления данных.

Создание объекта типа `FileOutputStream` не зависит от того, существует ли указанный файл. Он создается перед своим открытием при построении объекта класса `FileOutputStream`. При попытке открыть файл, доступный только для чтения, будет сгенерировано соответствующее исключение.

В приведенном ниже примере программы организуется буфер байтов. Сначала в ней создается объект типа `String`, а затем вызывается метод `getBytes()` для извлечения его эквивалента в виде массива байтов. Далее создаются три файла. Первый файл `file1.txt` будет содержать каждый второй байт образца текста, второй файл `file2.txt` — все байты из данного образца, а третий файл `file3.txt` — только последнюю четверть данного образца.

```
// Продемонстрировать применение класса FileOutputStream.
// В этой программе используется традиционный способ закрытия файла
```

```
import java.io.*;

class FileOutputStreamDemo {
    public static void main(String args[]) {
        String source = "Now is the time for all good men\n" +
                        "    to come to the aid of their country\n" +
                        "    and pay their due taxes.";
        byte buf[] = source.getBytes();
        FileOutputStream f0 = null;
        FileOutputStream f1 = null;
        FileOutputStream f2 = null;
```

```
try {
    f0 = new FileOutputStream("file1.txt");
    f1 = new FileOutputStream("file2.txt");
    f2 = new FileOutputStream("file3.txt");

    // записать данные в первый файл
    for (int i=0; i < buf.length; i += 2) f0.write(buf[i]);

    // записать данные во второй файл
    f1.write(buf);

    // записать данные в третий файл
    f2.write(buf, buf.length-buf.length/4, buf.length/4);
} catch(IOException e) {
    System.out.println("Произошла ошибка ввода-вывода");
} finally {
    try {
        if(f0 != null) f0.close();
    } catch(IOException e) {
        System.out.println("Ошибка закрытия файла file1.txt");
    }
    try {
        if(f1 != null) f1.close();
    } catch(IOException e) {
        System.out.println("Ошибка закрытия файла file2.txt");
    }
    try {
        if(f2 != null) f2.close();
    } catch(IOException e) {
        System.out.println("Ошибка закрытия файла file3.txt");
    }
}
}
```

Ниже показано, как будет выглядеть содержимое каждого упомянутого выше файла после выполнения данной программы. Сначала представлено содержимое файла file1.txt:

```
Nwi h iefralgo e
t oet h i ftercuty n a hi u ae.
```

Затем содержимое файла file2.txt:

```
Now is the time for all good men
to come to the aid of their country
and pay their due taxes.
```

И наконец, содержимое файла file3.txt:

```
nd pay their due taxes.
```

Как следует из комментариев к исходному коду данной программы, в ней демонстрируется применение традиционного способа закрытия файла, когда он больше не нужен. Такой способ требуется во всех версиях Java до JDK 7 и широко применяется в унаследованном коде. В данном примере можно обнаружить немало довольно неуклюжего кода, требующегося для явного вызова метода `close()`, поскольку при каждом его вызове может быть сгенерировано исключение типа `IOException`, если операция закрытия завершится неудачно. Программа из данного примера может быть значительно улучшена, если воспользоваться вторым

способом закрытия файла с помощью оператора `try` с ресурсами. Для сравнения ниже приведена переделанная версия этой программы. Обратите внимание на то, что ее исходный код намного короче и понятнее.

```
// Продемонстрировать применение класса FileOutputStream.
// Для закрытия файла в этой программе применяется оператор
// try с ресурсами. Требуется установка комплекта JDK,
// начиная с версии 7

import java.io.*;

class FileOutputStreamDemo {
    public static void main(String args[]) {
        String source = "Now is the time for all good men\n" +
            "    to come to the aid of their country\n" +
            "    and pay their due taxes.";
        byte buf[] = source.getBytes();

        // применить оператор try с ресурсами для закрытия файлов
        try (FileOutputStream f0 = new FileOutputStream("file1.txt");
            FileOutputStream f1 = new FileOutputStream("file2.txt");
            FileOutputStream f2 = new FileOutputStream("file3.txt") )
        {

            // записать данные в первый файл
            for (int i=0; i < buf.length; i += 2) f0.write(buf[i]);

            // записать данные во второй файл
            f1.write(buf);

            // записать данные в третий файл
            f2.write(buf, buf.length-buf.length/4, buf.length/4);
        } catch (IOException e) {
            System.out.println("Произошла ошибка ввода-вывода");
        }
    }
}
```

Класс `ByteArrayInputStream`

Класс `ByteArrayInputStream` реализует поток ввода, использующий массив байтов в качестве источника данных. У этого класса имеются два конструктора, каждому из которых требуется массив байтов в качестве источника данных:

```
ByteArrayInputStream(byte массив[])
ByteArrayInputStream(byte массив[], int начало, int количество_байтов)
```

где параметр *массив* обозначает источник данных. Второй конструктор данного класса создает объект типа `InputStream` из подмножества массива байтов, начиная с символа на позиции, определяемой параметром *начало*, и длиной, задаваемой параметром *количество_байтов*.

Метод `close()` не оказывает никакого влияния на объект класса `ByteArrayInputStream`, поэтому вызывать его для данного объекта класса не нужно, хотя это и не считается ошибкой.

В следующем примере программы создается пара объектов класса `ByteArrayInputStream`, которая инициализируется байтами, представляющими английский алфавит:

```
// Продемонстрировать применение класса ByteArrayInputStream
import java.io.*;

class ByteArrayInputStreamDemo {
    public static void main(String args[]) throws IOException {
        String tmp = "abcdefghijklmnopqrstuvwxyz";
        byte b[] = tmp.getBytes();

        ByteArrayInputStream input1 = new ByteArrayInputStream(b);
        ByteArrayInputStream input2 = new ByteArrayInputStream(b, 0, 3);
    }
}
```

Объект `input1` содержит полный алфавит в нижнем регистре, в то время как объект `input2` — только первые три буквы. Класс `ByteArrayInputStream` реализует методы `mark()` и `reset()`. Но если метод `mark()` не вызывается, то метод `reset()` устанавливает указатель на начало потока ввода (в данном случае — в начало массива байтов, передаваемого конструктору данного класса). В приведенном ниже примере показано, как пользоваться методом `reset()` для чтения одних и тех же вводимых данных дважды. В данном примере программа читает и выводит буквы "abc" сначала в нижнем регистре, а затем в верхнем.

```
import java.io.*;

class ByteArrayInputStreamReset {
    public static void main(String args[]) {
        String tmp = "abc";
        byte b[] = tmp.getBytes();
        ByteArrayInputStream in = new ByteArrayInputStream(b);

        for (int i=0; i<2; i++) {
            int c;
            while ((c = in.read()) != -1) {
                if (i == 0) {
                    System.out.print((char) c);
                } else {
                    System.out.print(Character.toUpperCase((char) c));
                }
            }
            System.out.println();
            in.reset();
        }
    }
}
```

В данном примере каждый символ сначала читается из потока ввода, а затем выводится без изменений в нижнем регистре. Далее поток ввода устанавливается в исходное состояние, и чтение данных начинается снова, но на этот раз каждый символ преобразуется в верхний регистр перед выводом. В итоге выводится следующий результат:

```
abc
ABC
```

Класс `ByteArrayOutputStream`

Класс `ByteArrayOutputStream` реализует поток вывода, использующий массив байтов в качестве адресата. В классе `ByteArrayOutputStream` имеются два конструктора, как показано ниже.

```
ByteArrayOutputStream()  
ByteArrayOutputStream(int количество_байтов)
```

В первой форме конструктора создается буфер размером 32 байта, а во второй форме — буфер, размер которого составляет заданное *количество_байтов*. Этот буфер хранится в защищенном поле `buf` класса `ByteArrayOutputStream`. Размер буфера увеличивается автоматически по мере необходимости. Количество байтов, содержащихся в буфере, хранится в защищенном поле `count` класса `ByteArrayOutputStream`.

Метод `close()` не оказывает никакого влияния на класс `ByteArrayOutputStream`. Поэтому вызывать его для объекта класса `ByteArrayOutputStream` не нужно, хотя это и не считается ошибкой.

В следующем примере программы демонстрируется применение класса `ByteArrayOutputStream`:

```
// Продемонстрировать применение класса ByteArrayOutputStream  
// В этой программе применяется оператор try с ресурсами.  
// Требуется установка комплекта JDK, начиная с версии 7  
  
import java.io.*;  
  
class ByteArrayOutputStreamDemo {  
    public static void main(String args[]) {  
        ByteArrayOutputStream f = new ByteArrayOutputStream();  
        String s = "Эти данные должны быть выведены в массив";  
        byte buf[] = s.getBytes();  
  
        try {  
            f.write(buf);  
        } catch(IOException e) {  
            System.out.println("Ошибка записи в буфер");  
            return;  
        }  
  
        System.out.println("Буфер в виде символьной строки");  
        System.out.println(f.toString());  
        System.out.println("В массив");  
        byte b[] = f.toByteArray();  
        for (int i=0; i<b.length; i++) System.out.print((char) b[i]);  
  
        System.out.println("\nБ поток вывода типа OutputStream()");  
  
        // использовать оператор try с ресурсами для управления  
        // потоком ввода-вывода в файл  
        try ( FileOutputStream f2 = new FileOutputStream("test.txt"))  
        {  
            f.writeTo(f2);  
        } catch(IOException e) {  
            System.out.println("Ошибка ввода-вывода: " + e);  
            return;  
        }  
    }  
}
```

```
System.out.println("Установка в исходное состояние");
f.reset();

for (int i=0; i<3; i++) f.write('X');

System.out.println(f.toString());
}
}
```

В результате выполнения этой программы выводится приведенный ниже результат. Обратите внимание на то, что после вызова метода `reset()` сначала выводятся три буквы 'X'.

```
Буфер в виде символьной строки
Эти данные должны быть выведены в массив
В массив
Эти данные должны быть выведены в массив
В поток вывода типа OutputStream()
Установка в исходное состояние
XXX
```

В данном примере для записи содержимого потока вывода `f` в файл `test.txt` вызывается служебный метод `writeTo()`. Просмотр файла `test.txt`, созданного в предыдущем примере, обнаруживает такой результат, как и следовало ожидать.

Эти данные должны быть выведены в массив

Фильтруемые потоки ввода-вывода байтов

Фильтруемые потоки ввода-вывода представляют собой оболочки, в которые заключаются базовые потоки ввода-вывода для расширения их функциональных возможностей. Такие потоки ввода-вывода обычно доступны методам, которые ожидают обобщенный поток, служащий суперклассом для фильтруемого потока ввода-вывода. Типичными расширениями функций этих потоков ввода-вывода являются буферизация, преобразование символов и исходных данных. Фильтруемые потоки ввода-вывода байтов реализуются в классах `FilterInputStream` и `FilterOutputStream`. Их конструкторы приведены ниже.

```
FilterOutputStream(OutputStream os)
FilterInputStream(InputStream is)
```

Методы, представленные в этих классах, аналогичны методам из классов `InputStream` и `OutputStream`.

Буферизованные потоки ввода-вывода байтов

Буферизованные потоки ввода-вывода расширяют класс фильтруемого потока ввода-вывода, присоединяя к нему буфер, выделяемый в памяти. Этот буфер позволяет выполнять операции одновременного ввода-вывода не одного, а не нескольких байтов, повышая тем самым производительность. Благодаря наличию буфера можно выполнять пропуск байтов, маркировку и установку потока ввода-вывода в исходное состояние. Буферизованные потоки ввода-вывода байтов реализуют-

ся классами `BufferedInputStream` и `BufferedOutputStream`, а также классом `PushbackInputStream`.

Класс `BufferedInputStream`

Буферизация ввода-вывода — очень распространенный способ оптимизации производительности. Класс `BufferedInputStream` позволяет заключить в оболочку любой поток ввода типа `InputStream` и добиться повышения производительности. У класса `BufferedInputStream` имеются два конструктора:

```
BufferedInputStream(InputStream поток_ввода)  
BufferedInputStream(InputStream поток_ввода, int размер_буфера)
```

В первой форме конструктора создается буферизованный поток ввода, использующий размер буфера по умолчанию. Во второй форме конструктора задается конкретный *размер_буфера*. Рекомендуется указывать размер буфера, кратный размеру страницы памяти, дисковому блоку и т.п., и это окажет существенное положительное влияние на производительность. Но, с другой стороны, это зависит от конкретной реализации. Оптимальный размер буфера обычно зависит от операционной системы, объема доступной памяти и конфигурации главной машины. Чтобы добиться эффективной буферизации, совсем не обязательно углубляться во все эти сложности. Поэтому целесообразно установить для потока ввода буфер размером 8192 байта или меньше. Таким образом, низкоуровневая система сможет читать блоки данных с диска или из сети и сохранять результат в установленном буфере. И даже если данные читаются байтами из потока ввода типа `InputStream`, то на практике придется, как правило, оперировать быстродействующей памятью.

Буферизация потока ввода служит также основанием для поддержки операции перемещения назад в потоке, имеющем свой буфер. Помимо методов `read()` и `skip()`, реализуемых в любом из классов `InputStream` и `BufferedInputStream`, поддерживаются также методы `mark()` и `reset()`. Наличие такой поддержки отражает возврат логического значения `true` из метода `BufferedInputStream.markSupported()`.

В приведенном ниже примере моделируется ситуация, когда с помощью метода `mark()` можно запомнить место в потоке ввода, чтобы в дальнейшем вернуться в это место с помощью метода `reset()`. В данном примере производится синтаксический анализ содержимого потока ввода с целью обнаружить в нем ссылку на элемент HTML-разметки знака авторского права. Такая ссылка начинается со знака амперсанда (&) и оканчивается точкой с запятой (;) без всяких промежуточных пробелов. Образец введенных данных содержит два амперсанда, чтобы наглядно показать, когда происходит установка в исходное состояние с помощью метода `reset()` и когда этого не происходит.

```
// Использовать буферизованный ввод.  
// В этой программе применяется оператор try с ресурсами.  
// Требуется установка комплекта JDK, начиная с версии 7
```

```
import java.io.*;  
  
class BufferedInputStreamDemo {  
    public static void main(String args[]) {  
        String s = "Это знак авторского права &copy; " +
```

```

        ", a &copy; – нет.\n";
byte buf[] = s.getBytes();

ByteArrayInputStream in = new ByteArrayInputStream(buf);
int c;
boolean marked = false;

// использовать оператор try с ресурсами для управления файлами
try (BufferedInputStream f = new BufferedInputStream(in))
{
    while ((c = f.read()) != -1) {
        switch(c) {
            case '&':
                if (!marked) {
                    f.mark(32);
                    marked = true;
                } else {
                    marked = false;
                }
                break;
            case ';':
                if (marked) {
                    marked = false;
                    System.out.print("(c)");
                } else
                    System.out.print((char) c);
                break;
            case ' ':
                if (marked) {
                    marked = false;
                    f.reset();
                    System.out.print("&");
                } else
                    System.out.print((char) c);
                break;
            default:
                if (!marked)
                    System.out.print((char) c);
                break;
        }
    }
} catch (IOException e) {
    System.out.println("Ошибка ввода-вывода: " + e);
}
}
}

```

Обратите внимание на то, что в данном примере делается вызов метода `mark(32)`, в результате которого сохраняется метка для чтения следующих 32 байтов, чего достаточно для любых ссылок на элементы HTML-разметки. Эта программа выводит следующий результат:

Это знак авторского права ©, а © – нет.

Класс `BufferedOutputStream`

Класс `BufferedOutputStream` подобен любому классу `OutputStream`, за исключением дополнительного метода `flush()`, обеспечивающего запись данных в буферизуемый поток вывода. Класс `BufferedOutputStream` предназначен

для повышения производительности за счет сокращения количества операций записи данных, фактически выполняемых системой, и поэтому может возникнуть потребность вызвать метод `flush()`, чтобы инициировать немедленную запись всех данных из буфера.

В отличие от буферизованного ввода, буферизованный вывод не обеспечивает дополнительные функциональные возможности. Буфера вывода в Java требуются для повышения производительности. Ниже приведены два конструктора класса `BufferedOutputStream`.

```
BufferedOutputStream(OutputStream поток_вывода)
BufferedOutputStream(OutputStream поток_вывода, int размер_буфера)
```

В первой форме конструктора создается буферизованный поток вывода с использованием буфера, размер которого выбирается по умолчанию. Во второй форме конструктора задается конкретный *размер_буфера*.

Класс `PushbackInputStream`

Одним из новшеств в буферизации ввода-вывода является реализация механизма *возврата в поток ввода*. Этот механизм используется в потоке ввода, чтобы обеспечить чтение байта с последующим его возвратом в поток. Принцип действия возврата в поток ввода реализован в классе `PushbackInputStream`. Благодаря этому механизму можно бегло просмотреть поток ввода и выяснить, что именно поступит из него в следующий раз, фактически не извлекая данные.

В классе `PushbackInputStream` имеются следующие конструкторы:

```
PushbackInputStream(InputStream поток_ввода)
PushbackInputStream(InputStream поток_ввода, int количество_байтов)
```

В первой форме конструктора создается объект потока ввода, позволяющий возвратить один байт в указанный *поток_ввода*. А во второй форме создается поток ввода с буфером возврата указанной длины. Это позволяет возвратить в поток ввода не один, а заданное *количество_байтов*.

Помимо уже известных методов из класса `InputStream`, в классе `PushbackInputStream` предоставляется метод `unread()`. Ниже приведены общие формы его объявления.

```
void unread(int b)
void unread(byte буфер[])
void unread(byte буфер, int смещение, int количество_байтов)
```

В первой форме данного метода в поток ввода возвращается заданный младший байт *b*. Это будет следующий байт, возвращаемый при последующем вызове метода `read()`. Во второй форме в поток ввода возвращаются байты из указанного буфера. А в третьей форме в поток ввода возвращается заданное *количество_байтов* из указанного буфера, начиная с позиции *смещение*. Исключение типа `IOException` генерируется при попытке возвратить байт в поток ввода, когда буфер возврата заполнен.

Ниже приведен пример программы, демонстрирующий применение класса `PushbackInputStream` и его метода `unread()` в синтаксическом анализаторе языка программирования для различения операций сравнения (`==`) и присваивания (`=`).

```
// Продемонстрировать применение метода unread()
// из класса PushbackInputStream.
// В этой программе применяется оператор try с ресурсами.
// Требуется установка комплекта JDK, начиная с версии 7

import java.io.*;

class PushbackInputStreamDemo {
    public static void main(String args[]) {
        String s = "if (a == 4) a = 0;\n";
        byte buf[] = s.getBytes();
        ByteArrayInputStream in = new ByteArrayInputStream(buf);
        int c;

        try ( PushbackInputStream f = new PushbackInputStream(in) )
        {
            while ((c = f.read()) != -1) {
                switch(c) {
                    case '=':
                        if ((c = f.read()) == '=')
                            System.out.print(".eq.");
                        else {
                            System.out.print("<-");
                            f.unread(c);
                        }
                        break;
                    default:
                        System.out.print((char) c);
                        break;
                }
            }
        } catch (IOException e) {
            System.out.println("Ошибка ввода-вывода: " + e);
        }
    }
}
```

Ниже приведен результат, выводимый данной программой. Обратите внимание на то, что операция "==" заменена на ".eq.", а операция "=" — на "<-".

```
if (a .eq. 4) a<- 0;
```

Внимание! Класс `PushbackInputStream` обладает побочным эффектом, делая невозможным вызов методов `mark()` и `reset()` из класса `InputStream`, применяемого для его создания. Поэтому следует вызвать метод `markSupported()`, чтобы проверить любой поток ввода, в котором предполагается использовать методы `mark()` и `reset()`, на предмет поддержки этих методов.

Класс `SequenceInputStream`

Класс `SequenceInputStream` позволяет соединить вместе несколько потоков ввода типа `InputStream`. Объект класса `SequenceInputStream` строится иначе, чем объект класса `InputStream`. Конструктор класса `SequenceInputStream` принимает в качестве аргумента пару объектов класса `InputStream` или перечисление типа `Enumeration` объектов класса `InputStream`, как следует из приведенных ниже общих форм объявления конструктора этого класса.

```
SequenceInputStream(
    InputStream первый_поток, InputStream второй_поток)
SequenceInputStream(
    Enumeration <? extends InputStream> перечисление_потоков)
```

В процессе своей работы этот класс делает запросы на чтение из первого потока ввода типа `InputStream` до его исчерпания, а затем переходит к другому потоку ввода. Если же указано перечисление типа `Enumeration` потоков, этот класс обращается ко всем потокам ввода типа `InputStream` по очереди до исчерпания самого последнего из них. По достижении конца каждого файла связанный с ним поток ввода закрывается. Закрытие потока ввода, созданного средствами класса `SequenceInputStream`, приводит к закрытию всех остальных открытых потоков.

Ниже приведен пример применения класса `SequenceInputStream` для вывода содержимого двух файлов. Ради наглядности в программе из этого примера применяется традиционный способ закрытия файла. В качестве упражнения попробуйте применить современный способ автоматического закрытия файла с помощью оператора `try` с ресурсами, внося соответствующие изменения в исходный код данной программы.

```
// Продемонстрировать организацию последовательного ввода.
// В этой программе используется оператор try с ресурсами.
// Требуется установка комплекта JDK, начиная с версии 7

import java.io.*;
import java.util.*;

class InputStreamEnumerator implements Enumeration<FileInputStream> {
    private Enumeration<String> files;

    public InputStreamEnumerator(Vector<String> files) {
        this.files = files.elements();
    }

    public boolean hasMoreElements() {
        return files.hasMoreElements();
    }

    public FileInputStream nextElement() {
        try {
            return new FileInputStream(files.nextElement().toString());
        } catch (IOException e) {
            return null;
        }
    }
}

class SequenceInputStreamDemo {
    public static void main(String args[]) {
        int c;
        Vector<String> files = new Vector<String>();

        files.addElement("file1.txt");
        files.addElement("file2.txt");
        files.addElement("file3.txt");
        InputStreamEnumerator ise = new InputStreamEnumerator(files);
        InputStream input = new SequenceInputStream(ise);
```

```

try {
    while ((c = input.read()) != -1)
        System.out.print((char) c);
} catch (NullPointerException e) {
    System.out.println("Ошибка открытия файла.");
} catch (IOException e) {
    System.out.println("Ошибка ввода-вывода: " + e);
} finally {
    try {
        input.close();
    } catch (IOException e) {
        System.out.println(
            "Ошибка закрытия потока ввода SequenceInputStream");
    }
}
}
}

```

В данном примере сначала создается объект типа `Vector`, а затем в него вводятся имена трех файлов. Далее этот вектор с именами файлов передается классу `InputStreamEnumerator`, который служит оболочкой для вектора, где его элементы возвращаются не в виде имен файлов, а в виде потоков ввода типа `FileInputStream`, открытых по этим именам. Поток ввода типа `SequenceInputStream` открывает каждый файл по очереди, чтобы затем вывести полученное содержимое этих файлов.

Обратите внимание на то, что если файл нельзя открыть, то в метод `nextElement()` возвращает пустое значение `null`. Это приводит к генерированию исключения типа `NullPointerException`, перехватываемого в методе `main()`.

Класс `PrintStream`

Класс `PrintStream` предоставляет все возможности для вывода данных по дескриптору файла `System.out` типа `System`, которые используются в примерах программ с самого начала данной книги. Благодаря этому класс `PrintStream` является одним из наиболее употребительных в Java. Он реализует интерфейсы `Appendable`, `AutoCloseable`, `Closeable` и `Flushable`.

В классе `PrintStream` определяется ряд конструкторов. Рассмотрим сначала следующие формы конструкторов этого класса:

```

PrintStream(OutputStream поток_вывода)
PrintStream(OutputStream поток_вывода, boolean автоочистка)
PrintStream(OutputStream поток_вывода, boolean автоочистка,
            String набор_символов) throws UnsupportedOperationException

```

где параметр `поток_вывода` обозначает открытый поток вывода типа `OutputStream`, который будет принимать выводимые данные. Параметр `автоочистка` определяет, будет ли буфер вывода автоматически очищаться при каждой записи последовательности символов новой строки (`\n`), записи массива байтов или вызове метода `println()`. Если параметр `автоочистка` принимает логическое значение `true`, то происходит автоматическая очистка буфера вывода. А если этот параметр принимает логическое значение `false`, то очистка буфера вывода не производится автоматически. Первая форма конструктора данного класса не предполагает автоматическую очистку буфера вывода. Кроме того, можно задать конкретную кодировку символов, передав ее имя в качестве параметра `набор_символов`.

Следующие формы конструкторов предоставляют простые способы создания объекта класса `PrintStream` для вывода данных в файл:

```
PrintStream(Файл_вывода) throws FileNotFoundException
PrintStream(Файл_вывода, String набор_символов)
    throws FileNotFoundException, UnsupportedEncodingException
PrintStream(String имя_файла_вывода) throws FileNotFoundException
PrintStream(String имя_файла_вывода, String набор_символов)
    throws FileNotFoundException, UnsupportedEncodingException
```

Они позволяют создавать объект класса `PrintStream` из объекта типа `File` или по указанному имени файла. Но в любом случае файл создается автоматически. Любой существующий файл с тем же именем уничтожается. Как только поток вывода будет создан в виде объекта класса `PrintStream`, он будет направлять все выводимые данные в указанный файл. Конкретную кодировку символов можно задать в качестве параметра `набор_символов`.

На заметку! При наличии диспетчера защиты некоторые конструкторы класса `PrintStream` будут генерировать исключение типа `SecurityException`, если произойдет нарушение защиты.

В классе `PrintStream` поддерживаются методы `print()` и `println()` для вывода всех типов данных, включая и тип `Object`. Если аргумент не относится к примитивному типу, то в методах из класса `PrintStream` сначала вызывается метод `toString()` для выводимого объекта, а затем выводится результат, получаемый из этого метода.

Относительно давно (после выпуска версии JDK 5) класс `PrintStream` был дополнен методом `printf()`. Этот метод позволяет задать точный формат вывода данных. Для форматирования данных в методе `printf()` используются соответствующие средства класса `Formatter`, описанного в главе 19, после чего отформатированные данные направляются в вызывающий поток вывода. Форматирование можно, конечно, выполнить и вручную, обращаясь непосредственно к классу `Formatter`, но метод `printf()` значительно упрощает данный процесс. Он является аналогом функции `printf()` в C/C++, упрощая перенос существующего кода из C/C++ на Java. Откровенно говоря, метод `printf()` стал очень полезным дополнением прикладного программного интерфейса Java API, поскольку он значительно упрощает вывод отформатированных данных на консоль.

Метод `printf()` имеет следующие общие формы:

```
PrintStream printf(String форматизирующая_строка, Object ... аргументы)
PrintStream printf(Locale региональные_настройки,
    String форматизирующая_строка, Object ... аргументы)
```

В первой форме данного метода заданные аргументы выводятся в стандартный поток вывода в формате, указанном в качестве параметра `форматизирующая_строка`, с учетом региональных настроек по умолчанию. А во второй форме можно указать конкретные `региональные_настройки`. Но в любом случае возвращается вызывающий поток вывода в виде объекта типа `PrintStream`.

В общем, метод `printf()` подобен методу `format()`, определенному в классе `Formatter`. Параметр `форматизирующая_строка` состоит из элементов двух типов. Элемент первого типа состоит из символов, которые просто копируются в буфер

вывода, а элемент второго типа — из спецификаторов формата, определяющих порядок вывода указанных далее *аргументов*. Подробнее о форматировании выводимых данных, включая описание спецификаторов формата, см. в описании класса `Formatter` в главе 19.

Стандартный поток вывода `System.out` относится к типу `PrintStream`, и поэтому для него можно вызывать метод `printf()`. Следовательно, метод `printf()` можно вызвать вместо метода `println()`, когда требуется вывести на консоль отформатированные данные. Так, в приведенном ниже примере программы метод `printf()` вызывается для вывода числовых значений в разных форматах. До версии JDK 5 для такого форматирования данных приходилось немало потрудиться. Но с внедрением метода `printf()` задача вывода отформатированных данных значительно упростилась.

```
// Продемонстрировать применение метода printf()

class PrintfDemo {
    public static void main(String args[]) {
        System.out.println("Ниже приведены некоторые " +
                           "числовые значения в разных форматах.\n");

        System.out.printf("Разные целочисленные форматы: ");
        System.out.printf("%d %(d %d %05d\n", 3, -3, 3, 3);

        System.out.println();
        System.out.printf("Формат чисел с плавающей точкой " +
                           "по умолчанию: %f\n", 1234567.123);
        System.out.printf("Формат чисел с плавающей точкой, " +
                           "разделяемых запятыми: %,f\n", 1234567.123);
        System.out.printf("Формат отрицательных чисел с " +
                           "плавающей точкой по умолчанию: %,f\n", -1234567.123);
        System.out.printf("Другой формат отрицательных чисел с " +
                           "плавающей точкой: %, (f\n", -1234567.123);

        System.out.println();

        System.out.printf("Выравнивание положительных и " +
                           "отрицательных числовых значений:\n");
        System.out.printf("% ,.2f\n% ,.2f\n",
                           1, 234, 567.12, -1, 234, 567.12);
    }
}
```

Эта программа выводит следующий результат:

Ниже приведены некоторые числовые значения в разных форматах.

Разные целочисленные форматы: 3 (3) +3 00003

формат чисел с плавающей точкой по умолчанию: 1234567.123000

формат чисел с плавающей точкой, разделяемых запятыми: 1,234,567.123000

формат отрицательных чисел с плавающей точкой по умолчанию: -1,234,567.123000

Другой формат отрицательных чисел с плавающей точкой: (1,234,567.123000)

Выравнивание положительных и отрицательных числовых значений:

1,234,567.12

-1,234,567.12

В классе `PrintStream` определяется также метод `format()`. Ниже приведены его общие формы. Он действует таким же образом, как и метод `printf()`.


```
PrintStream format(String форматирующая_строка, Object ... аргументы)
PrintStream format(Locale регион, String форматирующая_строка,
                   Object ... аргументы)
```

Классы DataOutputStream и DataInputStream

Эти классы позволяют выводить примитивные данные в поток или вводить их из потока. Они реализуют интерфейсы DataOutput и DataInput соответственно. В этих интерфейсах определяются методы, выполняющие взаимное преобразование значений примитивных типов и последовательностей байтов. Такие потоки ввода-вывода упрощают сохранение в файле данных, представленных в двоичной форме, например, целочисленных значений или числовых значений с плавающей точкой. Ниже будет показано, каким образом организуется ввод-вывод примитивных данных в двоичной форме.

Класс DataOutputStream расширяет класс FilterOutputStream, который, в свою очередь, расширяет класс OutputStream. Помимо интерфейса DataOutput, класс DataOutputStream реализует интерфейсы AutoCloseable, Closeable и Flushable. В классе DataOutputStream определяется следующий конструктор:

```
DataOutputStream(OutputStream поток_вывода)
```

где параметр *поток_вывода* обозначает поток, в который будут выводиться данные. Когда поток вывода типа DataOutputStream закрывается в результате вызова метода close(), базовый поток, определяемый параметром *поток_вывода*, также закрывается автоматически.

В классе DataOutputStream поддерживаются все методы, определенные в его суперклассах. Но по-настоящему привлекательным этот класс делают реализуемые в нем методы из интерфейса DataOutput. В интерфейсе DataOutput определяются методы, сначала преобразующие значения примитивных типов в последовательности байтов, а затем выводящие их в базовый поток. Ниже приведены общие формы этих методов, где параметр *значение* обозначает выводимое в поток значение примитивного типа.

```
final void writeDouble(double значение) throws IOException
final void writeBoolean(boolean значение) throws IOException
final void writeInt(int значение) throws IOException
```

Класс DataInputStream служит дополнением класса DataOutputStream для ввода примитивных типов данных в двоичной форме. Он расширяет класс FilterInputStream, который, в свою очередь, расширяет класс InputStream. Помимо интерфейса DataInput, класс DataInputStream реализует интерфейсы AutoCloseable и Closeable. В этом классе определяется следующий единственный конструктор:

```
DataStream(InputStream поток_ввода)
```

где параметр *поток_ввода* обозначает тот поток, из которого будут вводиться данные. Когда поток ввода типа DataInputStream закрывается в результате вызова метода close(), базовый поток ввода, определяемый параметром *поток_ввода*, также закрывается автоматически.

Как и в классе `DataOutputStream`, в классе `DataInputStream` поддерживаются все методы из его суперклассов, а также методы, определенные в интерфейсе `DataInput`, что делает его особенно привлекательным. Эти методы вводят последовательность байтов и преобразуют их в значения примитивных типов. Ниже приведены общие формы этих методов.

```
final double readDouble() throws IOException
final boolean readBoolean() throws IOException
final int readInt() throws IOException
```

В следующем примере программы демонстрируется применение классов `DataOutputStream` и `DataInputStream`:

```
// Продемонстрировать применение классов
// DataInputStream и DataOutputStream.
// В этой программе используется оператор try с ресурсами.
// Требуется установка комплекта JDK, начиная с версии 7

import java.io.*;

class DataIODemo {
    public static void main(String args[])

        // сначала вывести данные в файл
        try ( DataOutputStream dout =
            new DataOutputStream(new FileOutputStream("Test.dat")) )
        {
            dout.writeDouble(98.6);
            dout.writeInt(1000);
            dout.writeBoolean(true);
        } catch(FileNotFoundException e) {
            System.out.println("Нельзя открыть файл вывода");
            return;
        } catch(IOException e) {
            System.out.println("Ошибка ввода-вывода: " + e);
        }

        // а теперь ввести данные из файла
        try ( DataInputStream din =
            new DataInputStream(new FileInputStream("Test.dat")) )
        {
            double d = din.readDouble();
            int i = din.readInt();
            boolean b = din.readBoolean();

            System.out.println("Получаемые значения: " +
                d + " " + i + " " + b);

        } catch(FileNotFoundException e) {
            System.out.println("Нельзя открыть файл ввода ");
            return;
        } catch(IOException e) {
            System.out.println("Ошибка ввода-вывода: " + e);
        }
    }
}
```

Ниже приведен результат, выводимый данной программой.

Получаемые значения: 98.6 1000 true

Класс `RandomAccessFile`

Класс `RandomAccessFile` инкапсулирует файл произвольного доступа. Он не наследуется от класса `InputStream` или `OutputStream`. Вместо этого он реализует интерфейсы `DataInput` и `DataOutput`, в которых определяются основные методы ввода-вывода. Этот класс реализует также интерфейсы `AutoCloseable` и `Closeable`. Класс `RandomAccessFile` отличается поддержкой запросов на позиционирование, что позволяет установить указатель файла на любой позиции в пределах этого файла. У этого класса имеются следующие конструкторы:

```
RandomAccessFile(File объект_файла, String доступ)
    throws FileNotFoundException
RandomAccessFile(String имя_файла, String доступ)
    throws FileNotFoundException
```

В первой форме конструктора параметр *объект_файла* обозначает открываемый файл как объект типа `File`. А во второй форме имя конкретного файла передается в качестве параметра *имя_файла*. Но в любом случае параметр *доступ* определяет тип разрешенного доступа. Так, если параметр *доступ* принимает значение "r", данные можно прочитать из файла, но не записать в него. Если же параметр *доступ* принимает значение "rw", файл открывается в режиме чтения и записи. А если параметр *доступ* принимает значение "rws", то файл открывается для выполнения операций чтения и записи, и каждое изменение данных или метаданных в файле немедленно выводится на физическое устройство. Метод `seek()`, общая форма которого приведена ниже, служит для установки указателя файла на текущей позиции в этом файле.

```
void seek(long новая_позиция) throws IOException
```

Здесь параметр *новая_позиция* обозначает новую позицию указателя файла, отсчитываемую в байтах от начала файла. После вызова метода `seek()` следующая операция чтения или записи данных выполняется именно с этой новой позиции в файле.

Класс `RandomAccessFile` реализует стандартные методы ввода-вывода, которые можно использовать для чтения и записи файлов с произвольным доступом. Кроме того, он содержит ряд дополнительных методов, к числу которых относится метод `setLength()`. Ниже приведена его общая форма.

```
void setLength(long длина) throws IOException
```

Метод `setLength()` устанавливает заданную длину вызывающего файла. С помощью этого метода можно удлинять или укорачивать файл. Если файл удлиняется, добавляемая в него порция не определена.

Потоки ввода-вывода символов

Классы потоков ввода-вывода байтов предоставляют необходимые функциональные возможности для выполнения операций ввода-вывода любого типа, но они не в состоянии оперировать непосредственно символами в Юникоде. А поскольку одной из главных целей Java является соблюдение принципа "написано

однажды, выполняется везде”, то в Java пришлось внедрить поддержку непосредственного ввода-вывода символов. В этом разделе рассматривается ряд классов, предназначенных для ввода-вывода символов. Как пояснялось ранее, на вершине иерархии классов, реализующих потоки ввода-вывода символов, находятся абстрактные классы `Reader` и `Writer`. Именно с них и следует начать рассмотрение классов данной категории.

Класс Reader

Класс `Reader` является абстрактным и определяет потоковый ввод символов в Java. Он реализует интерфейсы `AutoCloseable`, `Closeable` и `Readable`. Все методы этого класса, за исключением метода `markSupported()`, генерируют исключение типа `IOException` при возникновении ошибок. В табл. 20.4 приведена краткая сводка методов из класса `Reader`.

Таблица 20.4. Методы из класса `Reader`

Метод	Описание
<code>abstract void close()</code>	Закрывает поток ввода. При последующих попытках чтения данных из этого потока ввода генерируется исключение типа <code>IOException</code>
<code>void mark(int количество_символов)</code>	Размещает на текущей позиции в потоке ввода метку, которая остается достоверной до тех пор, пока не будет прочитано заданное <i>количество_символов</i>
<code>boolean markSupported()</code>	Возвращает логическое значение <code>true</code> , если в потоке ввода поддерживаются методы <code>mark()</code> и <code>reset()</code>
<code>int read()</code>	Возвращает целочисленное представление следующего символа, доступного в вызывающем потоке ввода. По достижении конца файла возвращается значение <code>-1</code>
<code>int read(char буфер[])</code>	Пытается прочитать в указанный <i>буфер</i> количество символов, равное <i>буфер.длина</i> , возвращая количество успешно прочитанных символов. По достижении конца файла возвращается значение <code>-1</code>
<code>int read(CharBuffer буфер)</code>	Пытается прочитать символы в указанный <i>буфер</i> , возвращая количество успешно прочитанных символов. По достижении конца файла возвращается значение <code>-1</code>
<code>abstract int read(char буфер[], int смещение, int количество_символов)</code>	Пытается прочитать в указанный <i>буфер</i> заданное <i>количество_символов</i> , начиная с позиции <code>буфер[смещение]</code> и возвращая количество успешно прочитанных символов. По достижении конца файла возвращается значение <code>-1</code>
<code>boolean ready()</code>	Возвращает логическое значение <code>true</code> , если следующий запрос на ввод не будет ждать, а иначе — логическое значение <code>false</code>
<code>void reset()</code>	Перемещает указатель ввода на установленную ранее метку
<code>long skip(long количество_символов)</code>	Пропускает заданное для ввода <i>количество_символов</i> , возвращая количество фактически пропущенных символов

Класс `Writer`

Класс `Writer` является абстрактным и определяет потоковый вывод символов в Java. Этот класс реализует интерфейсы `AutoCloseable`, `Closeable`, `Flushable` и `Appendable`. При возникновении ошибок все методы этого класса генерируют исключение типа `IOException`. В табл. 20.5 приведена краткая сводка методов из класса `Writer`.

Таблица 20.5. Методы из класса `Writer`

Метод	Описание
<code>Writer append(char символ)</code>	Присоединяет указанный <i>символ</i> в конце вызывающего потока вывода. Возвращает ссылку на вызывающий поток вывода
<code>Writer append(CharSequence символы)</code>	Присоединяет указанные <i>символы</i> в конце вызывающего потока вывода. Возвращает ссылку на вызывающий поток вывода
<code>Writer append(CharSequence символы, int начало, int конец)</code>	Присоединяет указанные <i>символы</i> в заданных пределах от <i>начало</i> и до <i>конец</i> -1 в конце вызывающего потока вывода. Возвращает ссылку на вызывающий поток вывода
<code>abstract void close()</code>	Закрывает вызывающий поток вывода. Последующие попытки вывода в этот поток приведут к генерированию исключения типа <code>IOException</code>
<code>abstract void flush()</code>	Делает конечным состояние вывода, очищая все буфера, в том числе и буфера вывода
<code>void write(int символ)</code>	Записывает единственный символ в вызывающий поток вывода. Обратите внимание на то, что параметр <i>символ</i> относится к типу <code>int</code> , что позволяет вызывать метод <code>write()</code> в выражении, не прибегая к приведению обратно к типу <code>char</code> . Но выводятся только младшие 16 бит указанного символа
<code>void write(char буфер[])</code>	Записывает весь заданный массив символов в вызывающий поток вывода
<code>abstract void write(char буфер[], int смещение, int количество_символов)</code>	Записывает указанное <i>количество_символов</i> из заданного массива <i>буфер</i> в вызывающий поток вывода, начиная с позиции <i>буфер[смещение]</i>
<code>void write(String строка)</code>	Записывает указанную <i>строку</i> в вызывающий поток вывода
<code>void write(String строка, int смещение, int количество_символов)</code>	Записывает указанное <i>количество_символов</i> из заданной <i>строки</i> в вызывающий поток вывода, начиная с позиции, обозначаемой параметром <i>смещение</i>

Класс `FileReader`

Этот класс является производным от класса `Reader` и служит для чтения содержимого файла. Ниже приведены два наиболее употребительных конструктора этого класса.

```
FileReader(String путь_к_файлу)  
FileReader(File объект_файла)
```

Здесь параметр *путь_к_файлу* обозначает имя полного пути к файлу, а параметр *объект_файла* — объект типа `File`, описывающий файл. Оба конструктора могут сгенерировать исключение типа `FileNotFoundException`.

В приведенном ниже примере программы показано, как организовать построчное чтение и запись данных из файла в стандартный поток вывода. Программа читает собственный исходный файл, который должен находиться в текущем каталоге.

```
// Продемонстрировать применение класса FileReader  
// В этой программе используется оператор try с ресурсами.  
// Требуется установка комплекта JDK, начиная с версии 7  
  
import java.io.*;  
  
class FileReaderDemo {  
    public static void main(String args[]) {  
        try ( FileReader fr = new FileReader("FileReaderDemo.java") )  
        {  
            int c;  
  
            // прочитать и вывести содержимое файла  
            while((c = fr.read()) != -1) System.out.print((char) c);  
  
        } catch(IOException e) {  
            System.out.println("Ошибка ввода-вывода: " + e);  
        }  
    }  
}
```

Класс `FileWriter`

Этот класс создает поток вывода типа `Writer` для записи данных в файл. Ниже приведены наиболее употребительные конструкторы класса `FileWriter`.

```
FileWriter(String путь_к_файлу)  
FileWriter(String путь_к_файлу, boolean добавить)  
FileWriter(File объект_файла)  
FileWriter(File объект_файла, boolean присоединить)
```

Здесь параметр *путь_к_файлу* обозначает имя полного пути к файлу, а параметр *объект_файла* — объект типа `File`, описывающий файл. Если параметр *присоединить* принимает логическое значение `true`, то выводимые данные присоединяются в конце файла. Все конструкторы данного класса могут генерировать исключение типа `IOException`.

Создание объекта типа `FileWriter` не зависит от того, существует ли файл. Когда создается объект типа `FileWriter`, то попутно создается и файл, прежде

чем открыть его для вывода. Если же предпринимается попытка открыть файла, доступный только для чтения, то генерируется исключение типа `IOException`.

В приведенном ниже примере представлена переделанная под ввод-вывод символов версия программы из рассмотренного ранее примера, демонстрировавшего применение класса `FileOutputStream`. В этой версии организуется буфер символов для хранения образца текста. С этой целью сначала создается объект типа `String`, а затем вызывается метод `getChars()` для извлечения эквивалентного символьного массива. Далее создаются три файла. Первый файл, `file.txt`, должен содержать каждый второй символ из образца текста, второй файл, `file2.txt`, — все символы из образца текста, а третий файл, `file3.txt`, — только последнюю четверть символов из образца текста.

```
// Продемонстрировать применение класса FileWriter
// В этой программе используется оператор try с ресурсами.
// Требуется установка комплекта JDK, начиная с версии 7

import java.io.*;

class FileWriterDemo {
    public static void main(String args[]) throws IOException {
        String source = "Now is the time for all good men\n" +
            "    to come to the aid of their country\n" +
            "    and pay their due taxes.";
        char buffer[] = new char[source.length()];
        source.getChars(0, source.length(), buffer, 0);

        try ( FileWriter f0 = new FileWriter("file1.txt");
            FileWriter f1 = new FileWriter("file2.txt");
            FileWriter f2 = new FileWriter("file3.txt") )
        {
            // вывести символы в первый файл
            for (int i=0; i < buffer.length; i += 2) {
                f0.write(buffer[i]);
            }

            // вывести символы во второй файл
            f1.write(buffer);

            // вывести символы в третий файл
            f2.write(buffer, buffer.length - buffer.length/4, buffer.length/4);
        } catch (IOException e) {
            System.out.println("Произошла ошибка ввода-вывода");
        }
    }
}
```

Класс `CharArrayReader`

Класс `CharArrayReader` реализует поток вывода, использующий массив в качестве источника данных. У этого класса имеются два конструктора, каждый из которых принимает массив символов в качестве источника данных.

```
CharArrayReader(char массив[])
CharArrayReader(char массив[], int начало, int количество_символов)
```

Здесь параметр *массив* обозначает источник ввода данных. Вторым конструктор создает объект класса, производного от класса `Reader`, из подмножества массива символов, начинающегося с позиции, обозначаемой параметром *начало*, и длиной, определяемой параметром *количество_символов*.

Метод `close()`, реализуемый классом `CharArrayReader`, не генерирует исключений. Это связано с тем, что его вызов не может завершиться неудачно. В следующем примере применяется пара объектов класса `CharArrayReaders`:

```
// Продемонстрировать применение класса CharArrayReader
// В этой программе используется оператор try с ресурсами.
// Требуется установка комплекта JDK, начиная с версии 7

import java.io.*;

public class CharArrayReaderDemo {
    public static void main(String args[]) {
        String tmp = "abcdefghijklmnopqrstuvwxyz";
        int length = tmp.length();
        char c[] = new char[length];

        tmp.getChars(0, length, c, 0);
        int i;

        try (CharArrayReader input1 = new CharArrayReader(c))
        {
            System.out.println("input1:");
            while((i = input1.read()) != -1) {
                System.out.print((char)i);
            }
            System.out.println();
        } catch(IOException e) {
            System.out.println("Ошибка ввода-вывода: " + e);
        }

        try (CharArrayReader input2 = new CharArrayReader(c, 0, 5))
        {
            System.out.println("input2:");
            while((i = input2.read()) != -1) {
                System.out.print((char)i);
            }
            System.out.println();
        } catch(IOException e) {
            System.out.println("Ошибка ввода-вывода: " + e);
        }
    }
}
```

Объект `input1` создается с использованием всего английского алфавита в нижнем регистре, в то время как объект `input2` содержит только первые пять букв. Эта программа выводит следующий результат:

```
input1:
abcdefghijklmnopqrstuvwxyz
input2:
abcde
```


Класс CharArrayWriter

Класс CharArrayWriter реализует поток вывода, использующий массив в качестве адресата для выводимых данных. У класса CharArrayWriter имеются два конструктора:

```
CharArrayWriter()
CharArrayWriter(int количество_символов)
```

В первой форме конструктора создается буфер размером, выбираемым по умолчанию. Во второй форме буфер создается размером, задаваемым параметром количество_символов. Буфер находится в поле buf класса CharArrayWriter. Размер буфера будет последовательно увеличиваться по мере надобности. Количество байтов, содержащихся в буфере, находится в поле count того же класса. Оба поля, buf и count, являются защищенными.

Метод close() не оказывает никакого влияния на класс CharArrayWriter. В приведенном ниже примере представлена переделанная под ввод-вывод символов версия программы из рассмотренного ранее примера, демонстрировавшего применение класса ByteArrayOutputStream. А в этой версии демонстрируется применение класса CharArrayWriter, хотя выводимый результат оказывается таким же, как и в предыдущей версии.

```
// Продемонстрировать применение класса CharArrayWriter
// В этой программе используется оператор try с ресурсами.
// Требуется установка комплекта JDK, начиная с версии 7

import java.io.*;

class CharArrayWriterDemo {
    public static void main(String args[]) throws IOException {
        CharArrayWriter f = new CharArrayWriter();
        String s = " Эти данные должны быть выведены в массив";
        char buf[] = new char[s.length()];

        s.getChars(0, s.length(), buf, 0);

        try {
            f.write(buf);
        } catch(IOException e) {
            System.out.println("Ошибка записи в буфер");
            return;
        }

        System.out.println("Буфер в виде символьной строки");
        System.out.println(f.toString());
        System.out.println("В массив");

        char c[] = f.toCharArray();
        for (int i=0; i<c.length; i++) {
            System.out.print(c[i]);
        }

        System.out.println("\nВ поток вывода типа FileWriter()");

        // использовать оператор try с ресурсами для управления
        // потоком ввода-вывода в файл
        try ( FileWriter f2 = new FileWriter("test.txt") )
        {
```

```
        f.writeTo(f2);
    } catch(IOException e) {
        System.out.println("Ошибка ввода-вывода: " + e);
    }

    System.out.println("Установка в исходное состояние");
    f.reset();

    for (int i=0; i<3; i++) f.write('X');

    System.out.println(f.toString());
}
}
```

Класс `BufferedReader`

Класс `BufferedReader` увеличивает производительность благодаря буферизации ввода. У него имеются следующие два конструктора:

```
BufferedReader(Reader поток_ввода)
BufferedReader(Reader поток_ввода, int размер_буфера)
```

В первой форме конструктора создается буферизованный поток ввода символов, использующий размер буфера по умолчанию. Во второй форме конструктора задается *размер_буфера*.

Закрытие потока типа `BufferedReader` приводит также к закрытию базового потока, определяемого параметром *поток_ввода*. Аналогично потоку ввода байтов, буферизованный поток ввода символов также поддерживает механизм перемещения обратно по потоку ввода в пределах доступного буфера. Для этой цели в классе `BufferedReader` реализуются методы `mark()` и `reset()`, а метод `BufferedReader.markSupported()` возвращает логическое значение `true`. В версии JDK 8 класс `BufferedReader` дополнен новым методом `lines()`. Этот метод возвращает ссылку типа `Stream` на последовательность строк, введенных из потока чтения. (Класс `Stream` входит в состав прикладного программного интерфейса API потоков данных, обсуждаемого в главе 29.)

В приведенном ниже примере представлена переделанная под ввод-вывод символов версия программы из рассмотренного ранее примера, демонстрировавшего применение класса `BufferedInputStream`. В новой версии демонстрируется применение класса `BufferedReader` для организации потока буферизованного ввода. Как и прежде, для синтаксического анализа с целью обнаружить ссылку на элемент HTML-разметки знака авторского права в данной версии программы используются методы `mark()` и `reset()`. Такая ссылка начинается со знака амперсанда (&) и оканчивается точкой с запятой (;) без всяких промежуточных пробелов. Образец введенных данных содержит два амперсанда, чтобы наглядно показать, когда происходит установка в исходное состояние с помощью метода `reset()` и когда этого не происходит. Эта версия программы выводит такой же результат, как и предыдущая ее версия:

```
// Использовать буферизованный ввод.
// В этой программе применяется оператор try с ресурсами.
// Требуется установка комплекта JDK, начиная с версии 7
```

```

import java.io.*;

class BufferedReaderDemo {
    public static void main(String args[]) throws IOException {
        String s = " Это знак авторского права &copy; " +
            ", a &copy; – нет.\n";
        char buf[] = new char[s.length()];
        s.getChars(0, s.length(), buf, 0);

        CharArrayReader in = new CharArrayReader(buf);
        int c;
        boolean marked = false;

        try ( BufferedReader f = new BufferedReader(in) )
        {
            while ((c = f.read()) != -1) {
                switch(c) {
                    case '&':
                        if (!marked) {
                            f.mark(32);
                            marked = true;
                        } else {
                            marked = false;
                        }
                        break;
                    case ';':
                        if (marked) {
                            marked = false;
                            System.out.print("(c)");
                        } else
                            System.out.print((char) c);
                        break;
                    case ' ':
                        if (marked) {
                            marked = false;
                            f.reset();
                            System.out.print("&");
                        } else
                            System.out.print((char) c);
                        break;
                    default:
                        if (!marked)
                            System.out.print((char) c);
                        break;
                }
            }
        } catch(IOException e) {
            System.out.println("Ошибка ввода-вывода: " + e);
        }
    }
}

```

Класс **BufferedWriter**

Класс **BufferedWriter** является производным от класса **Writer** и буферизует выводимые данные. Применяя класс **BufferedWriter**, можно повысить производительность за счет снижения количества операций физической записи в устройство вывода.

У класса `BufferedWriter` имеются два конструктора:

```
BufferedWriter(Writer поток_вывода)
BufferedWriter(Writer поток_вывода, int размер_буфера)
```

В первой форме конструктора создается буферизованный поток вывода, использующий буфер размером, выбираемым по умолчанию. А во второй форме задается конкретный *размер_буфера*.

Класс `PushbackReader`

Класс `PushbackReader` позволяет вернуть в поток ввода один или больше символов, чтобы просматривать этот поток, не вводя из него данные. Ниже приведены два конструктора данного класса.

```
PushbackReader(Reader поток_ввода)
PushbackReader(Reader поток_ввода, int размер_буфера)
```

В первой форме конструктора создается буферизованный поток ввода, в который можно вернуть один символ, а во второй задается конкретный *размер_буфера* для возврата символов обратно в поток ввода.

При закрытии потока типа `PushbackReader` также базовый поток, определяемый параметром *поток_ввода*. В классе `PushbackReader` предоставляется метод `unread()`, возвращающий один или больше символов в вызывающий поток ввода. Ниже приведены три общие формы объявления этого метода.

```
void unread(int символ) throws IOException
void unread(char буфер[]) throws IOException
void unread(char буфер[], int смещение, int количество_символов)
    throws IOException
```

В первой форме в поток ввода возвращается указанный *символ*. Это будет следующий символ, возвращаемый при последующем вызове метода `read()`. Во второй форме в поток ввода возвращаются символы из указанного *буфера*. А в третьей форме в поток ввода возвращается заданное *количество_символов* из указанного *буфера*, начиная с позиции *смещение*. Исключение типа `IOException` генерируется при попытке вернуть символ в поток ввода, когда буфер возврата заполнен.

В приведенном ниже примере представлена переделанная версия программы из рассмотренного ранее примера, демонстрировавшего применение класса `PushbackInputStream`. В новой версии демонстрируется применение класса `PushbackReader`, но, как и прежде, данный пример показывает, как возврат данных (в данном случае символов) в поток ввода можно использовать в синтаксическом анализаторе языка программирования для различения операций сравнения (`==`) и присваивания (`=`). Результат выполнения данной версии программы такой же, как и в прежней ее версии.

```
// Продемонстрировать применение метода unread()
// из класса PushbackInputStream.
// В этой программе применяется оператор try с ресурсами.
// Требуется установка комплекта JDK, начиная с версии 7
import java.io.*;

class PushbackReaderDemo {
```

```

public static void main(String args[]) {
    String s = "if (a == 4) a = 0;\n";
    char buf[] = new char[s.length()];
    s.getChars(0, s.length(), buf, 0);
    CharArrayReader in = new CharArrayReader(buf);

    int c;

    try ( PushbackReader f = new PushbackReader(in) )
    {
        while ((c = f.read()) != -1) {
            switch(c) {
                case '=':
                    if ((c = f.read()) == '=')
                        System.out.print(".eq.");
                    else {
                        System.out.print("<-");
                        f.unread(c);
                    }
                    break;
                default:
                    System.out.print((char) c);
                    break;
            }
        } catch(IOException e) {
            System.out.println("Ошибка ввода-вывода: " + e);
        }
    }
}

```

Класс `PrintWriter`

Класс `PrintWriter`, по существу, является символьной версией класса `PrintStream`. Он реализует интерфейсы `Appendable`, `Closeable` и `Flushable`. У класса `PrintWriter` имеется несколько конструкторов. Рассмотрим сначала следующие формы конструкторов этого класса:

```

PrintWriter(OutputStream поток_вывода)
PrintWriter(OutputStream поток_вывода, boolean автоочистка)
PrintWriter(Writer поток_вывода)
PrintWriter(Writer поток_вывода, boolean автоочистка)

```

где параметр *поток_вывода* обозначает открытый поток вывода типа `OutputStream`, который будет принимать выводимые данные. Параметр *автоочистка* определяет, будет ли буфер вывода автоматически очищаться всякий раз, когда вызывается метод `println()`, `printf()` или `format()`. Если параметр *автоочистка* принимает логическое значение `true`, то происходит автоматическая очистка буфера вывода. А если этот параметр принимает логическое значение `false`, то очистка буфера вывода не производится автоматически. Конструкторы, не принимающие параметр *автоочистка*, не производят очистку буфера вывода автоматически.

Следующий ряд конструкторов предоставляет простую возможность создать объект класса `PrintWriter` для вывода данных в файл:

```

PrintWriter(File файл_вывода) throws FileNotFoundException
PrintWriter(File файл_вывода, String набор_символов)
    throws FileNotFoundException, UnsupportedEncodingException
PrintWriter(String имя_файла_вывода) throws FileNotFoundException
PrintWriter(String имя_файла_вывода, String набор_символов)
    throws FileNotFoundException, UnsupportedEncodingException

```

Эти конструкторы позволяют создать объект класса `PrintWriter` из объекта типа `File` или по имени файла. Но в любом случае файл создается автоматически. Любой существующий файл с тем же именем уничтожается. Как только поток вывода будет создан в виде объекта класса `PrintWriter`, он будет направлять все выводимые данные в указанный файл. Конкретную кодировку символов можно задать в качестве параметра *набор_символов*.

Класс `PrintWriter` предоставляет методы `print()` и `println()` для всех типов, включая тип `Object`. Если аргумент не относится к примитивному типу, методы из класса `PrintWriter` вызывают сначала метод `toString()` такого объекта, а затем выводят результат его выполнения.

В классе `PrintWriter` поддерживается также метод `printf()`. Он действует точно так же, как и в описанном ранее классе `PrintStream`, позволяя задать точный формат данных. Метод `printf()` объявляется в классе `PrintWriter` следующим образом:

```

PrintWriter printf(String форматирующая_строка, Object ... аргументы)
PrintWriter printf(Locale региональные_настройки,
    String форматирующая_строка, Object ... аргументы)

```

В первой форме данного метода заданные *аргументы* выводятся в стандартный поток вывода в формате, указанном в качестве параметра *форматирующая_строка*, с учетом региональных настроек по умолчанию. А во второй форме можно указать конкретные региональные настройки. Но в любом случае возвращается вызывающий поток вывода в виде объекта типа `PrintWriter`.

В классе `PrintWriter` поддерживается также метод `format()`. Ниже приведены общие формы данного метода. Этот метод действует подобно методу `printf()`.

```

PrintWriter format(String форматирующая_строка, Object ... аргументы)
PrintWriter format(Locale региональные_настройки,
    String форматирующая_строка, Object ... аргументы)

```

Класс Console

Класс `Console` был введен в состав пакета `java.io` в версии JDK 6. Он служит для ввода-вывода данных на консоль, если таковая имеется, и реализует интерфейс `Flushable`. Класс `Console` является служебным, поскольку он функционирует главным образом через стандартные потоки ввода-вывода `System.in` и `System.out`. Тем не менее он упрощает некоторые виды консольных операций, особенно при чтении символьных строк с консоли.

Конструкторы в классе `Console` не предоставляются. Его объект получается в результате вызова метода `System.console()`, как показано ниже.

```

static System.console()

```

Если консоль доступна, то возвращается ссылка на нее, в противном случае — пустое значение `null`. Консоль будет доступна не во всех классах, и если возвращается пустое значение `null`, то консольные операции ввода-вывода невозможны.

В классе `Console` определяются методы, перечисленные в табл. 20.6. Следует иметь в виду, что методы ввода, например метод `readLine()`, генерируют исключение типа `IOException`, когда возникают ошибки ввода. Класс исключения `IOException` является производным от класса `Error` и обозначает фатальную ошибку ввода-вывода, которая не поддается контролю в прикладной программе. Это означает, что исключение типа `IOException` обычно не перехватывается. Откровенно говоря, если исключение типа `IOException` возникает при обращении к консоли, обычно это свидетельствует об аварийном сбое системы.

Следует также иметь в виду, что методы типа `readPassword()` позволяют считывать пароль, не выводя его на экран. Читая пароли, следует “обнулять” как массив, содержащий символьную строку, введенную пользователем, так и массив, содержащий правильный пароль, с которым требуется сравнить эту строку. Благодаря этому уменьшается вероятность того, что вредоносная программа получит пароль, просмотрев оперативную память.

Таблица 20.6. Методы из класса `Console`

Метод	Описание
<code>void flush()</code>	Выполняет физический вывод буферизованных данных на консоль
<code>Console format(String <i>форматирующая_строка</i>, Object... <i>аргументы</i>)</code>	Выводит на консоль указанные <i>аргументы</i> , используя формат, определяемый параметром <i>форматирующая_строка</i>
<code>Console printf(String <i>форматирующая_строка</i>, Object... <i>аргументы</i>)</code>	Выводит на консоль указанные <i>аргументы</i> , используя формат, определяемый параметром <i>форматирующая_строка</i>
<code>Reader reader()</code>	Возвращает ссылку на поток чтения типа Reader , связанный с консолью
<code>String readLine()</code>	Читает и возвращает символьную строку, введенную с клавиатуры. Ввод прекращается нажатием клавиши <Enter>. Если достигнут конец потока ввода с консоли, то возвращается пустое значение <code>null</code> . А если происходит фатальная ошибка ввода, то генерируется исключение типа IOException
<code>String readLine(String <i>форматирующая_строка</i>, Object... <i>аргументы</i>)</code>	Выводит строку приглашения, используя формат, определяемый параметром <i>форматирующая_строка</i> , а также указанные <i>аргументы</i> , затем читает и возвращает символьную строку, введенную с клавиатуры. Ввод прекращается нажатием клавиши <Enter>. Если достигнут конец потока ввода с консоли, то возвращается пустое значение <code>null</code> . А если происходит фатальная ошибка ввода, то генерируется исключение типа IOException

Окончание табл. 20.6

Метод	Описание
<code>char[] readPassword()</code>	Читает и возвращает символьную строку, введенную с клавиатуры. Ввод прекращается нажатием клавиши <Enter>. Введенная строка не выводится на экран. Если достигнут конец потока ввода с консоли, то возвращается пустое значение <code>null</code> . А если происходит неустраняемая ошибка ввода, то генерируется исключение типа <code>IOException</code>
<code>char[] readPassword(String форматизирующая_строка, Object... аргументы)</code>	Выводит строку приглашения, используя формат, определяемый параметром <i>форматирующая_строка</i> , а также указанные <i>аргументы</i> , затем читает и возвращает символьную строку, введенную с клавиатуры. Ввод прекращается нажатием клавиши <Enter>. Введенная строка не выводится на экран. Если достигнут конец потока ввода с консоли, то возвращается пустое значение <code>null</code> . А если происходит фатальная ошибка ввода, то генерируется исключение типа <code>IOException</code>
<code>PrintWriter writer()</code>	Возвращает ссылку на поток записи типа <code>Writer</code> , связанный с консолью

В следующем примере программы демонстрируется применение класса `Console`:

```
// Продемонстрировать применение класса Console
import java.io.*;

class ConsoleDemo {
    public static void main(String args[]) {
        String str;
        Console con;

        // получить ссылку на консоль
        con = System.console();

        // выйти из программы, если консоль недоступна
        if(con == null) return;

        // прочитать строку и вывести ее
        str = con.readLine("Введите строку: ");
        con.printf("Введенная вами строка: %s\n", str);
    }
}
```

Ниже приведен пример выполнения данной программы.

```
Введите строку: Это тест.
Введенная вами строка: Это тест.
```

Сериализация

Сериализация — это процесс записи состояния объектов в поток вывода байтов. Она оказывается удобной в том случае, когда требуется сохранить состояние при-

кладной программы в таком месте постоянного хранения, как файл. В дальнейшем эти объекты можно восстановить в процессе десериализации.

Сериализация также требуется для реализации *удаленного вызова методов* (RMI – Remote Method Invocation). Механизм RMI позволяет объекту Java на одной машине обращаться к методу объекта Java на другой машине. Объект может быть представлен в виде аргумента этого удаленного метода. Передающая машина сериализует и посылает объект, а принимающая машина десериализует его. (Подробнее механизм RMI рассматривается в главе 30.)

Допустим, сериализуемый объект ссылается на объекты, которые, в свою очередь, ссылаются на какие-нибудь другие объекты. Такой ряд объектов и отношений между ними образует направленный граф, где могут присутствовать и циклические ссылки. Иными словами, объект X может содержать ссылку на объект Y, а объект Y – обратную ссылку на объект X. Объекты могут также содержать ссылки на самих себя. Для правильного разрешения подобных ситуаций и предназначены средства сериализации и десериализации объектов. Если попытаться сериализовать объект, находящийся на вершине направленного графа, то все прочие объекты, на которые делаются ссылки, также будут рекурсивно найдены и сериализованы. Аналогичным образом все эти объекты и их ссылки правильно восстанавливаются в процессе десериализации. Ниже делается краткий обзор интерфейсов и классов, поддерживающих сериализацию.

Интерфейс `Serializable`

Средствами сериализации может быть сохранен и восстановлен только объект класса, реализующего интерфейс `Serializable`. В интерфейсе `Serializable` не определяется никаких членов. Он служит лишь для того, чтобы указать, что класс может быть сериализован. Если класс сериализуется, то сериализуются и все его подклассы.

Переменные, объявленные как `transient`, не сохраняются средствами сериализации. Не сохраняются и статические переменные.

Интерфейс `Externalizable`

Средства Java для сериализации и десериализации разработаны таким образом, чтобы большая часть операций сохранения и восстановления состояния объекта выполнялась автоматически. Но иногда требуется управлять этим процессом вручную, например, чтобы воспользоваться алгоритмами сжатия и шифрования данных. Именно для таких случаев и предназначен интерфейс `Externalizable`.

В интерфейсе `Externalizable` определяются следующие методы:

```
void readExternal(ObjectInput поток_ввода)
    throws IOException, ClassNotFoundException
void writeExternal(ObjectOutput поток_вывода)
    throws IOException
```

В этих методах параметр `поток_ввода` обозначает поток байтов, из которого может быть введен объект, а параметр `поток_вывода` – поток байтов, куда это объект может быть выведен.

Интерфейс `ObjectOutput`

Интерфейс `ObjectOutput` расширяет интерфейсы `AutoCloseable` и `DataOutput`, поддерживая сериализацию объектов. В нем определяются методы, перечисленные в табл. 20.7. Следует особо отметить метод `writeObject()`, который вызывается для сериализации объекта. При возникновении ошибок все методы этого интерфейса генерируют исключение типа `IOException`.

Таблица 20.7. Методы из интерфейса `ObjectOutput`

Метод	Описание
<code>void close()</code>	Закрывает вызывающий поток вывода. Последующие попытки вывести данные в этот поток приведут к генерированию исключения типа <code>IOException</code>
<code>void flush()</code>	Делает конечным состояние вывода, чтобы очистить все буфера, в том числе и буфера вывода
<code>void write(byte буфер[])</code>	Записывает массив байтов в вызывающий поток вывода
<code>void write(byte буфер[], int смещение, int количество_байтов)</code>	Записывает заданное количество байтов из указанного массива <code>буфер</code> , начиная с позиции <code>буфер[смещение]</code>
<code>void write(int b)</code>	Записывает одиночный байт в вызывающий поток вывода. Из указанного аргумента <code>b</code> выводится только младший байт
<code>void writeObject(Object объект)</code>	Записывает заданный объект в вызывающий поток вывода

Класс `ObjectOutputStream`

Класс `ObjectOutputStream` расширяет класс `OutputStream` и реализует интерфейс `ObjectOutput`. Этот класс отвечает за вывод объекта в поток. Ниже приведен конструктор этого класса.

`ObjectOutputStream(OutputStream поток_вывода) throws IOException`

Аргумент `поток_вывода` обозначает поток, в который могут быть выведены сериализуемые объекты. Закрытие потока вывода типа `ObjectOutputStream` приводит также к закрытию базового потока, определяемого аргументом `поток_вывода`.

Некоторые наиболее употребительные методы из класса `ObjectOutputStream` перечислены в табл. 20.8. При возникновении ошибки все они генерируют исключение типа `IOException`. Имеется также внутренний класс `PutField`, вложенный в класс `ObjectOutputStream`. Он упрощает запись постоянных полей, но описание его применения выходит за рамки данной книги.

Таблица 20.8. Наиболее употребительные методы из класса `ObjectOutputStream`

Метод	Описание
<code>void close()</code>	Закрывает вызывающий поток вывода. Последующие попытки вывода данных в этот поток приведут к генерированию исключения типа <code>IOException</code> . Базовый поток вывода также закрывается
<code>void flush()</code>	Делает конечным состояние вывода, очищая все буфера, в том числе и буфера вывода
<code>void write(byte буфер[])</code>	Записывает массив байтов в вызывающий поток вывода
<code>void write(byte буфер[], int смещение, int количество_байтов)</code>	Записывает в вызывающий поток вывода заданное <i>количество_байтов</i> из указанного массива <i>буфер</i> , начиная с позиции <i>буфер[смещение]</i>
<code>void write(int b)</code>	Записывает одиночный байт в вызывающий поток вывода. Из указанного аргумента <i>b</i> записывается только младший байт
<code>void writeBoolean(boolean b)</code>	Записывает логическое значение типа <code>boolean</code> в вызывающий поток вывода
<code>void writeByte(int b)</code>	Записывает значение типа <code>byte</code> в вызывающий поток вывода. Выводимый байт является младшим из указанного аргумента <i>b</i>
<code>void writeBytes(String строка)</code>	Записывает байты, составляющие заданную <i>строку</i> , в вызывающий поток вывода
<code>void writeChar(int c)</code>	Записывает заданное значение <i>c</i> типа <code>char</code> в вызывающий поток вывода
<code>void writeChars(String строка)</code>	Записывает символы, составляющие заданную <i>строку</i> , в вызывающий поток вывода
<code>void writeDouble(double d)</code>	Записывает заданное значение <i>d</i> типа <code>double</code> в вызывающий поток вывода
<code>void writeFloat(float f)</code>	Записывает заданное значение <i>f</i> типа <code>float</code> в вызывающий поток вывода
<code>void writeInt(int i)</code>	Записывает заданное значение <i>i</i> типа <code>int</code> в вызывающий поток вывода
<code>void writeLong(long l)</code>	Записывает заданное значение <i>l</i> типа <code>long</code> в вызывающий поток вывода
<code>final void writeObject(Object объект)</code>	Записывает заданный <i>объект</i> в вызывающий поток вывода
<code>void writeShort(int i)</code>	Записывает заданное значение <i>i</i> типа <code>short</code> в вызывающий поток вывода

Интерфейс `ObjectInput`

Интерфейс `ObjectInput` расширяет интерфейсы `AutoCloseable` и `DataInput`. В нем определяются методы, перечисленные в табл. 20.9, и поддерживается

сериализация объектов. Следует особо отметить метод `readObject()`, который вызывается для десериализации объекта. При возникновении ошибок все методы данного интерфейса генерируют исключение типа `IOException`. Метод `readObject()` может также сгенерировать исключение типа `ClassNotFoundException`.

Таблица 20.9. Методы из интерфейса `ObjectInput`

Метод	Описание
<code>int available()</code>	Возвращает количество байтов, доступных на данный момент в буфере ввода
<code>void close()</code>	Закрывает вызывающий поток ввода. Последующие попытки ввода данных из этого потока приведут к генерированию исключения типа <code>IOException</code> . Базовый поток ввода также закрывается
<code>int read()</code>	Возвращает целочисленное представление следующего байта, доступного для ввода. По достижении конца файла возвращается значение <code>-1</code>
<code>int read(byte буфер[])</code>	Пытается прочитать в указанный <i>буфер</i> количество байтов, равное <i>буфер.длина</i> , возвращая количество успешно прочитанных байтов. По достижении конца файла возвращается значение <code>-1</code>
<code>int read(byte буфер[], int смещение, int количество_байтов)</code>	Пытается прочитать в указанный <i>буфер</i> заданное <i>количество_байтов</i> , начиная с позиции <i>буфер[смещение]</i> и возвращая количество успешно прочитанных байтов. По достижении конца файла возвращается значение <code>-1</code>
<code>Object readObject()</code>	Читает объект из вызывающего потока ввода
<code>Long skip(long количество_байтов)</code>	Игнорирует (т.е. пропускает) заданное для ввода <i>количество_байтов</i> , возвращая количество фактически проигнорированных байтов

Класс `ObjectInputStream`

Этот класс расширяет класс `InputStream` и реализует интерфейс `ObjectInput`. Класс `ObjectInputStream` отвечает за ввод объектов из потока. Ниже приведен конструктор этого класса.

`ObjectInputStream(InputStream поток_ввода) throws IOException`

Аргумент *поток_ввода* обозначает поток, из которого должен быть введен сериализованный объект. Закрытие потока ввода типа `ObjectInputStream` приводит также к закрытию базового потока ввода, определяемого аргументом *поток_ввода*.

Некоторые наиболее употребительные методы из класса `ObjectInputStream` перечислены в табл. 20.10. При возникновении ошибки все они генерируют исключение типа `IOException`. Метод `readObject()` может также сгенерировать исключение типа `ClassNotFoundException`. Имеется также внутренний класс `GetField`, вложенный в класс `ObjectInputStream`. Он упрощает чтение постоянных полей, но описание его применения выходит за рамки данной книги.

Таблица 20.10. Наиболее употребительные методы из класса `ObjectInputStream`

Метод	Описание
<code>int available()</code>	Возвращает количество байтов, доступных в данный момент в буфере ввода
<code>void close()</code>	Закрывает вызывающий поток ввода. Последующие попытки ввода данных из этого потока приведут к генерированию исключения типа <code>IOException</code> . Базовый поток ввода также закрывается
<code>int read()</code>	Возвращает целочисленное представление следующего байта, доступного для ввода. По достижении конца файла возвращается значение <code>-1</code>
<code>int read(byte буфер[], int смещение, int количество_байтов)</code>	Пытается прочитать заданное <i>количество_байтов</i> в указанный <i>буфер</i> , начиная с позиции <i>буфер[смещение]</i> и возвращая количество байтов, которые удалось прочитать. По достижении конца файла возвращается значение <code>-1</code>
<code>Boolean readBoolean()</code>	Читает и возвращает логическое значение типа <code>boolean</code> из вызывающего потока ввода
<code>byte readByte()</code>	Читает и возвращает значение типа <code>byte</code> из вызывающего потока ввода
<code>char readChar()</code>	Читает и возвращает значение типа <code>char</code> из вызывающего потока ввода
<code>double readDouble()</code>	Читает и возвращает значение типа <code>double</code> из вызывающего потока ввода
<code>double readFloat()</code>	Читает и возвращает значение типа <code>float</code> из вызывающего потока ввода
<code>void readFully(byte буфер[])</code>	Читает в указанный <i>буфер</i> количество байтов, равное <i>буфер.длина</i> . Возвращает управление только тогда, когда прочитаны все байты
<code>void readFully(byte буфер[], int смещение, int количество_байтов)</code>	Читает заданное <i>количество_байтов</i> в указанный <i>буфер</i> , начиная с позиции <i>буфер[смещение]</i> . Возвращает управление только тогда, когда прочитано заданное <i>количество_байтов</i>
<code>int readInt()</code>	Читает и возвращает значение типа <code>int</code> из вызывающего потока ввода
<code>int readLong()</code>	Читает и возвращает значение типа <code>long</code> из вызывающего потока ввода
<code>final Object readObject()</code>	Читает и возвращает объект из вызывающего потока ввода
<code>short readShort()</code>	Читает и возвращает значение типа <code>short</code> из вызывающего потока ввода
<code>int readUnsignedByte()</code>	Читает и возвращает значение типа <code>unsigned byte</code> из вызывающего потока ввода
<code>int readUnsignedShort()</code>	Читает и возвращает значение типа <code>unsigned short</code> из вызывающего потока ввода

Пример сериализации

В приведенном ниже примере программы демонстрируется применение сериализации и десериализации объектов. Эта программа начинается с создания экземпляра объекта типа `MyClass`. У этого объекта имеются три переменные экземпляра типа `String`, `int` и `double`. Данные, хранящиеся именно в этих переменных, требуется сохранять и восстанавливать.

С этой целью в данной программе создается поток вывода типа `FileOutputStream`, ссылающийся на файл по имени "serial", а для него — поток вывода типа `ObjectOutputStream`. Затем для сериализации объекта вызывается метод `writeObject()` из класса `ObjectOutputStream`. По завершении данного процесса очищается и закрывается поток вывода объектов.

Далее создается поток ввода типа `FileInputStream`, ссылающийся на файл по имени "serial", а для него — поток ввода типа `ObjectInputStream`. Для последующей десериализации объекта вызывается метод `readObject()` из класса `ObjectInputStream`. По завершении данного процесса очищается и закрывается поток ввода объектов.

Обратите внимание на то, что объект типа `MyClass` определяется для реализации интерфейса `Serializable`. Если не сделать этого, будет сгенерировано исключение типа `NotSerializableException`. Поэкспериментируйте с этой программой, объявляя как `transient` некоторые переменные экземпляра класса `MyClass`. Хранящиеся в них данные не будут сохраняться при сериализации.

```
// Продемонстрировать применение сериализации и десериализации
// В этой программе используется оператор try с ресурсами.
// Требуется установка комплекта JDK, начиная с версии 7
```

```
import java.io.*;

public class SerializationDemo {
    public static void main(String args[]) {
        // произвести сериализацию объекта

        try ( ObjectOutputStream objOStrm =
              new ObjectOutputStream(new FileOutputStream("serial")) )
        {
            MyClass object1 = new MyClass("Hello", -7, 2.7e10);
            System.out.println("object1: " + object1);

            objOStrm.writeObject(object1);
        }
        catch (IOException e) {
            System.out.println(
                "Исключение при сериализации: " + e);
        }

        // произвести десериализацию объекта
        try ( ObjectInputStream objIStrm =
              new ObjectInputStream(new FileInputStream("serial")) )
        {
            MyClass object2 = (MyClass)objIStrm.readObject();
            System.out.println("object2: " + object2);
        }
        catch (Exception e) {
```

```
        System.out.println(
            "Исключение при десериализации: " + e);
        System.exit(0);
    }
}

class MyClass implements Serializable {
    String s;
    int i;
    double d;

    public MyClass(String s, int i, double d) {
        this.s = s;
        this.i = i;
        this.d = d;
    }

    public String toString() {
        return "s=" + s + "; i=" + i + "; d=" + d;
    }
}
```

Эта программа демонстрирует идентичность переменных экземпляра `object1` и `object2`. Ниже приведен результат ее выполнения.

```
object1: s=Hello; i=-7; d=2.7E10
object2: s=Hello; i=-7; d=2.7E10
```

Преимущества потоков ввода-вывода

Потоковый интерфейс ввода-вывода в Java предоставляет чистую абстракцию для решения сложных и зачастую обременительных задач. Структура классов фильтрующих потоков позволяет динамически строить собственные настраиваемые потоковые интерфейсы, отвечающие требованиям передачи данных. Те программы на Java, где применяются эти классы с высоким уровнем абстракции, в том числе `InputStream`, `OutputStream`, `Reader` и `Writer`, будут правильно функционировать и впредь — даже в том случае, если появятся новые и усовершенствованные конкретные классы потоков ввода-вывода. Как будет показано в главе 22, эта модель оказывается вполне работоспособной при переходе от ряда потоков ввода-вывода в файлы к потокам ввода-вывода через сеть и сокет. И наконец, сериализация объектов играет важную роль в самых разных программах на Java. Классы сериализации ввода-вывода в Java обеспечивают переносимое решение этой порой не совсем простой задачи.

Система ввода-вывода NIO

Начиная с версии 1.4 в Java предоставляется вторая система ввода-вывода под названием NIO (сокращение от New I/O — новый ввод-вывод). В этой системе поддерживается канальный подход к операциям ввода-вывода, ориентированный на применение буферов. А в версии JDK 7 система ввода-вывода NIO была существенно расширена, и теперь она оказывает улучшенную поддержку средств обработки файлов и файловых систем. На самом деле изменения в этой системе настолько значительны, что она нередко обозначается термином *NIO.2*. Благодаря возможностям, предоставляемым новыми классами файлов из системы ввода-вывода NIO, ожидается, что значение этой системы в обработке файлов будет только возрастать. В этой главе рассматривается ряд основных характеристик системы ввода-вывода NIO.

Классы системы ввода-вывода NIO

В табл. 21.1 перечислены пакеты, в которых содержатся классы системы ввода-вывода NIO.

Таблица 21.1. Пакеты, содержащие классы системы ввода-вывода NIO

Пакет	Назначение
<code>java.nio</code>	Это пакет верхнего уровня в системе ввода-вывода NIO. Он инкапсулирует различные типы буферов, содержащих данные, которыми оперирует система ввода-вывода NIO
<code>java.nio.channels</code>	Поддерживает каналы, открывающие соединения для ввода-вывода
<code>java.nio.channels.spi</code>	Поддерживает поставщики услуг для каналов
<code>java.nio.charset</code>	Инкапсулирует наборы символов. Поддерживает также функционирование кодеров и декодеров для взаимного преобразования символов и байтов
<code>java.nio.charset.spi</code>	Поддерживает поставщики услуг для наборов символов
<code>java.nio.file</code>	Поддерживает ввод-вывод в файлы
<code>java.nio.file.attribute</code>	Поддерживает атрибуты файлов
<code>java.nio.file.spi</code>	Поддерживает поставщики услуг для файловых систем

Прежде чем приступить к рассмотрению системы ввода-вывода NIO, следует заметить, что эта система не предназначена для замены классов ввода-вывода, входящих в состав пакета `java.io` и представленных в главе 20. Напротив, практические знания о классах из этого пакета помогают легче усвоить систему ввода-вывода NIO.

На заметку! В этой главе предполагается, что вы уже проработали материал глав 13 и 20, посвященный принципам ввода-вывода вообще и потокового ввода-вывода в частности.

Основные положения о системе ввода-вывода NIO

Система ввода-вывода NIO построена на двух основополагающих элементах: буферах и каналах. В *буфере* хранятся данные, а *канал* предоставляет открытое соединение с устройством ввода-вывода, например файлом или сокетом. В общем, для применения системы ввода-вывода NIO требуется получить канал для устройства ввода-вывода и буфер для хранения данных. После этого можно обращаться с буфером, вводя или выводя данные по мере необходимости. Поэтому в последующих разделах буфера и каналы будут рассмотрены подробно.

Буфера

Буфера определяются в пакете `java.nio`. Все буфера являются подклассами, производными от класса `Buffer`, в котором определяются основные функциональные возможности, характерные для каждого буфера, в том числе текущая позиция, предел и емкость. *Текущая позиция* определяет индекс в буфере, с которого в следующий раз начнется операция чтения или записи данных. Текущая позиция перемещается после выполнения большинства операций чтения или записи. *Предел* определяет значение индекса за позицией последней доступной ячейки в буфере. *Емкость* определяет количество элементов, которые можно хранить в буфере. Зачастую предел равен емкости буфера. В классе `Buffer` поддерживается также отметка и очистка буфера. В нем определяется ряд методов, перечисленных в табл. 21.2.

Таблица 21.2. Методы из класса `Buffer`

Метод	Описание
<code>abstract Object array()</code>	Возвращает ссылку на массив, если вызывающий буфер поддерживается массивом, иначе генерирует исключение типа <code>UnsupportedOperationException</code> . Если же массив доступен только для чтения, то генерируется исключение типа <code>ReadOnlyBufferException</code>
<code>abstract int arrayOffset()</code>	Возвращает индекс первого элемента массива, если вызывающий буфер поддерживается массивом, а иначе генерируется исключение типа <code>UnsupportedOperationException</code> . Если же массив доступен только для чтения, то генерируется исключение типа <code>ReadOnlyBufferException</code>

Окончание табл. 21.2

Метод	Описание
final int capacity()	Возвращает количество элементов, которые можно хранить в вызывающем буфере
final Buffer clear()	Очищает вызывающий буфер и возвращает ссылку на него
final Buffer flip()	Задаёт текущую позицию в качестве предела для вызывающего буфера и затем устанавливает текущую позицию в нуль. Возвращает ссылку на буфер
abstract boolean hasArray()	Возвращает логическое значение true , если вызывающий буфер поддерживается массивом, доступным для чтения и записи, а иначе — логическое значение false
final boolean hasRemaining()	Возвращает логическое значение true , если в вызывающем буфере ещё остались какие-нибудь элементы, а иначе — логическое значение false
abstract boolean isDirect()	Возвращает логическое значение true , если вызывающий буфер оказывается прямым. Иными словами, операции ввода-вывода выполняются над ним напрямую. В противном случае возвращается логическое значение false
abstract boolean isReadOnly()	Возвращает логическое значение true , если вызывающий буфер является буфером только для чтения, а иначе — логическое значение false
final int limit()	Возвращает предел для вызывающего буфера
final Buffer limit(int n)	Задаёт предел <i>n</i> для вызывающего буфера. Возвращает ссылку на буфер
final Buffer mark()	Устанавливает метку и возвращает ссылку на вызывающий буфер
final int position()	Возвращает текущую позицию
final Buffer position(int n)	Задаёт текущую позицию буфера равной <i>n</i> . Возвращает ссылку на буфер
int remaining()	Возвращает количество элементов, доступных до того, как будет достигнут предел. Иными словами, возвращается предел минус текущая позиция
final Buffer reset()	Устанавливает текущую позицию в вызывающем буфере на установленной ранее метке. Возвращает ссылку на буфер
final Buffer rewind()	Устанавливает текущую позицию в вызывающем буфере в нуль. Возвращает ссылку на буфер

От класса `Buffer` происходят приведенные ниже классы конкретных буферов, где тип хранимых данных можно определить по их именам. Класс `MappedByteBuffer` является производным от класса `ByteBuffer` и используется для сопоставления файла с буфером.

ByteBuffer	CharBuffer	DoubleBuffer	FloatBuffer
IntBuffer	LongBuffer	MappedByteBuffer	ShortBuffer

Все упомянутые выше буфера предоставляют различные методы `get()` и `put()`, которые позволяют получать данные из буфера или вносить их в него. (Разумеется, метод `put()` недоступен, если буфер предназначен только для чтения.) В табл. 21.3 перечислены методы `get()` и `put()`, определенные в классе `ByteBuffer`. Другие классы буферов имеют похожие методы. Во всех классах буферов поддерживают-

ся также методы, выполняющие различные операции с буфером. Например, с помощью метода `allocate()` можно вручную выделить оперативную память под буфер, с помощью метода `wrap()` — организовать массив в пределах буфера, а с помощью метода `slice()` — создать подпоследовательность в буфере.

Таблица 21.3. Методы `get()` и `put()` из класса `ByteBuffer`

Метод	Описание
<code>abstract byte get()</code>	Возвращает байт на текущей позиции
<code>ByteBuffer get(byte значения[])</code>	Копирует вызывающий буфер в заданный массив <i>значения</i> . Возвращает ссылку на буфер. Если же в буфере не осталось больше элементов, количество которых равно <i>значения.length</i> , то генерируется исключение типа <code>BufferUnderflowException</code>
<code>ByteBuffer get(byte значения[], int начало, int количество)</code>	Копирует заданное <i>количество</i> элементов из вызывающего буфера в указанный массив <i>значения</i> , начиная с позиции по индексу <i>начало</i> . Возвращает ссылку на буфер. Если в буфере больше не осталось заданное <i>количество</i> элементов, то генерируется исключение типа <code>BufferUnderflowException</code>
<code>abstract byte get(int индекс)</code>	Возвращает из вызывающего буфера байт по указанному <i>индексу</i>
<code>abstract ByteBufferput(byte b)</code>	Копирует заданный байт <i>b</i> на текущую позицию в вызывающем буфере. Возвращает ссылку на буфер. Если буфер заполнен, то генерируется исключение типа <code>BufferOverflowException</code>
<code>final ByteBufferput(byte значения[])</code>	Копирует все элементы из указанного массива <i>значения</i> в вызывающий буфер, начиная с текущей позиции. Возвращает ссылку на буфер. Если буфер не может вместить все элементы, то генерируется исключение типа <code>BufferOverflowException</code>
<code>ByteBuffer put(byte значения[], int начало, int количество)</code>	Копирует в вызывающий буфер заданное <i>количество</i> элементов из указанного массива <i>значения</i> , начиная с указанной позиции <i>начало</i> . Возвращает ссылку на буфер. Если буфер не может хранить все элементы, то генерируется исключение типа <code>BufferOverflowException</code>
<code>ByteBufferput(ByteBuffer bb)</code>	Копирует элементы из заданного буфера <i>bb</i> в вызывающий буфер, начиная с текущей позиции. Если буфер не может хранить все элементы, то генерируется исключение типа <code>BufferOverflowException</code> . Возвращает ссылку на буфер
<code>abstract ByteBufferput(int индекс, byte b)</code>	Копирует байт <i>b</i> на позицию по указанному <i>индексу</i> в вызывающем буфере. Возвращает ссылку на буфер

Каналы

Каналы определены в пакете `java.nio.channels`. Канал представляет открытое соединение с источником или адресатом ввода-вывода. Классы каналов реализуют ин-

терфейс `Channel`, расширяющий интерфейс `Closeable`, а начиная с JDK 7 — интерфейс `AutoCloseable`. При реализации интерфейса `AutoCloseable` каналами можно управлять в блоке оператора `try` с ресурсами, где канал закрывается автоматически, когда он больше не нужен. (Подробнее об операторе `try` с ресурсами см. в главе 13.)

Один из способов получения канала подразумевает вызов метода `getChannel()` для объекта, поддерживающего каналы. Например, метод `getChannel()` поддерживается в следующих классах ввода-вывода:

DatagramSocket	FileInputStream	FileOutputStream
RandomAccessFile	ServerSocket	Socket

Конкретный тип возвращаемого канала зависит от типа объекта, для которого вызывается метод `getChannel()`. Например, когда метод `getChannel()` вызывается для объекта типа `FileInputStream`, `FileOutputStream` или `RandomAccessFile`, он возвращает канал типа `FileChannel`. А если этот метод вызывается для объекта типа `Socket`, то он возвращает канал типа `SocketChannel`.

Еще один способ получения канала подразумевает использование одного из статических методов, определенных в классе `Files`, который был введен в версии JDK 7. Например, используя класс `Files`, можно получить байтовый канал при вызове метода `newByteChannel()`. Он возвращает канал типа `SeekableByteChannel`, т.е. интерфейса, реализуемого классом `FileChannel`. (Более подробно класс `Files` рассматривается далее в этой главе.)

В каналах типа `FileChannel` и `SocketChannel` поддерживаются различные методы `read()` и `write()`, которые позволяют выполнять операции ввода-вывода через канал. Например, в табл. 21.4 перечислен ряд методов `read()` и `write()`, определенных в классе `FileChannel`.

Таблица 21.4. Методы `read()` и `write()` из класса `FileChannel`

Метод	Описание
abstract int read(ByteBuffer bb) throws IOException	Считывает байты из вызывающего канала в указанный буфер <i>bb</i> до тех пор, пока буфер не будет заполнен или же не исчерпаны вводимые данные. Возвращает количество прочитанных байтов
abstract int read(ByteBuffer bb, long начало) throws IOException	Считывает байты из вызывающего канала в указанный буфер <i>bb</i> , начиная с позиции <i>начало</i> и до тех пор, пока буфер не будет заполнен или же не исчерпаны вводимые данные. Текущая позиция не изменяется. Возвращает количество прочитанных байтов или значение <code>-1</code> , если позиция <i>начало</i> окажется за пределами файла
abstract int write(ByteBuffer bb) throws IOException	Записывает содержимое байтового буфера в вызывающий канал, начиная с текущей позиции. Возвращает количество записанных байтов
abstract int write(ByteBuffer bb, long начало) throws IOException	Записывает содержимое байтового буфера в вызывающий канал, начиная с позиции <i>начало</i> в файле. Текущая позиция не изменяется. Возвращает количество записанных байтов

Все каналы поддерживают дополнительные методы, предоставляющие доступ к каналу и позволяющие управлять им. Например, канал типа `FileChannel` поддерживает среди прочего методы для получения и установки текущей позиции, передачи данных между файловыми каналами, получения текущего размера канала и его блокировки. В классе `FileChannel` предоставляется статический метод `open()`, который открывает файл и возвращает для него канал. Такой результат достигается другим способом получения канала. В классе `FileChannel` предоставляется также метод `map()`, с помощью которого можно сопоставить файл с буфером.

Наборы символов и селекторы

В системе ввода-вывода NIO применяются наборы символов и селекторы. *Набор символов* определяет способ сопоставления байтов с символами. С помощью *кодера* можно закодировать последовательность символов в виде байтов. Процесс декодирования производится с помощью *декодера*. Наборы символов, кодеры и декодеры поддерживаются в классах, определяемых в пакете `java.nio.charset`. Кодеры и декодеры предоставляются по умолчанию, и поэтому обращаться непосредственно к наборам символов приходится крайне редко.

Селектор обеспечивает возможность многоканального ввода-вывода по ключам, не прибегая к блокировке. Иными словами, с помощью селекторов можно выполнять операции ввода-вывода через несколько каналов. Селекторы поддерживаются классами, определяемыми в пакете `java.nio.channels`. Они чаще всего применяются в каналах, опирающихся на сокет. В примерах, представленных в этой главе, наборы символов и селекторы не применяются, тем не менее они могут оказаться полезными в ряде приложений.

Усовершенствования в системе NIO, начиная с версии JDK 7

В версии JDK 7 система ввода-вывода NIO была значительно расширена и усовершенствована. Помимо поддержки оператора `try` с ресурсами, который обеспечивает автоматическое управление ресурсами, усовершенствования включают три новых пакета (`java.nio.file`, `java.nio.file.attribute` и `java.nio.file.spi`), несколько новых классов, интерфейсов и методов, а также прямую поддержку потокового ввода-вывода. Эти усовершенствования существенно расширили возможности для применения системы ввода-вывода NIO, особенно в файлы. В следующих разделах описывается ряд ключевых дополнений данной системы.

Интерфейс `Path`

Возможно, одним из наиболее важных дополнений системы ввода-вывода NIO является интерфейс `Path`, поскольку он инкапсулирует путь к файлу. Как будет показано далее, интерфейс `Path` служит связующим звеном для большинства новых файловых средств в системе ввода-вывода NIO.2. Он описывает расположе-

ние файла в структуре каталогов. Интерфейс `Path` находится в пакете `java.nio.file` и наследует интерфейсы `Watchable`, `Iterable<Path>` и `Comparable<Path>`. Интерфейс `Watchable` описывает объект, который можно наблюдать и изменять. Интерфейсы `Iterable` и `Comparable` были представлены ранее в данной книге.

В интерфейсе `Path` объявляется немало методов для манипулирования путями к файлам. Некоторые из них приведены в табл. 21.5. Обратите особое внимание на метод `getName()`. Он служит для получения элемента пути. С этой целью в данном методе применяется индекс. Нулевому значению индекса соответствует ближайшая к корневому каталогу часть пути, являющаяся его крайним слева элементом. Последующие индексы определяют элементы вправо от корневого каталога. Количество элементов в пути может быть получено в результате вызова метода `getNameCount()`. Если же требуется получить строковое представление всего пути, достаточно вызвать метод `toString()`. Следует также заметить, что для распознавания относительного и абсолютного пути достаточно вызвать метод `resolve()`.

Таблица 21.5. Избранные методы из интерфейса `Path`

Метод	Описание
<code>boolean endsWith(String <i>путь</i>)</code>	Возвращает логическое значение true , если вызывающий объект типа Path оканчивается путем, определяемым параметром <i>путь</i> , а иначе — логическое значение false
<code>boolean endsWith(Path <i>путь</i>)</code>	Возвращает логическое значение true , если вызывающий объект типа Path оканчивается путем, определяемым параметром <i>путь</i> , а иначе — логическое значение false
<code>Path getFileName()</code>	Возвращает имя файла, связанное с вызывающим объектом типа Path
<code>Path getName(int <i>индекс</i>)</code>	Возвращает объект типа Path , содержащий имя элемента пути по указанному <i>индексу</i> в вызывающем объекте. Крайний слева элемент имеет нулевой индекс и находится ближе всего к корневому каталогу. А крайний справа элемент имеет индекс <code>getNameCount() - 1</code>
<code>int getNameCount()</code>	Возвращает количество элементов (кроме корневого) в вызывающем объекте типа Path
<code>Path getParent()</code>	Возвращает объект типа Path , который содержит весь путь, кроме имени файла, определяемого вызывающим объектом типа Path
<code>Path getRoot()</code>	Возвращает корневой каталог из вызывающего объекта типа Path
<code>boolean isAbsolute()</code>	Возвращает логическое значение true , если вызывающий объект типа Path обозначает абсолютный путь, а иначе — логическое значение false

Метод	Описание
Path resolve(Path путь)	Если указанный <i>путь</i> является абсолютным, то возвращается именно он. А если указанный <i>путь</i> не содержит корневой каталог, то этот <i>путь</i> предваряется корневым каталогом из вызывающего объекта типа Path , а затем возвращается полученный результат. Если же указанный <i>путь</i> пуст, то возвращается вызывающий объект типа Path . В противном случае поведение данного метода не определено
Path resolve(String путь)	Если указанный <i>путь</i> является абсолютным, возвращается именно этот <i>путь</i> . А если указанный <i>путь</i> не содержит корневой каталог, то этот <i>путь</i> предваряется корневым каталогом из вызывающего объекта типа Path , а затем возвращается полученный результат. Если же указанный <i>путь</i> пуст, то возвращается вызывающий объект типа Path . В противном случае поведение данного метода не определено
boolean startsWith(String путь)	Возвращает логическое значение true , если вызывающий объект типа Path начинается с указанного <i>пути</i> , а иначе — логическое значение false
boolean startsWith(Path путь)	Возвращает логическое значение true , если вызывающий объект типа Path начинается с указанного <i>пути</i> , а иначе — логическое значение false
Path toAbsolutePath()	Возвращает вызывающий объект типа Path в виде абсолютного пути
String toString()	Возвращает строковое представление вызывающего объекта типа Path

Следует также иметь в виду, что при обновлении унаследованного кода, в котором используется класс `File`, определенный в пакете `java.io`, экземпляр класса `File` можно преобразовать в экземпляр интерфейса `Path`, вызвав метод `toPath()` для объекта типа `File`. Этот метод был введен в класс `File` в версии `JDK 7`. Кроме того, экземпляр класса `File` можно получить, вызвав метод `toFile()`, определяемый в интерфейсе `Path`.

Класс Files

Большинство действий, которые выполняются над файлами, предоставляются статическими методами из класса `Files`. Путь к файлу, над которым выполняются определенные действия, задает объект типа `Path`. Таким образом, методы из класса `Files` используют объект типа `Path`, чтобы указать используемый файл. Класс `Files` обладает обширным рядом функциональных возможностей. Так, в нем имеются методы, позволяющие открывать или создавать файл по указанному пути. Кроме того, из объекта типа `Path` можно получить следующие сведения о файле: является ли он исполняемым, скрытым или доступным только для чтения. В классе

Files предоставляются также методы, позволяющие копировать или перемещать файлы. Некоторые методы, определенные в этом классе, перечислены в табл. 21.6. Помимо исключения типа `IOException`, возможны и другие исключения. В версии JDK 8 класс `Files` дополнен следующими четырьмя методами: `list()`, `walk()`, `lines()` и `find()`. Все эти методы возвращают объект типа `Stream`. Они способствуют интеграции системы ввода-вывода NIO с новым прикладным программным интерфейсом API потоков ввода-вывода, определенным в версии JDK 8 и описываемым в главе 29.

Таблица 21.6. Избранные методы из класса `Files`

Метод	Описание
<code>static Path copy(Path <i>источник</i>, Path <i>адресат</i> CopyOption ... <i>способ</i>) throws IOException</code>	Копирует файл из <i>источника</i> по указанному <i>адресату</i> заданным <i>способам</i>
<code>static Path createDirectory(Path <i>путь</i>, FileAttribute<?> ... <i>атрибуты</i>) throws IOException</code>	Создает каталог по указанному <i>пути</i> . Атрибуты каталога определяются параметром <i>атрибуты</i>
<code>static Path createFile(Path <i>путь</i>, FileAttribute<?> ... <i>атрибуты</i>) throws IOException</code>	Создает файл по указанному <i>пути</i> . Атрибуты файла определяются параметром <i>атрибуты</i>
<code>static void delete(Path <i>путь</i>) throws IOException</code>	Удаляет файл по указанному <i>пути</i>
<code>static boolean exists(Path <i>путь</i>, LinkOptions ... <i>параметры</i>)</code>	Возвращает логическое значение true , если файл существует по указанному <i>пути</i> , а иначе — логическое значение false . Если же аргумент <i>параметры</i> не определен, то используются символические ссылки. С целью предотвратить следование по символическим ссылкам аргумент <i>параметры</i> должен принимать значение NOFOLLOW_LINKS
<code>static boolean isDirectory(Path <i>путь</i>, LinkOptions ... <i>параметры</i>)</code>	Возвращает логическое значение true , если параметр <i>путь</i> определяет каталог, а иначе — логическое значение false . Если же аргумент <i>параметры</i> не определен, то используются символические ссылки. С целью предотвратить следование по символическим ссылкам заданный аргумент <i>параметры</i> должен принимать значение NOFOLLOW_LINKS
<code>static boolean isExecutable(Path <i>путь</i>)</code>	Возвращает логическое значение true , если файл по указанному <i>пути</i> является исполняемым, а иначе — логическое значение false
<code>static boolean isHidden(Path <i>путь</i>) throws IOException</code>	Возвращает логическое значение true , если файл по указанному <i>пути</i> является скрытым, а иначе — логическое значение false
<code>static boolean isReadable(Path <i>путь</i>)</code>	Возвращает логическое значение true , если файл по указанному <i>пути</i> доступен для чтения, а иначе — логическое значение false

Метод	Описание
<code>static boolean isRegularFile(Path <i>путь</i>, LinkOptions ... <i>параметры</i>)</code>	Возвращает логическое значение true , если параметр <i>путь</i> определяет файл, а иначе — логическое значение false . Если же аргумент <i>параметры</i> не определен, то используются символические ссылки. С целью предотвратить следование по символическим ссылкам аргумент <i>параметры</i> должен принимать значение NOFOLLOW_LINKS
<code>static boolean isWritable(Path <i>путь</i>)</code>	Возвращает логическое значение true , если файл по указанному <i>пути</i> доступен для записи, а иначе — логическое значение false
<code>static Path move(Path <i>источник</i>, Path <i>адресат</i>, CopyOption ... <i>способ</i>) throws IOException</code>	Копирует файл из <i>источника</i> по указанному <i>адресату</i> заданным <i>способом</i>
<code>static SeekableByteChannel newByteChannel(Path <i>путь</i>, OpenOption ... <i>способ</i>) throws IOException</code>	Открывает файл по указанному <i>пути</i> заданным <i>способом</i> . Возвращает для файла байтовый канал типа SeekableByteChannel . Текущая позиция в этом канале может быть изменена. Интерфейс SeekableByteChannel реализуется классом FileChannel
<code>static DirectoryStream<Path> newDirectoryStream(Path <i>путь</i>) throws IOException</code>	Открывает каталог по указанному <i>пути</i> . Возвращает поток ввода каталога типа DirectoryStream , связанный с каталогом
<code>static InputStream newInputStream(Path <i>путь</i>, OpenOption ... <i>способ</i>) throws IOException</code>	Открывает файл по указанному <i>пути</i> заданным <i>способом</i> . Возвращает поток ввода типа InputStream , связанный с файлом
<code>static OutputStream newOutputStream(Path <i>путь</i>, OpenOption ... <i>способ</i>) throws IOException</code>	Открывает файл по указанному <i>пути</i> заданным <i>способом</i> . Возвращает поток вывода типа OutputStream , связанный с файлом
<code>static boolean notExists(Path <i>путь</i>, LinkOption ... <i>параметры</i>)</code>	Возвращает логическое значение true , если файл по указанному <i>пути</i> не существует, а иначе — логическое значение false . Если же аргумент <i>параметры</i> не определен, то используются символические ссылки. С целью предотвратить следование по символическим ссылкам аргумент <i>параметры</i> должен принимать значение NOFOLLOW_LINKS
<code>static <A extends BasicFileAttributes> A readAttributes(Path <i>путь</i>, Class<A> <i>тип_атрибута</i>, LinkOption ... <i>параметры</i>) throws IOException</code>	Получает атрибуты, связанные с файлом. Тип передаваемых атрибутов определяется параметром <i>тип_атрибута</i> . Если же аргумент <i>параметры</i> не определен, то используются символические ссылки. С целью предотвратить следование по символическим ссылкам аргумент <i>параметры</i> должен принимать значение NOFOLLOW_LINKS
<code>static long size(Path <i>путь</i>) throws IOException</code>	Возвращает размер файла по указанному <i>пути</i>

Обратите внимание на то, что некоторые методы, перечисленные в табл. 21.6, получают аргумент типа `OpenOption`. Это интерфейс, описывающий способ открытия файла. Он реализуется классом `StandardOpenOption`, где определяется перечисление, значения которого представлены в табл. 21.7.

Таблица 21.7. Стандартные значения параметров открытия файлов

Значение	Назначение
APPEND	Присоединить выводимые данные в конце файла
CREATE	Создать файл, если он еще не существует
CREATE_NEW	Создать файл только в том случае, если он еще не существует
DELETE_ON_CLOSE	Удалить файл, когда он закрывается
DSYNC	Немедленно записать вносимые изменения в физический файл. Как правило, для повышения производительности изменения в файле буферизируются файловой системой и записываются только по мере надобности
READ	Открыть файл для операций ввода
SPARSE	Указать файловой системе, что файл разрежен, а следовательно, не может быть полностью заполнен данными. Если файловая система не поддерживает разреженные файлы, это значение параметра игнорируется
SYNC	Немедленно записать вносимые изменения в файл или его метаданные в физический файл. Как правило, для повышения производительности изменения в файле буферизируются файловой системой и записываются только по мере надобности
TRUNCATE_EXISTING	Укоротить до нуля длину уже существующего файла, открываемого для вывода
WRITE	Открыть файл для операций вывода

Класс `Paths`

Экземпляр типа `Path` нельзя создать непосредственно с помощью конструктора, поскольку это интерфейс, а не класс. Вместо этого можно получить объект типа `Path`, вызвав метод, который возвращает этот объект. Как правило, для этой цели служит метод `get()`, определяемый в классе `Paths`. Существуют две формы метода `get()`. Ниже приведена та его форма, которая употребляется в примерах этой главы.

```
static Path get(String имя_пути, String ... части)
```

Этот метод возвращает объект, инкапсулирующий определенный путь. Путь может быть задан двумя способами. Если параметр *части* не указан, то путь должен полностью определяться параметром *имя_пути*. В качестве альтернативы путь можно передать по частям, причем первую часть в качестве параметра *имя_пути*, а остальные части — в качестве параметра *части* переменной длины. Но в любом случае метод `get()` сгенерирует исключение типа `InvalidPathException`, если указанный путь синтаксически недостоверен.

Во второй форме метода `get()` объект типа `Path` создается из `URI`. Эта форма выглядит следующим образом:

```
static Path get(URI uri)
```

В итоге возвращается объект типа `Path`, соответствующий заданному параметру `uri`. Следует, однако, иметь в виду, что создание объекта типа `Path` не приводит к открытию или созданию файла. Вместо этого лишь создается объект, инкапсулирующий путь к каталогу, в котором находится файл.

Интерфейсы атрибутов файлов

С файлами связан ряд атрибутов, обозначающих время создания файла, время его последней модификации, размер файла или каталог. Система ввода-вывода NIO организует атрибуты файлов в виде иерархии различных интерфейсов, определенных в пакете `java.nio.file.attribute`. На вершине этой иерархии находится интерфейс `BasicFileAttributes`, инкапсулирующий ряд атрибутов, которые обычно применяются в большинстве файловых систем. Методы, определенные в интерфейсе `BasicFileAttributes`, перечислены в табл. 21.8.

Таблица 21.8. Методы из интерфейса `BasicFileAttributes`

Метод	Описание
FileTime <code>creationTime()</code>	Возвращает время создания файла. Если этот атрибут не поддерживается файловой системой, то возвращается значение, зависящее от конкретной реализации
Object <code>fileKey()</code>	Возвращает файловый ключ. Если этот атрибут не поддерживается файловой системой, возвращается пустое значение <code>null</code>
boolean <code>isDirectory()</code>	Возвращает логическое значение <code>true</code> , если файл является каталогом
boolean <code>isOther()</code>	Возвращает логическое значение <code>true</code> , если файл является символической ссылкой или каталогом, а не файлом
boolean <code>isRegularFile()</code>	Возвращает логическое значение <code>true</code> , если файл является обычным файлом, а не каталогом или символической ссылкой
boolean <code>isSymbolicLink()</code>	Возвращает логическое значение <code>true</code> , если файл является символической ссылкой
FileTime <code>lastAccessTime()</code>	Возвращает время последнего обращения к файлу. Если этот атрибут не поддерживается файловой системой, то возвращается значение, зависящее от конкретной реализации
FileTime <code>lastModifiedTime()</code>	Возвращает время последней модификации файла. Если этот атрибут не поддерживается файловой системой, то возвращается значение, зависящее от конкретной реализации
long <code>size()</code>	Возвращает размер файла

Производными от интерфейса `BasicFileAttributes` являются следующие два интерфейса: `DosFileAttributes` и `PosixFileAttributes`. В частности, интерфейс `DosFileAttributes` описывает атрибуты, связанные с файловой системой FAT, которые были первоначально определены в файловой системе DOS. В этом интерфейсе определяются методы, перечисленные в табл. 21.9.

Таблица 21.9. Методы из интерфейса `DosFileAttributes`

Метод	Описание
<code>boolean isArchive()</code>	Возвращает логическое значение true , если файл помечен как архивный, а иначе — логическое значение false
<code>boolean isHidden()</code>	Возвращает логическое значение true , если файл помечен как скрытый, а иначе — логическое значение false
<code>boolean isReadOnly()</code>	Возвращает логическое значение true , если файл помечен как доступный только для чтения, а иначе — логическое значение false
<code>boolean isSystem()</code>	Возвращает логическое значение true , если файл помечается как системный, а иначе — логическое значение false

Интерфейс `PosixFileAttributes` инкапсулирует атрибуты, определенные по стандартам POSIX (Portable Operating System Interface — переносимый интерфейс операционных систем). В этом интерфейсе определяются методы, перечисленные в табл. 21.10.

Таблица 21.10. Методы из интерфейса `PosixFileAttributes`

Метод	Описание
<code>GroupPrincipal group()</code>	Возвращает группового владельца файла
<code>UserPrincipal owner()</code>	Возвращает отдельного владельца файла
<code>Set<PosixFilePermission> permissions()</code>	Возвращает полномочия доступа к файлу

Имеются разные способы доступа к атрибутам файлов. В частности, вызвав статический метод `readAttributes()`, определенный в классе `Files`, можно получить объект, инкапсулирующий атрибуты файла. Ниже приведена одна из общих форм объявления этого метода.

```
static <A extends BasicFileAttributes>
    A readAttributes(Path путь, Class<A> тип атрибута,
        LinkOption... параметры) throws IOException
```

Этот метод возвращает ссылку на объект, обозначающий атрибуты файла по указанному пути. Конкретный тип атрибутов указывается в виде объекта типа `Class` с помощью параметра `тип_атрибута`. Например, для получения основных атрибутов файла следует передать в качестве параметра `тип_атрибута` объект типа `BasicFileAttributes.class`, для получения атрибутов DOS — объект типа `DosFileAttributes.class`, а для получения атрибутов POSIX — объект типа `PosixFileAttributes.class`. Дополнительные, но необязательные параметры ссылок передаются как аргумент `параметры`. Если же аргумент `параметры` не опре-

делен, то следуют символические ссылки. Метод `readAttributes()` возвращает ссылку на требуемый атрибут. Если же тип требуемого атрибута недоступен, то генерируется исключение типа `UnsupportedOperationException`. Используя объект, возвращаемый этим методом, можно обратиться к атрибутам файла.

Еще один способ доступа к атрибутам файла состоит в том, чтобы вызвать метод `getFileAttributeView()`, определенный в классе `Files`. В системе ввода-вывода NIO определяется несколько интерфейсов для представлений атрибутов, в том числе `AttributeView`, `BasicFileAttributeView`, `DosFileAttributeView` и `PosixFileAttributeView`. В примерах из этой главы представления атрибутов не употребляются, но в некоторых случаях это средство может оказаться полезным.

Иногда можно и не прибегать непосредственно к интерфейсам атрибутов файлов, поскольку в классе `Files` предоставляются служебные статические методы, позволяющие обращаться к некоторым атрибутам. Для этой цели в классе `Files` имеются такие методы, как `isHidden()` и `isWritable()`.

Следует, однако, иметь в виду, что все допустимые атрибуты файлов поддерживаются не во всех файловых системах. Например, атрибуты файлов DOS относятся к файловой системе FAT, хотя первоначально они были определены в файловой системе DOS. Те атрибуты, которые применяются в обширном ряде файловых систем, описаны в интерфейсе `BasicFileAttributes`. Поэтому именно они и употребляются в примерах из этой главы.

Классы `FileSystem`, `FileSystems` и `FileStore`

Для упрощения доступа к файловой системе в пакете `java.nio.file` предоставляются классы `FileSystem` и `FileSystems`. В действительности, используя метод `newFileSystem()`, определенный в классе `FileSystems`, можно даже получить новую файловую систему. А класс `FileStore` инкапсулирует систему хранения файлов. И хотя упоминаемые здесь классы не употребляются в примерах из этой главы, им можно найти применение в своих прикладных программах.

Применение системы ввода-вывода NIO

В этом разделе демонстрируется применение системы ввода-вывода NIO для решения самых разных задач. Но прежде следует подчеркнуть, что в версии JDK 7 была значительно расширена как сама система ввода-вывода NIO, так и область ее применения. Как упоминалось ранее, усовершенствованную версию этой системы иногда называют NIO.2. Вследствие столь значительных усовершенствований в системе ввода-вывода NIO.2 изменился и способ написания кода на ее основе, а также расширился круг задач, для решения которых можно ее применять. В силу этого обстоятельства в большей части примеров из этой главы применяются средства из системы ввода-вывода NIO.2, и поэтому для проработки этих примеров потребуется комплект JDK 7, JDK 8 или более поздняя его версия. Тем не менее в конце этой главы дается краткое описание кода, написанного до версии JDK 7, в помощь тем программистам, которые пользуются системой ввода-вывода до версии JDK 7 или сопровождают унаследованный код.

Помните! Для компиляции большинства примеров из этой главы требуется комплект JDK 7, JDK 8 или более поздняя его версия.

Раньше система NIO предназначалась в основном для канального ввода-вывода, и эта разновидность ввода-вывода по-прежнему остается важнейшей областью ее применения. Но теперь систему ввода-вывода NIO можно также использовать для потокового ввода-вывода и выполнения операций в файловой системе. Таким образом, обсуждение областей применения системы ввода-вывода NIO можно разделить на три части:

- канальный ввод-вывод;
- потоковый ввод-вывод;
- операции в файловой системе.

Самым распространенным средством ввода-вывода является файл на диске, поэтому в примерах, представленных далее в этой главе, употребляются файлы на диске. А поскольку все канальные операции ввода-вывода в файлы основываются на передаче байтов, то для их выполнения будут использоваться буфера типа `ByteBuffer`.

Прежде чем открыть файл для доступа к нему средствами системы ввода-вывода NIO, следует получить объект типа `Path`, описывающий этот файл. Это можно, в частности, сделать, вызвав упоминавшийся ранее фабричный метод `Paths.get()`. В приведенных далее примерах употребляется следующая общая форма объявления метода `get()`:

```
static Path get(String имя_пути, String ... части)
```

Напомним, что путь к файлу можно указать двумя способами. Во-первых, передать первую его часть в качестве параметра `имя_пути`, а остальные части — в качестве параметра `части` переменной длины. И во-вторых, указать весь путь в качестве параметра `имя_пути`, а параметр `части` опустить. Именно такой способ и применяется в рассматриваемых далее примерах.

Применение системы NIO для канального ввода-вывода

Важнейшей областью применения системы ввода-вывода NIO является получение доступа к файлу через каналы и буфера. В последующих разделах демонстрируются некоторые способы применения канала для чтения и записи данных в файл.

Чтение файла через канал

Имеется несколько способов чтения данных из файла через канал. Наиболее распространенный из них, вероятно, состоит в том, чтобы сначала выделить оперативную память под буфер вручную, а затем выполнить явным образом операцию чтения для загрузки этого буфера данными из файла. Поэтому рассмотрим этот способ в первую очередь.

Прежде чем прочитать данные из файла, его нужно открыть. Для этого сначала создается объект типа `Path`, описывающий файл, а затем он используется для открытия файла. Имеются разные способы открыть файл в зависимости от того, как он будет использоваться. В рассматриваемом здесь примере файл будет открыт для выполнения явных операций байтового ввода. Поэтому в данном примере для открытия файла и установления канала доступа к нему вызывается метод `Files.newByteChannel()`. Метод `newByteChannel()` имеет следующую общую форму:

```
static SeekableByteChannel newByteChannel(
    Path путь, OpenOption ... способ)
    throws IOException
```

Этот метод возвращает объект типа `SeekableByteChannel`, инкапсулирующий канал для файловых операций. Объект типа `Path`, описывающий файл, передается в качестве параметра *путь*. А параметр *способ* определяет порядок открытия файла. Это параметр переменной длины, и поэтому в качестве его можно указать любое количество аргументов через запятую. (Допустимые значения параметров данного метода обсуждались ранее и приведены в табл. 21.7.) Если же никаких аргументов не указано, то файл открывается для операций ввода. Интерфейс `SeekableByteChannel` описывает канал, применяемый для файловых операций. Он реализуется классом `FileChannel`. Когда используется выбираемая по умолчанию файловая система, возвращаемый объект может быть приведен к типу `FileChannel`. Завершив работу с каналом, следует закрыть его. Классы всех каналов, включая и класс `FileChannel`, реализуют интерфейс `AutoCloseable`, поэтому для автоматического закрытия файла вместо явного вызова метода `close()` можно воспользоваться оператором `try` с ресурсами. Именно такой подход и применяется в примерах из этой главы.

Затем следует получить буфер, который будет использоваться каналом, включив буфер в оболочку существующего массива или динамически выделив оперативную память под буфер. В примерах из этой главы применяется выделение оперативной памяти под буфер, но вы вольны выбрать любой из этих двух способов. Файловые каналы оперируют буферами байтов, и поэтому для их получения в примерах из этой главы вызывается метод `allocate()`, определенный в классе `ByteBuffer`. Ниже приведена его общая форма, где *емкость* обозначает конкретную емкость буфера, а в итоге возвращается ссылка на буфер.

```
static ByteBuffer allocate(int емкость)
```

После создания буфера для канала вызывается метод `read()`, которому передается ссылка на буфер. Ниже приведена общая форма метода `read()`, которая употребляется в примерах, представленных далее в главе.

```
int read(ByteBuffer буфер) throws IOException
```

При каждом вызове метода `read()` указанный буфер заполняется данными из файла. Чтение осуществляется последовательно, а следовательно, при каждом вызове метода `read()` из файла в буфер читается следующая порция байтов. Метод `read()` возвращает количество фактически прочитанных байтов. При попытке прочитать данные по достижении конца файла возвращается значение `-1`.

В приведенном ниже примере программы все изложенное выше демонстрируется на практике. В этой программе данные из файла `test.txt` читаются через канал посредством явных операций ввода.

```
// Использовать канал ввода-вывода для чтения файла.
// Требуется установка комплекта JDK, начиная с версии 7

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class ExplicitChannelRead {
    public static void main(String args[]) {
        int count;
        Path filepath = null;

        // сначала получить путь к файлу
        try {
            filepath = Paths.get("test.txt");
        } catch (InvalidPathException e) {
            System.out.println("Path Error " + e);
            return;
        }

        // затем получить канал к этому файлу в
        // блоке оператора try с ресурсами
        try (SeekableByteChannel fChan = Files.newByteChannel(filepath))
        {

            // выделить память под буфер
            ByteBuffer mBuf = ByteBuffer.allocate(128);

            do {
                // читать данные из файла в буфер
                count = fChan.read(mBuf);

                // прекратить чтение по достижении конца файла
                if(count != -1) {

                    // подготовить буфер к чтению из него данных
                    mBuf.rewind();

                    // читать байты данных из буфера и
                    // выводить их на экран как символы
                    for(int i=0; i < count; i++)
                        System.out.print((char)mBuf.get());
                }
            } while(count != -1);

            System.out.println();
        } catch (IOException e) {
            System.out.println("Ошибка ввода-вывода " + e);
        }
    }
}
```

Эта программа действует следующим образом. Сначала создается объект типа `Path`, содержащий относительный путь к файлу `test.txt`. Ссылка на этот объект присваивается переменной `filepath`. Затем для создания канала, связанного с фай-

лом, вызывается метод `newByteChannel()`, которому передается ссылка на файл в переменной `filepath`. А поскольку никаких параметров открытия файла не указано, то файл по умолчанию открывается для чтения. Обратите внимание на то, что созданный в итоге канал является объектом, управляемым оператором `try` с ресурсами. Таким образом, канал автоматически закрывается в конце блока этого оператора. Далее в данной программе вызывается метод `allocate()` из класса `ByteBuffer`, чтобы выделить оперативную память под буфер для хранения содержимого файла во время чтения. Ссылка на этот буфер хранится в переменной экземпляра `mBuf`. После этого вызывается метод `read()`, и содержимое файла читается по очереди в буфер `mBuf`. Количество прочитанных байтов сохраняется в переменной `count`. Затем вызывается метод `rewind()`, чтобы подготовить буфер к чтению из него данных. Этот метод нужно вызвать потому, что после вызова метода `read()` текущая позиция находится в конце буфера. Ее следует возратить в начало буфера, чтобы при вызове метода `get()` можно было прочитать байты данных из буфера `mBuf`. (Напомним, что метод `get()` определяется в классе `ByteBuffer`.) Буфер `mBuf` может содержать только байты данных, поэтому из метода `get()` возвращаются байты. Они приводятся к типу `char`, чтобы выводить на экран содержимое файла в текстовом виде. (В качестве альтернативы можно создать буфер, автоматически преобразующий байты в символы, а затем прочитать их из этого буфера.) По достижении конца файла метод `read()` возвращает значение `-1`. В таком случае программа завершается и канал автоматически закрывается.

Обратите внимание на следующий интересный момент: программа получает объект типа `Path` в пределах одного блока оператора `try`, а затем использует его в другом блоке оператора `try` для получения и манипулирования каналом, связанным с файлом по пути, который задается объектом типа `Path`. И хотя в таком подходе нет ничего плохого, во многих случаях его можно упростить, чтобы организовать ввод-вывод данных только в одном блоке `try`. В этом случае вызовы методов `Paths.get()` и `newByteChannel()` следуют друг за другом. В качестве примера ниже приведена переделанная версия программы из предыдущего примера, где применяется данный подход.

```
// Более компактный способ открытия канала.
// Требуется установка комплекта JDK, начиная с версии 7

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class ExplicitChannelRead {
    public static void main(String args[]) {
        int count;

        // Здесь канал открывается по пути, возвращаемому
        // методом Paths.get() в виде объекта типа Path.
        // Переменная filepath больше не нужна
        try (SeekableByteChannel fChan =
            Files.newByteChannel(Paths.get("test.txt"))) {
            {
                // выделить память под буфер
                ByteBuffer mBuf = ByteBuffer.allocate(128);

                do {
```

```

// читать данные из файла в буфер
count = fChan.read(mBuf);

// прекратить чтение по достижении конца файла
if(count != -1) {

    // подготовить буфер к чтению из него данных
    mBuf.rewind();

    // читать байты данных из буфера и
    // выводить их на экран как символы
    for(int i=0; i < count; i++)
        System.out.print((char)mBuf.get());
}
} while(count != -1);
System.out.println();
} catch(InvalidPathException e) {
    System.out.println("Ошибка указания пути " + e);
} catch (IOException e) {
    System.out.println("Ошибка ввода-вывода " + e);
}
}
}

```

В этой версии программы переменная `filepath` больше не требуется, и оба исключения обрабатываются в одном и том же блоке оператора `try`. Такой подход компактнее, поэтому именно он и применяется в остальных примерах из этой главы. Разумеется, при написании прикладного кода возможны случаи, когда создание объекта типа `Path` должно быть отделено от получения канала. В подобных случаях применяется предыдущий подход.

Другой способ чтения файла подразумевает его сопоставление с буфером. Преимущество такого способа состоит в том, что буфер автоматически получает содержимое файла. Никаких явных операций чтения не требуется. Сопоставление и чтение содержимого файла осуществляется в ходе следующей общей процедуры. Сначала получается объект типа `Path`, инкапсулирующий файл, как описано ранее. Затем получается канал к этому файлу. С этой целью вызывается метод `Files.newByteChannel()`, которому передается объект типа `Path`, а тип возвращаемого из него объекта приводится к типу `FileChannel`. Как упоминалось ранее, метод `newByteChannel()` возвращает объект типа `SeekableByteChannel`. Если используется выбираемая по умолчанию файловая система, этот объект может быть приведен к типу `FileChannel`. Затем для канала вызывается метод `map()`, чтобы сопоставить этот канал с буфером. Метод `map()` определяется в классе `FileChannel`, поэтому и требуется приведение к типу `FileChannel`. Ниже приведена общая форма объявления метода `map()`.

```

MappedByteBuffer map(
    FileChannel.MapMode способ, long позиция, long размер)
    throws IOException

```

В методе `map()` данные из файла сопоставляются с буфером в оперативной памяти. Значение параметра *способ* определяет вид разрешенной операции. Этот параметр может принимать одно из следующих допустимых значений:

<code>MapMode.READ_ONLY</code>	<code>MapMode.READ_WRITE</code>	<code>MapMode.PRIVATE</code>
--------------------------------	---------------------------------	------------------------------

Для чтения данных из файла следует указать значение `MapMode.READ_ONLY`, а для чтения и записи данных в файл — значение `MapMode.READ_WRITE`. Выбор значения `MapMode.PRIVATE` приводит к созданию закрытой копии файла, чтобы внесенные в буфере изменения не повлияли на основной файл. Место для начала сопоставления в пределах файла определяется параметром *позиция*, а количество сопоставляемых байтов — параметром *размер*. Ссылка на буфер возвращается в виде объекта класса `MappedByteBuffer`, производного от класса `ByteBuffer`. Как только файл будет сопоставлен с буфером, его содержимое можно прочитать в файл из буфера. Такой способ демонстрируется в следующем примере программы:

```
// Использовать сопоставление для чтения данных из файла.
// Требуется установка комплекта JDK, начиная с версии 7

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class MappedChannelRead {
    public static void main(String args[]) {

        // получить канал к файлу в блоке оператора try с ресурсами
        try ( FileChannel fChan =
            (FileChannel) Files.newByteChannel(Paths.get("test.txt")) )
        {

            // получить размер файла
            long fSize = fChan.size();

            // а теперь сопоставить файл с буфером
            MappedByteBuffer mBuf =
                fChan.map(FileChannel.MapMode.READ_ONLY, 0, fSize);

            // читать байты из буфера и выводить их на экран
            for(int i=0; i < fSize; i++)
                System.out.print((char)mBuf.get());

            System.out.println();

        } catch(InvalidPathException e) {
            System.out.println("Ошибка указания пути " + e);
        } catch (IOException e) {
            System.out.println("Ошибка ввода-вывода " + e);
        }
    }
}
```

В данной программе сначала создается путь к файлу, обозначаемый объектом типа `Path`, а затем открывается файл с помощью метода `newByteChannel()`. Получаемый в итоге канал приводится к типу `FileChannel` и сохраняется в переменной экземпляра `fChan`. Затем в результате вызова метода `size()` для канала получается размер файла. Далее вызывается метод `map()` по ссылке `fChan` на объект канала, чтобы сопоставить весь файл с областью в памяти, выделяемой под буфер, а ссылка на буфер сохраняется в переменной экземпляра `mBuf`. Обратите внимание на то, что переменная `mBuf` объявляется как ссылка на объект типа `MappedByteBuffer`. Байты из буфера в переменной `mBuf` читаются непосредственно методом `get()`.

Запись данных в файл через канал

Как и при чтении данных из файла, для записи данных в файл через канал имеется несколько способов. Рассмотрим сначала один из наиболее распространенных способов. Он предполагает выделение оперативной памяти под буфер вручную, запись в него данных, а затем выполнение явной операции записи этих данных в файл.

Прежде чем записать данные в файл, его следует открыть. Для этого нужно получить сначала объект типа `Path`, обозначающий путь к файлу, а затем использовать этот путь, чтобы открыть файл. В рассматриваемом здесь примере файл будет открыт для выполнения явных операций байтового ввода. Поэтому в данном примере для открытия файла и установления канала доступа к нему вызывается метод `Files.newByteChannel()`. Как показано в предыдущем разделе, общая форма метода `newByteChannel()` такова:

```
static SeekableByteChannel newByteChannel(  
    Path путь, OpenOption ... способ)  
    throws IOException
```

Этот метод возвращает объект типа `SeekableByteChannel`, инкапсулирующий канал для файловых операций. Чтобы открыть файл для вывода, в качестве параметра *способ* следует передать значение `StandardOpenOption.WRITE`. Если файл еще не существует и его нужно создать, то следует указать также значение `StandardOpenOption.CREATE`. (Другие доступные значения стандартных параметров открытия файлов перечислены в табл. 21.7.) Как пояснялось в предыдущем разделе, интерфейс `SeekableByteChannel` описывает канал, применяемый для файловых операций. Его реализует класс `FileChannel`. Когда используется выбираемая по умолчанию файловая система, возвращаемый объект может быть приведен к типу `FileChannel`. Завершив работу с каналом, следует закрыть его.

Один из способов записи данных в файл через канал подразумевает явные вызовы метода `write()`. Сначала получается объект типа `Path`, обозначающий путь к файлу, а затем для открытия этого файла вызывается метод `newByteChannel()` и возвращаемый результат приводится к типу `FileChannel`. Далее выделяется оперативная память под буфер байтов, в который записываются выводимые в файл данные. Прежде чем данные будут записаны в файл, для буфера следует вызвать метод `rewind()`, чтобы обнулить его текущую позицию. (Каждая операция вывода в буфер увеличивает его текущую позицию. Поэтому перед записью в файл ее следует вернуть в исходное положение.) Далее для канала вызывается метод `write()`, которому передается буфер. Вся эта процедура демонстрируется в следующем примере программы, где весь английский алфавит записывается в файл `test.txt`:

```
// Записать данные в файл средствами системы ввода-вывода NIO.  
// Требуется установка комплекта JDK, начиная с версии 7
```

```
import java.io.*;  
import java.nio.*;  
import java.nio.channels.*;  
import java.nio.file.*;  
  
public class ExplicitChannelWrite {
```

```

public static void main(String args[]) {
    // получить канал к файлу в блоке оператора try с ресурсами
    try ( FileChannel fChan = (FileChannel)
        Files.newByteChannel(Paths.get("test.txt"),
                               StandardOpenOption.WRITE,
                               StandardOpenOption.CREATE) )
    {
        // создать буфер
        ByteBuffer mBuf = ByteBuffer.allocate(26);

        // записать некоторое количество байтов в буфер
        for(int i=0; i<26; i++)
            mBuf.put((byte)('A' + i));

        // подготовить буфер к записи данных
        mBuf.rewind();

        // записать данные из буфера в выходной файл
        fChan.write(mBuf);
    } catch(InvalidPathException e) {
        System.out.println("Ошибка указания пути " + e);
    } catch (IOException e) {
        System.out.println("Ошибка ввода-вывода: " + e);
        System.exit(1);
    }
}
}

```

Следует отметить одну важную особенность данной программы. Как упоминалось ранее, после записи данных в буфер байтов `mBuf`, но перед их записью в файл для буфера `mBuf` вызывается метод `rewind()`. Это требуется для обнуления текущей позиции после записи данных в буфер `mBuf`. Не следует забывать, что после каждого вызова метода `put()` для буфера `mBuf` текущая позиция смещается. Поэтому текущую позицию необходимо возвратить в начало буфера, прежде чем вызывать метод `write()`. Если не сделать этого, метод `write()` не сумеет обнаружить в буфере никаких данных, посчитав, что их там вообще нет.

Еще один способ обнуления буфера между операциями ввода и вывода подразумевает вызов метода `flip()` вместо метода `rewind()`. Метод `flip()` устанавливает для текущей позиции нулевое значение, а для предела — значение предыдущей текущей позиции. В приведенном выше примере емкость буфера совпадает с его пределом, поэтому метод `flip()` можно использовать вместо метода `rewind()`. Но эти два метода взаимозаменяемы далеко не всегда.

Как правило, буфер следует обнулять между любыми операциями чтения и записи. Например, в результате выполнения приведенного ниже цикла, составленного на основе предыдущего примера, английский алфавит будет записан в файл три раза. Обратите особое внимание на то, что метод `rewind()` вызывается каждый раз в промежутке между операциям чтения и записи.

```

int h=0; h<3; h++) {
    // записать заданное количество байтов в буфер
    for(int i=0; i<26; i++)
        mBuf.put((byte)('A' + i));
}

```

```
// подготовить буфер к записи данных
mBuf.rewind();

// записать данные из буфера в выходной файл
fChan.write(mBuf);

// снова подготовить буфер к записи данных
mBuf.rewind();
}
```

В отношении рассматриваемой здесь программы следует также иметь в виду, что в процессе записи данных из буфера в файл первые 26 байт в файле будут содержать выводимые данные. Если файл `test.txt` существовал ранее, то после выполнения программы первые 26 байт в файле `test.txt` будут содержать алфавит, а остальная часть файла останется без изменения.

Еще один способ записи данных в файл подразумевает его сопоставление с буфером. Преимущество такого подхода заключается в том, что занесенные в буфер данные будут автоматически записаны в файл. Никаких явных операций записи не требуется. Для сопоставления и записи содержимого буфера в файла необходимо придерживаться следующей общей процедуры. Сначала получается объект типа `Path`, инкапсулирующий файл, а затем создается канал к этому файлу, для чего вызывается метод `Files.newByteChannel()`, которому передается объект типа `Path`. Ссылку, возвращаемую методом `newByteChannel()`, следует привести к типу `FileChannel`. Затем для канала вызывается метод `map()`, чтобы сопоставить канал с буфером. Метод `map()` был подробно описан в предыдущем разделе, а здесь он упоминается ради удобства изложения. Ниже приведена его общая форма.

```
MappedByteBuffer map(
    FileChannel.MapMode способ, long позиция, long размер)
    throws IOException
```

Метод `map()` сопоставляет данные из файла с буфером в памяти. Значение параметра *способ* определяет разрешенные операции. Чтобы записать данные в файл, в качестве параметра *способ* следует указать значение `MapMode.READ_WRITE`. Место для начала сопоставления в файле определяется параметром *позиция*, а количество сопоставляемых байтов — параметром *размер*. В итоге возвращается ссылка на буфер. Как только файл будет сопоставлен с буфером, в буфер можно вывести данные, которые будут автоматически записываться в файл. Поэтому никаких явных операций записи в канал не требуется.

Ниже приведена новая версия программы из предыдущего примера, переделанная таким образом, чтобы использовать сопоставление для записи данных в файл. Обратите внимание на то, что при вызове метода `newByteChannel()` в качестве параметра указывается значение `StandardOpenOption.READ`. Дело в том, что сопоставляемый буфер может использоваться только для чтения или же для чтения и записи. Таким образом, для записи в сопоставляемый буфер канал должен быть открыт как для чтения, так и для записи.

```
// Записать данные в сопоставляемый файл
// Требуется установка комплекта JDK, начиная с версии 7

import java.io.*;
import java.nio.*;
```

```

import java.nio.channels.*;
import java.nio.file.*;

public class MappedChannelWrite {
    public static void main(String args[]) {

        // получить канал к файлу в блоке try с ресурсами
        try ( FileChannel fChan = (FileChannel)
            Files.newByteChannel(Paths.get("test.txt"),
                StandardOpenOption.WRITE,
                StandardOpenOption.READ,
                StandardOpenOption.CREATE) )
        {

            // затем сопоставить файл с буфером
            MappedByteBuffer mBuf =
                fChan.map(FileChannel.MapMode.READ_WRITE, 0, 26);

            // записать заданное количество байтов в буфер
            for(int i=0; i<26; i++)
                mBuf.put((byte)('A' + i));
        } catch(InvalidPathException e) {
            System.out.println("Ошибка указания пути " + e);
        } catch (IOException e) {
            System.out.println("Ошибка ввода-вывода " + e);
        }
    }
}

```

Как видите, в данном примере отсутствуют явные операции записи непосредственно в канал. Буфер `mBuf` сопоставляется с файлом, поэтому изменения в буфере автоматически отражаются в основном файле.

Копирование файлов средствами системы ввода-вывода NIO

Система ввода-вывода NIO упрощает несколько видов файловых операций. Хотя здесь недостаточно места, чтобы рассмотреть все эти операции, приведенный ниже пример программы дает общее представление о доступных средствах. В этой программе файл копируется единственным методом `copy()` — статическим методом из класса `Files` в системе ввода-вывода NIO. У этого метода имеется несколько общих форм. Ниже приведена та общая форма, которая будет использоваться в примерах, представленных далее.

```

static Path copy(Path источник, Path адресат, CopyOption ... способ)
    throws IOException

```

Файл, определяемый параметром *источник*, копируется в файл, обозначаемый параметром *адресат*. А порядок копирования определяется параметром *способ*. Это параметр переменной длины, поэтому он может отсутствовать. Если же он определен, то позволяет передать одно или несколько приведенных ниже значений, допустимых для всех файловых систем. В зависимости от конкретной реализации могут поддерживаться и другие значения.

<code>StandardCopyOption.COPY_ATTRIBUTES</code>	Запросить копирование атрибутов файла
<code>StandardLinkOption.NOFOLLOW_LINKS</code>	Не следовать по символическим ссылкам
<code>StandardCopyOption.REPLACE_EXISTING</code>	Перезаписать прежний файл

В приведенном ниже примере программы демонстрируется применение метода `copy()`. Исходный и результирующие файлы указываются в командной строке, причем исходный файл указывается первым. Обратите внимание на краткость программы. Если сравнить эту версию программы копирования файла с ее аналогом из главы 13, то окажется, что та часть программы, которая фактически копирует файл, существенно короче в представленной здесь версии на основе системы ввода-вывода NIO.

```
// Скопировать файл средствами системы ввода-вывода NIO.
// Требуется установка комплекта JDK, начиная с версии 7

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class NIOCopy {

    public static void main(String args[]) {

        if(args.length != 2) {
            System.out.println("Применение: откуда и куда копировать");
            return;
        }

        try {
            Path source = Paths.get(args[0]);
            Path target = Paths.get(args[1]);

            // скопировать файл
            Files.copy(source, target,
                       StandardCopyOption.REPLACE_EXISTING);

        } catch(InvalidPathException e) {
            System.out.println("Ошибка указания пути " + e);
        } catch (IOException e) {
            System.out.println("Ошибка ввода-вывода " + e);
        }
    }
}
```

Применение системы NIO для потокового ввода-вывода

Начиная с версии NIO.2, систему NIO можно использовать для открытия потока ввода-вывода. Получив объект типа `Path`, следует открыть файл, вызвав статический метод `newInputStream()` или `newOutputStream()`, определенный в классе `Files`. Эти методы возвращают поток ввода-вывода, связанный с указанным файлом. В любом случае поток ввода-вывода может быть затем использован так, как описано в главе 20, и для этого пригодны те же самые способы. Преимущество использования объекта типа `Path` для открытия файла заключается в том, что доступны все средства системы ввода-вывода NIO.

Для открытия файла с целью потокового ввода служит метод `Files.newInputStream()`. Он имеет следующую общую форму:

```
static InputStream newInputStream(Path путь, OpenOption ... способ)
    throws IOException
```


где параметр *путь* обозначает открываемый файл, а параметр *способ* — порядок открытия файла. Это параметр переменной длины, и поэтому он должен принимать одно или несколько значений, определенных в упомянутом ранее классе `StandardOpenOption`. (Безусловно, в данном случае применимы только те значения, которые относятся к потоку ввода.) Если же параметр *способ* не определен, то файл открывается так, как будто в качестве этого параметра передано значение `StandardOpenOption.READ`. После открытия файла можно использовать любой из методов, определенных в классе `InputStream`. Например, метод `read()` можно использовать для чтения байтов из файла.

В приведенном ниже примере программы демонстрируется применение потокового ввода-вывода на основе системы NIO. Этот пример содержит версию программы `ShowFile` из главы 13, переделанную таким образом, чтобы для открытия файла и получения потока ввода-вывода использовались средства системы NIO. Нетрудно заметить, что эта версия программы очень похожа на первоначальный ее вариант, за исключением используемого интерфейса `Path` и метода `newInputStream()`.

```
/* Эта программа выводит текстовый файл, используя код
   потокового ввода-вывода на основе системы NIO.
   Требуется установка комплекта JDK, начиная с версии 7
```

Чтобы воспользоваться этой программой, укажите имя файла, который требуется просмотреть. Например, чтобы просмотреть файл **TEST.TXT**, введите в режиме командной строки следующую команду:

```
java ShowFile TEST.TXT
*/

import java.io.*;
import java.nio.file.*;

class ShowFile {
    public static void main(String args[])
    {
        int i;

        // сначала удостовериться, что указано имя файла
        if(args.length != 1) {
            System.out.println("Применение: ShowFile имя_файла");
            return;
        }

        // открыть файл и получить связанный с ним поток ввода-вывода
        try (InputStream fin = Files.newInputStream(Paths.get(args[0])))
        {
            do {
                i = fin.read();
                if(i != -1) System.out.print((char) i);
            } while(i != -1);

        } catch(InvalidPathException e) {
            System.out.println("Ошибка указания пути " + e);
        } catch(IOException e) {
            System.out.println("Ошибка ввода-вывода " + e);
        }
    }
}
```

Поток ввода-вывода, возвращаемый методом `newInputStream()`, является обычным, и поэтому он применяется как и любой другой поток ввода-вывода. Например, один поток ввода можно заключить в оболочку другого, буферизованного потока ввода, например, типа `BufferedInputStream`, чтобы обеспечить буферизацию так, как показано ниже. В итоге все операции чтения будут автоматически буферизованы.

```
new BufferedInputStream(Files.newInputStream(Paths.get(args[0])))
```

Для открытия файла с целью вывода служит метод `Files.newOutputStream()`, который имеет следующую общую форму:

```
static OutputStream newOutputStream(Path путь, OpenOption ... способ)
    throws IOException
```

где параметр *путь* обозначает открываемый файл, а параметр *способ* — порядок открытия файла. Это параметр переменной длины, и поэтому он должен принимать одно или несколько значений, определенных в упомянутом ранее классе `StandardOpenOption`. (Безусловно, в данном случае применимы только те значения, которые относятся к потоку вывода.) Если же параметр *способ* не определен, то файл открывается так, как будто в качестве этого параметра переданы значения `StandardOpenOption.WRITE`, `StandardOpenOption.CREATE` и `StandardOpenOption.TRUNCATE_EXISTING`.

Метод `newOutputStream()` применяется таким же способом, как и описанный ранее метод `newInputStream()`. После открытия файла можно вызвать любой метод, определенный в классе `OutputStream`. Например, вызвать метод `write()` для записи байтов в файл. Кроме того, поток вывода можно заключить в поток вывода байтов типа `BufferedOutputStream`, чтобы буферизовать его.

В приведенном ниже примере программы демонстрируется применение метода `newOutputStream()`. В этой программе английский алфавит записывается в файл `test.txt`. Обратите внимание на использование буферизованного ввода-вывода.

```
// Продемонстрировать потоковый вывод на основе системы NIO
// Требуется установка комплекта JDK, начиная с версии 7
```

```
import java.io.*;
import java.nio.file.*;

class NIOStreamWrite {
    public static void main(String args[])
    {
        // открыть файл и получить связанный с ним поток вывода
        try (OutputStream fout =
            new BufferedOutputStream(
                Files.newOutputStream(Paths.get("test.txt"))))
        {
            // вывести в поток заданное количество байтов
            for(int i=0; i < 26; i++)
                fout.write((byte)('A' + i));

        } catch(InvalidPathException e) {
            System.out.println("Ошибка указания пути " + e);
        } catch(IOException e) {
            System.out.println("Ошибка ввода-вывода: " + e);
        }
    }
}
```

Применение системы ввода-вывода NIO для операций в файловой системе

В начале главы 20 был представлен класс `File`, входящий в пакет `java.io`. Как упоминалось ранее, класс `File` обращается к файловой системе и оперирует различными атрибутами файлов, обозначающими, например, доступ только для чтения, скрытый файл и т.д. Он служит также для получения сведений о пути к файлу. Начиная с версии JDK 7 интерфейсы и классы, определенные в системе ввода-вывода NIO.2, предоставляют лучший способ выполнения этих операций. К их преимуществам относятся улучшенная поддержка символических ссылок, обхода дерева каталогов, усовершенствованная обработка метаданных и многое другое. В последующих подразделах приведены примеры двух наиболее распространенных операций в файловой системе: получения сведений о пути к файлу и самом файле, а также сведений о содержимом каталога.

Помните! Если требуется обновить устаревший код, в котором применяется класс `java.io.File`, новым кодом, в котором применяется интерфейс `Path`, воспользуйтесь методом `toPath()`, чтобы получить экземпляр интерфейса `Path` из экземпляра класса `File`.

Получение сведений о пути к файлу и самом файле

Сведения о пути к файлу могут быть получены методами, определенными в интерфейсе `Path`. Некоторые атрибуты файлов, описываемые в интерфейсе `Path` (например, скрытый файл), получаются методами, определенными в классе `Files`. В рассматриваемом здесь примере употребляются такие методы из интерфейса `Path`, как `getName()`, `getParent()` и `toAbsolutePath()`, а также методы `isExecutable()`, `isHidden()`, `isReadable()`, `isWritable()` и `exists()` из класса `Files` (они представлены в табл. 21.5 и 21.6).

Внимание! Такими методами, как `isExecutable()`, `isReadable()`, `isWritable()` и `exists()`, следует пользоваться осторожно, поскольку состояние файловой системы после их вызова может измениться, что может привести к нарушению нормальной работы программы и отрицательно сказаться на состоянии защиты системы.

Другие атрибуты файлов получают по запросу из списка, создаваемого при вызове метода `Files.readAttributes()`. В рассматриваемом здесь примере программы этот метод вызывается для получения связанного с файлом объекта типа `BasicFileAttributes`, но аналогичный общий подход можно применить и к другим типам атрибутов.

В приведенном ниже примере программы демонстрируется применение некоторых методов из интерфейса `Path` и класса `Files` наряду с методами из интерфейса `BasicFileAttributes`. В этой программе подразумевается, что файл `test.txt` находится в каталоге `examples`, входящем в текущий каталог.

```
// Получить сведения о пути к файлу и самом файле
// Требуется установка комплекта JDK, начиная с версии 7
```

```
import java.io.*;
```

```
import java.nio.file.*;
import java.nio.file.attribute.*;

class PathDemo {
    public static void main(String args[]) {
        Path filepath = Paths.get("examples\\test.txt");

        System.out.println("Имя файла: " + filepath.getName(1));
        System.out.println("Путь к файлу: " + filepath);
        System.out.println("Абсолютный путь к файлу: " +
            filepath.toAbsolutePath());
        System.out.println(
            "Родительский каталог: " + filepath.getParent());
        if(Files.exists(filepath))
            System.out.println("Файл существует");
        else
            System.out.println("Файл не существует");

        try {
            if(Files.isHidden(filepath))
                System.out.println("Файл скрыт");
            else
                System.out.println("Файл не скрыт");
        } catch(IOException e) {
            System.out.println("Ошибка ввода-вывода: " + e);
        }

        Files.isWritable(filepath);
        System.out.println("Файл доступен для записи");

        Files.isReadable(filepath);
        System.out.println("Файл доступен для чтения");

        try {
            BasicFileAttributes attribs =
                Files.readAttributes(filepath, BasicFileAttributes.class);

            if(attribs.isDirectory())
                System.out.println("Это каталог");
            else
                System.out.println("Это не каталог");

            if(attribs.isRegularFile())
                System.out.println("Это обычный файл");
            else
                System.out.println("Это не обычный файл");

            if(attribs.isSymbolicLink())
                System.out.println("Это символическая ссылка");
            else
                System.out.println("Это не символическая ссылка");

            System.out.println("Время последней модификации файла: " +
                attribs.lastModifiedTime());
            System.out.println("Размер файла: " + attribs.size() +
                " байтов");
        } catch(IOException e) {
            System.out.println("Ошибка чтения атрибутов: " + e);
        }
    }
}
```

Если запустить эту программу на выполнение из каталога MyDir, в котором имеется каталог examples, содержащий файл test.txt, то в конечном итоге будет выведен результат, аналогичный приведенному ниже. (Разумеется, размер файла и его временные характеристики будут иными.)

```
Имя файла: test.txt
Путь к файлу: examples\test.txt
Абсолютный путь к файлу: C:\MyDir\examples\test.txt
Родительский каталог: examples
Файл существует
Файл не скрыт
Файл доступен для записи
Файл доступен для чтения
Это не каталог
Это обычный файл
Это не символическая ссылка
Время последней модификации файла: 2014-01-01T18:20:46.380445Z
Размер файла: 18 байтов
```

Если вы пользуетесь компьютером с файловой системой FAT (т.е. файловой системой DOS), то попытайтесь применить методы, определенные в интерфейсе `DosFileAttributes`. А если вы пользуетесь системой, совместимой с POSIX, то попробуйте применить методы, определенные в интерфейсе `PosixFileAttributes`.

Получение содержимого каталога

Если путь описывает каталог, можно прочитать содержимое этого каталога, используя статические методы, определенные в классе `Files`. Для этого следует сначала получить поток ввода из каталога, вызвав метод `newDirectoryStream()` и передав ему объект типа `Path`, обозначающий каталог. Ниже приведена одна из форм метода `newDirectoryStream()`.

```
static DirectoryStream<Path> newDirectoryStream(Path путь_к_каталогу)  
throws IOException
```

Здесь параметр `путь_к_каталогу` инкапсулирует путь к конкретному каталогу. Этот метод возвращает объект типа `DirectoryStream<Path>`, применяемый для получения содержимого каталога. Он генерирует исключение типа `IOException` при возникновении ошибки ввода-вывода, а также исключение типа `NotDirectoryException` (его класс является производным от класса `IOException`), если указанный путь не приводит к каталогу. Кроме того, может быть сгенерировано исключение типа `SecurityException`, если доступ к каталогу запрещен.

Класс `DirectoryStream<Path>` реализует интерфейс `AutoCloseable`, поэтому объектом этого класса можно управлять в блоке оператора `try` с ресурсами. Этот класс реализует также интерфейс `Iterable<Path>`. Это означает, что содержимое каталога можно получить, перебрав содержимое объекта типа `DirectoryStream`. При переборе каждая запись каталога представлена экземпляром интерфейса `Path`. Простейший способ перебрать объект типа `DirectoryStream` — организовать цикл `for` в стиле `for each`. Следует, однако, иметь в виду, что итератор, реализуемый в классе `DirectoryStream<Path>`, может быть получен только один раз для каждого экземпляра. Следовательно, метод `iterator()` может быть вызван, а цикл `for` в стиле `for each` — выполнен только один раз.

В следующем примере программы выводится содержимое каталога MyDir:

```
// Вывести содержимое каталога. Требуется установка комплекта JDK,
// начиная с версии 7

import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;

class DirList {
    public static void main(String args[]) {
        String dirname = "\\MyDir";

        // получить и обработать поток ввода каталога
        // в блоке оператора try
        try ( DirectoryStream<Path> dirstrm =
            Files.newDirectoryStream(Paths.get(dirname)) )
        {
            System.out.println("Каталог " + dirname);

            // Класс DirectoryStream реализует интерфейс Iterable,
            // поэтому для вывода содержимого каталога можно
            // организовать цикл for в стиле for each
            for(Path entry : dirstrm) {
                BasicFileAttributes attrs =
                    Files.readAttributes(entry, BasicFileAttributes.class);

                if(attrs.isDirectory())
                    System.out.print("<DIR> ");
                else
                    System.out.print("      ");

                System.out.println(entry.getName(1));
            }
        } catch(InvalidPathException e) {
            System.out.println("Ошибка указания пути " + e);
        } catch(NotDirectoryException e) {
            System.out.println(dirname + " не является каталогом.");
        } catch (IOException e) {
            System.out.println("Ошибка ввода-вывода: " + e);
        }
    }
}
```

Эта программа выводит следующий результат:

```
Каталог \MyDir
DirList.class
DirList.java
<DIR> examples
Test.txt
```

Содержимое каталога можно отфильтровать двумя способами. Самый простой из них — воспользоваться следующей общей формой метода `newDirectoryStream()`:

```
static DirectoryStream<Path> newDirectoryStream(
    Path путь к каталогу, String шаблон)
    throws IOException
```

В этой форме получают только те файлы, имена которых совпадают с заданным *шаблоном*. В качестве параметра *шаблон* можно указать полное имя файла или маску. *Маска* — это символьная строка, определяющая глобальный или общий шаблон, с которым будет совпадать один или несколько файлов, и содержащая общеупотребительные метасимволы * и ?. Они соответствуют любому количеству символов и любому одиночному символу соответственно. Ниже приведены другие шаблоны для сопоставления с маской.

**	Совпадает с любым количеством различных символов в каталогах
[символы]	Совпадает с любым из указанных <i>символов</i> . Символы * и ? среди указанных <i>символов</i> будут рассматриваться как обычные символы, а не как метасимволы. Через дефис можно указать пределы сопоставления с шаблоном, например, [x-z]
{список_масок}	Совпадает с любой из масок, задаваемых в <i>списке_масок</i> через запятую

Метасимволы * и ? можно указать, используя последовательности символов * и \?, а для того чтобы указать знак \ — последовательность символов \\. Можете поэкспериментировать с маской, подставив ее в вызов метода `newDirectoryStream()` из предыдущего примера программы следующим образом:

```
Files.newDirectoryStream(Paths.get(dirname), "{Path,Dir}*.{java,class}")
```

Еще один способ отфильтровать каталог — воспользоваться приведенной ниже общей формой метода `newDirectoryStream()`.

```
static DirectoryStream<Path> newDirectoryStream(Path путь_к_каталогу,
        DirectoryStream.Filter<? super Path> фильтр_файлов)
        throws IOException
```

Здесь `DirectoryStream.Filter` — это интерфейс, в котором определяется следующий метод:

```
boolean accept(T элемент) throws IOException
```

В данном случае типом *T* будет `Path`. Если требуется включить указанный *элемент* в список, возвращается логическое значение `true`, а иначе — логическое значение `false`. Эта форма метода `newDirectoryStream()` предоставляет возможность отфильтровать каталог по другому критерию, кроме имени файла. В частности, каталог можно отфильтровать по размеру, дате создания, дате модификации или атрибуту.

Этот процесс демонстрируется в приведенном ниже примере программы. В этой программе перечисляются только те файлы, которые доступны для записи.

```
// Вывести только те файлы из каталога,
// которые доступны для записи
```

```
import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;
```

```
class DirList {
    public static void main(String args[]) {
        String dirname = "\\MyDir";
```

```
// создать фильтр, возвращающий логическое значение true
```

```

// только в отношении доступных для записи файлов
DirectoryStream.Filter<Path> how =
    new DirectoryStream.Filter<Path>() {
        public boolean accept(Path filename) throws IOException {
            if(Files.isWritable(filename)) return true;
            return false;
        }
    };

// получить и использовать поток ввода из каталога
// только доступных для записи файлов
try (DirectoryStream<Path> dirstrm =
    Files.newDirectoryStream(Paths.get(dirname), how) )
{
    System.out.println("Каталог " + dirname);

    for(Path entry : dirstrm) {
        BasicFileAttributes attribs =
            Files.readAttributes(entry, BasicFileAttributes.class);

        if(attribs.isDirectory())
            System.out.print("<DIR> ");
        else
            System.out.print("      ");

        System.out.println(entry.getName(1));
    }
} catch (InvalidPathException e) {
    System.out.println("Ошибка указания пути " + e);
} catch (NotDirectoryException e) {
    System.out.println(dirname + " не является каталогом.");
} catch (IOException e) {
    System.out.println("Ошибка ввода-вывода: " + e);
}
}
}

```

Обход дерева каталогов с помощью метода `walkFileTree()`

В предыдущих примерах получалось содержимое только одного каталога. Но иногда требуется получить список файлов из дерева каталогов. В прошлом решить подобную задачу было нелегко, но система ввода-вывода NIO.2 значительно упрощает ее решение, поскольку теперь из класса `Files` можно вызвать метод `walkFileTree()`, способный обработать дерево каталогов. Этот метод имеет две общие формы объявления. В примерах, приведенных в этой главе, употребляется следующая форма этого метода:

```

static Path walkFileTree(Path корень, FileVisitor<? extends Path> fv)
    throws IOException

```

Исходная точка обхода дерева каталогов передается в качестве параметра `корень`. Экземпляр интерфейса `FileVisitor` передается в качестве параметра `fv`. Реализация интерфейса `FileVisitor` определяет способ обхода дерева каталогов, а также позволяет обращаться к сведениям о каталоге. При возникновении ошибки ввода-вывода генерируется исключение типа `IOException`. Кроме того, может быть сгенерировано исключение типа `SecurityException`.

В интерфейсе `FileVisitor` определяется, каким образом посещаются файлы при обходе дерева каталогов. Этот обобщенный интерфейс объявляется следующим образом:

```
interface FileVisitor<T>
```

При вызове метода `walkFileTree()` в качестве параметра типа `T` указывается интерфейс `Path` (или любой производный от него тип). В интерфейсе `FileVisitor` определены методы, перечисленные в табл. 21.11.

Таблица 21.11. Методы из интерфейса `FileVisitor`

Метод	Описание
<code>FileVisitResult</code> <code>postVisitDirectory(T каталог, IOException исключение) throws IOException</code>	Вызывается после посещения каталога. Каталог передается в качестве параметра <i>каталог</i> , а любое исключение типа <code>IOException</code> — в качестве параметра <i>исключение</i> . Если параметр <i>исключение</i> принимает пустое значение <code>null</code> , то никакого исключения не происходит. Возвращает полученный результат
<code>FileVisitResult</code> <code>preVisitDirectory(T каталог, BasicFileAttributes атрибуты) throws IOException</code>	Вызывается перед посещением каталога. Каталог передается в качестве параметра <i>каталог</i> , а связанные с ним атрибуты — в качестве параметра <i>атрибуты</i> . Возвращает полученный результат. Чтобы исследовать каталог, следует вернуть значение <code>FileVisitResult.CONTINUE</code>
<code>FileVisitResult</code> <code>visitFile(T файл, BasicFileAttributes атрибуты) throws IOException</code>	Вызывается при посещении файла. Файл передается в качестве параметра <i>файл</i> , а связанные с ним атрибуты — в качестве параметра <i>атрибуты</i> . Возвращает полученный результат
<code>FileVisitResult</code> <code>visitFileFailed(T файл, IOException исключение) throws IOException</code>	Вызывается при неудачной попытке посетить файл. Файл, который не удалось посетить, передается в качестве параметра <i>файл</i> , а исключение типа <code>IOException</code> — в качестве параметра <i>исключение</i> . Возвращает полученный результат

Обратите внимание на то, что каждый метод возвращает значение из перечисления `FileVisitResult`. В этом перечислении определяются следующие значения:

CONTINUE	SKIP_SIBLINGS	SKIP_SUBTREE	TERMINATE
-----------------	----------------------	---------------------	------------------

В общем, для продолжения обхода каталога и находящихся в нем каталогов метод должен вернуть значение `CONTINUE`. Для того чтобы пропустить каталог и его содержимое, а также предотвратить вызов метода `postVisitDirectory()`, из метода `preVisitDirectory()` должно быть возвращено значение `SKIP_SIBLINGS`, чтобы пропустить только каталог и подкаталоги — значение `SKIP_SUBTREE`, а для того чтобы остановить обход каталога — значение `TERMINATE`.

Безусловно, можно создать собственный класс для обхода каталогов и реализовать в нем методы, определенные в интерфейсе `FileVisitor`, но обычно так не поступают, поскольку предоставляется простая их реализация в классе `SimpleFileVisitor`. В этом случае достаточно переопределить реализацию

по умолчанию одного или нескольких нужных методов. Ниже приведен краткий пример программы, демонстрирующий этот процесс. В этой программе выводятся все файлы из дерева каталогов, в корне которого находится каталог `\MyDir`. Обратите внимание на краткость этой программы.

```
// Простой пример применения метода walkFileTree()
// для вывода дерева каталогов. Требуется установка
// комплекта JDK, начиная с версии 7

import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;

// создать специальную версию класса SimpleFileVisitor,
// в которой переопределяется метод visitFile()
class MyFileVisitor extends SimpleFileVisitor<Path> {
    public FileVisitResult visitFile(
        Path path, BasicFileAttributes attribs)
        throws IOException
    {
        System.out.println(path);
        return FileVisitResult.CONTINUE;
    }
}

class DirTreeList {
    public static void main(String args[]) {
        String dirname = "\\MyDir";

        System.out.println("Дерево каталогов, начиная с каталога " +
            dirname + ":\n");
        try {
            Files.walkFileTree(Paths.get(dirname), new MyFileVisitor());
        } catch (IOException exc) {
            System.out.println("Ошибка ввода-вывода");
        }
    }
}
```

Ниже приведен примерный результат, выводимый данной программой для обхода того же каталога `MyDir`, что и прежде. В данном примере подкаталог `examples` содержит только один файл `MyProgram.java`.

Дерево каталогов, начиная с каталога `\MyDir`:

```
\MyDir\DirList.class
\MYDir\DirList.java
\MYDir\examples\MyProgram.java
\MYDir\Test.txt
```

В данной программе класс `MyFileVisitor` расширяет класс `SimpleFileVisitor`, переопределяя только метод `visitFile()`, который просто выводит их, хотя совсем не трудно достичь и более сложных функциональных возможностей. Например, можно было бы отфильтровать файлы или выполнить над ними такие действия, как копирование на резервное устройство. Ради простоты в данном примере для переопределения метода `visitFile()` выбран именованный класс, но вместо него ничто не мешает воспользоваться анонимным внутренним классом. И последнее замечание: средствами класса `java.nio.file.WatchService` можно отследить изменения в каталоге.

Примеры организации канального ввода-вывода до версии JDK 7

Прежде чем завершить эту главу, следует рассмотреть еще одну особенность системы ввода-вывода NIO. В приведенных выше примерах употреблялись некоторые из новых средств, внедренных в систему ввода-вывода NIO, начиная с версии JDK 7. Но ведь имеется еще немало кода, который написан до версии JDK 7 и требует надлежащего сопровождения, а возможно, и преобразования для применения новых средств. Поэтому в последующих разделах будет показано, как организовать чтение и запись в файлы средствами системы ввода-вывода NIO, доступными до версии JDK 7. Некоторые из приведенных выше примеров переделаны таким образом, чтобы использовать предыдущие средства системы ввода-вывода NIO, а не новые средства, поддерживаемые в системе ввода-вывода NIO.2. Это означает, что приведенные далее примеры пригодны для версий Java, выпущенных до версии JDK 7.

Главное отличие прежнего кода от нового, в котором применяется система ввода-вывода NIO, заключается в интерфейсе `Path`, который был внедрен в версии JDK 7. Следовательно, для описания файла или открытия канала к нему в прежнем коде не применяется интерфейс `Path`. Кроме того, в прежнем коде не применяются операторы `try` с ресурсами, поскольку автоматическое управление ресурсами также было внедрено только в версии JDK 7.

Помните! В примерах программ из этого раздела описывается действие унаследованного кода, в котором применяется система ввода-вывода NIO. Материал этого раздела предназначен для тех программистов, которые продолжают работать с унаследованным кодом или пользуются компилятором, выпущенным до версии JDK 7. В новом коде должны применяться средства системы ввода-вывода NIO, внедренные в версии JDK 7.

Чтение из файла до версии JDK 7

В этом разделе рассматриваются два предыдущих примера канального ввода из файла, переделанных для использования средств системы ввода-вывода NIO, доступных только до версии JDK 7. В первом примере для чтения данных из файла сначала выделяется вручную оперативная память под буфер, а затем выполняется явным образом операция чтения. Во втором примере производится сопоставление файла с буфером, автоматизирующее весь процесс чтения данных из файла.

Если для чтения данных из файла через канал и выделяемый вручную буфер используются версии Java, выпущенные до версии JDK 7, то сначала файл открывается с помощью потока ввода типа `FileInputStream` таким же способом, как описано в главе 20. Затем вызывается метод `getChannel()` для потока ввода типа `FileInputStream`, чтобы получить канал к открытому файлу. Ниже приведена общая форма этого метода.

```
FileChannel getChannel()
```

В этой форме возвращается объект типа `FileChannel`, инкапсулирующий канал для файловых операций. Далее вызывается метод `allocate()` для выделения оперативной памяти под буфер. Файловые каналы оперируют буферами байтов, и поэтому для их получения в примерах этой главы вызывается метод `allocate()`, определенный в классе `ByteBuffer`, как пояснялось ранее.

В следующем примере программы демонстрируется чтение и вывод содержимого файла `test.txt` через канал с использованием явных операций ввода для версий Java, выпущенных до JDK 7:

```
// Использовать каналы для чтения данных из файла.
// Версия до JDK 7
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class ExplicitChannelRead {
    public static void main(String args[]) {
        FileInputStream fin = null;
        FileChannel fChan = null;
        ByteBuffer mBuf;
        int count;

        try {
            // сначала открыть файл для ввода
            fin = new FileInputStream("test.txt");

            // затем получить канал к этому файлу
            fChan = fin.getChannel();

            // выделить оперативную память под буфер
            mBuf = ByteBuffer.allocate(128);

            do {
                // читать данные в буфер
                count = fChan.read(mBuf);

                // прекратить чтение по достижении конца файла
                if(count != -1) {

                    // подготовить буфер к чтению из него данных
                    mBuf.rewind();

                    // читать байты данных из буфера и
                    // выводить их на экран как символы
                    for(int i=0; i < count; i++)
                        System.out.print((char)mBuf.get());

                }
            } while(count != -1);

            System.out.println();

        } catch (IOException e) {
            System.out.println("Ошибка ввода-вывода " + e);
        } finally {
            try {
                if(fChan != null) fChan.close(); // закрыть канал
            } catch(IOException e) {
                System.out.println("Ошибка закрытия канала.");
            }
        }
    }
}
```

```

    }
    try {
        if(fIn != null) fIn.close(); // закрыть файл
    } catch(IOException e) {
        System.out.println("Ошибка закрытия файла.");
    }
}
}
}

```

Обратите внимание на то, что в данной программе файл открывается с помощью конструктора класса `FileInputStream`, а ссылка на создаваемый объект этого класса присваивается переменной экземпляра `fIn`. А когда вызывается метод `getChannel()` по ссылке на этот объект в переменной экземпляра `fIn`, то создается канал, подключаемый к открытому файлу. После этого программа работает таким же образом, как и в рассмотренной ранее версии для JDK 7. Напомним вкратце, что сначала вызывается метод `allocate()` из класса `ByteBuffer`, чтобы выделить оперативную память под буфер для хранения содержимого, прочитанного из файла. Для этого создается буфер байтов, поскольку класс `FileChannel` оперирует байтами. Ссылка на этот буфер хранится в переменной экземпляра `mBuf`. Затем содержимое файла читается по очереди в буфер `mBuf` методом `read()`. Количество прочитанных байтов хранится в переменной `count`. Далее вызывается метод `rewind()`, чтобы подготовить буфер к чтению данных. Этот метод приходится вызывать потому, что после вызова метода `read()` текущая позиция находится в конце буфера, а ее следует возратить в начало буфера, чтобы все байты данных могли быть прочитаны из буфера `mBuf` при вызове метода `get()`. По достижении конца файла метод `read()` возвратит значение `-1`. Когда это произойдет, программа завершится, закрыв канал и файл явным образом.

Другой способ чтения данных из файла состоит в его сопоставлении с буфером. Как пояснялось ранее, основное преимущество такого подхода заключается в том, что буфер автоматически получает содержимое файла. Никаких явных операций чтения для этого не требуется. Чтобы сопоставить и прочитать содержимое файла, используя средства системы ввода-вывода NIO, доступные до версии JDK 7, следует открыть сначала файл с помощью потока ввода типа `FileInputStream`, а затем получить канал к этому файлу, вызвав метод `getChannel()` для файлового объекта. После этого вызывается метод `map()` для объекта типа `FileChannel`, чтобы сопоставить полученный канал с буфером, как описано ранее.

Ниже приведена версия программы из предыдущего примера, переделанная таким образом, чтобы для сопоставления файла с буфером использовались только средства системы ввода-вывода NIO, доступные до версии JDK 7.

```

// Использовать сопоставление для чтения данных из файла.
// Версия до JDK 7

```

```

import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MappedChannelRead {
    public static void main(String args[]) {
        FileInputStream fIn = null;
    }
}

```

```

FileChannel fChan = null;
long fSize;
MappedByteBuffer mBuf;

try {
    // сначала открыть файл для ввода
    fIn = new FileInputStream("test.txt");

    // затем получить канал к этому файлу
    fChan = fIn.getChannel();

    // получить размер файла
    fSize = fChan.size();

    // а теперь сопоставить файл с буфером
    mBuf = fChan.map(FileChannel.MapMode.READ_ONLY, 0, fSize);

    // читать байты из буфера и выводить их на экран
    for(int i=0; i < fSize; i++)
        System.out.print((char)mBuf.get());
} catch (IOException e) {
    System.out.println("Ошибка ввода-вывода " + e);
} finally {
    try {
        if(fChan != null) fChan.close(); // закрыть канал
    } catch(IOException e) {
        System.out.println("Ошибка закрытия канала.");
    }
    try {
        if(fIn != null) fIn.close(); // закрыть файл
    } catch(IOException e) {
        System.out.println("Ошибка закрытия файла.");
    }
}
}
}

```

В данной программе файл открывается с помощью конструктора класса `FileInputStream`, а ссылка на создаваемый объект этого класса присваивается переменной экземпляра `fIn`. Затем вызывается метод `getChannel()` по ссылке на этот объект в переменной экземпляра `fIn`, чтобы создать канал, подключаемый к открытому файлу. Далее определяется размер файла. После этого вызывается метод `map()`, чтобы сопоставить весь файл с областью в памяти, выделяемой под буфер, а ссылка на этот буфер сохраняется в переменной экземпляра `mBuf`. И наконец, вызывается метод `get()`, чтобы прочитать байты из буфера `mBuf`.

Запись в файл до версии JDK 7

В этом разделе представлены два предыдущих примера канального вывода данных в файл, переделанные таким образом, чтобы использовать только средства системы ввода-вывода NIO, доступные до версии JDK 7. В первом примере для записи данных в файл сначала выделяется вручную оперативная память под буфер, а затем выполняется явным образом операция записи. А во втором примере производится сопоставление файла с буфером, автоматизирующее весь

процесс записи данных в файл. Но в обоих случаях ни интерфейс `Path`, ни оператор `try` с ресурсами не применяются, поскольку они доступны лишь с версии JDK 7.

Если для записи данных в файл через канал и выделяемый вручную буфер используются версии Java, выпущенные до версии JDK 7, то сначала файл открывается с помощью потока вывода типа `FileOutputStream` таким же способом, как описано в главе 20. Затем вызывается метод `getChannel()` для потока вывода типа `FileOutputStream`, чтобы получить канал к открытому файлу, а после этого — метод `allocate()`, чтобы выделить оперативную память под буфер, как описано в предыдущем разделе. Далее записываемые в файл данные размещаются в этом буфере и вызывается метод `write()` для открытого канала. Вся эта процедура демонстрируется в приведенном ниже примере программы. В этой программе английский алфавит записывается в файл `test.txt`.

```
// Записать данные в файл средствами системы ввода-вывода NIO.
// Версия до JDK 7.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class ExplicitChannelWrite {
    public static void main(String args[]) {
        FileOutputStream fOut = null;
        FileChannel fChan = null;
        ByteBuffer mBuf;

        try {
            // сначала открыть файл для вывода данных
            fOut = new FileOutputStream("test.txt");

            // затем получить канал к файлу для вывода данных
            fChan = fOut.getChannel();

            // создать буфер
            mBuf = ByteBuffer.allocate(26);

            // записать некоторое количество байтов в буфер
            for(int i=0; i<26; i++)
                mBuf.put((byte)('A' + i));

            // подготовить буфер к записи данных
            mBuf.rewind();

            // записать данные из буфера в выходной файл
            fChan.write(mBuf);

        } catch (IOException e) {
            System.out.println("Ошибка ввода-вывода " + e);
        } finally {
            try {
                if(fChan != null) fChan.close(); // закрыть канал
            } catch (IOException e) {
                System.out.println("Ошибка закрытия канала.");
            }
        }
        try {
            if(fOut != null) fOut.close(); // закрыть файл
        }
```

```

        } catch(IOException e) {
            System.out.println("Ошибка закрытия файла.");
        }
    }
}

```

Вызов метода `rewind()` по ссылке на буфер в переменной экземпляра `mBuf` требуется для возврата текущей позиции в начало буфера `mBuf` после записи данных. Напомним, что при каждом вызове метода `put()` текущая позиция продвигается к концу буфера. Поэтому, прежде чем вызвать метод `write()`, следует установить текущую позицию в начало буфера. В противном случае метод `write()` не сумеет обнаружить в буфере никаких данных, посчитав, что их там вообще нет.

Если для записи данных в файл производится сопоставление файла с буфером и для этой цели используются версии Java, выпущенные до версии JDK 7, то сначала для выполнения операций чтения и записи создается объект класса `RandomAccessFile`, чтобы открыть файл. Для открываемого файла требуется разрешение на чтение и запись. Затем для данного объекта вызывается метод `map()`, чтобы сопоставить открытый файл с буфером. Далее записываемые данные размещаются в буфере. А поскольку буфер сопоставлен с файлом, то любые изменения в нем автоматически отражаются в файле. Таким образом, никаких явных операций записи в канал не требуется.

Ниже приведена версия программы из предыдущего примера, переделанная для сопоставления файла с буфером.

```

// Записать данные в сопоставленный файл. Версия до JDK 7.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MappedChannelWrite {
    public static void main(String args[]) {
        RandomAccessFile fOut = null;
        FileChannel fChan = null;
        ByteBuffer mBuf;

        try {
            fOut = new RandomAccessFile("test.txt", "rw");

            // получить канал к открытому файлу
            fChan = fOut.getChannel();

            // затем сопоставить файл с буфером
            mBuf = fChan.map(FileChannel.MapMode.READ_WRITE, 0, 26);

            // записать некоторое количество байтов в буфер
            for(int i=0; i<26; i++)
                mBuf.put((byte)('A' + i));

        } catch (IOException e) {
            System.out.println("Ошибка ввода-вывода " + e);
        } finally {
            try {
                if(fChan != null) fChan.close(); // закрыть канал
            } catch(IOException e) {

```



```
        System.out.println("Ошибка закрытия канала.");
    }
    try {
        if(fOut != null) fOut.close(); // закрыть файл
    } catch(IOException e) {
        System.out.println("Ошибка закрытия файла.");
    }
}
}
```

Как видите, в данном примере нет никаких явных операций записи непосредственно в канал. Благодаря тому что буфер `mBuf` сопоставляется с файлом, изменения в буфере автоматически отражаются в базовом файле.

Как известно, язык Java служит практически синонимом программирования для Интернета. На то имеется немало причин, и далеко не самой последней из них является способность создавать безопасный код, переносимый между платформами. Но одна из наиболее важных причин, по которым язык Java отлично подходит для сетевого программирования, кроется в классах, определенных в пакете `java.net`. Эти классы обеспечивают простые в употреблении средства, с помощью которых программисты всех уровней квалификации могут обращаться к сетевым ресурсам.

Эта глава посвящена пакету `java.net`. Следует особо подчеркнуть, что работа в сети — очень обширная и сложная тема. В данной книге недостаточно места, чтобы полностью описать все средства, входящие в пакет `java.net`. Поэтому в этой главе основное внимание уделяется лишь самым основным классам и интерфейсам для работы в сети.

Основы работы в сети

Прежде всего полезно дать хотя бы самое общее представление о ключевых понятиях и терминах, связанных с работой в сети. В основу работы в сети, поддерживаемой в Java, положено понятие *сокета*, обозначающего конечную точку в сети. Понятие сокета стало употребляться в ОС UNIX, начиная с версии 4.2 BSD Berkley еще в начале 1980-х годов. По этой причине употребляется также термин *сокет Беркли*. Сокеты составляют основу современных способов работы в сети, поскольку сокет позволяет отдельному компьютеру одновременно обслуживать много разных клиентов, предоставляя разные виды информации. Эта цель достигается благодаря применению *порта* — нумерованного сокета на отдельной машине. Говорят, что серверный процесс “прослушивает” порт до тех пор, пока клиент не соединится с ним. Сервер в состоянии принять запросы от многих клиентов, подключаемых к порту с одним и тем же номером, хотя каждый сеанс связи индивидуален. Для управления соединениями со многими клиентами серверный процесс должен быть многопоточным или располагать какими-то другими средствами для мультиплексирования одновременного ввода-вывода.

Связь между сокетами устанавливается и поддерживается по определенному сетевому протоколу. *Протокол Интернета* (IP) является низкоуровневым маршру-

тизирующим сетевым протоколом, разбивающим данные на небольшие пакеты и посылающим их через сеть по определенному адресу, что не гарантирует доставки всех этих пакетов по этому адресу. *Протокол управления передачей* (TCP) является сетевым протоколом более высокого уровня, обеспечивающим связывание, сортировку и повторную передачу пакетов, чтобы обеспечить надежную доставку данных. Еще одним сетевым протоколом более низкого уровня, чем TCP, является *протокол пользовательских дейтаграмм* (UDP). Этот сетевой протокол может быть использован непосредственно для поддержки быстрой, не требующей постоянного соединения и ненадежной транспортировки пакетов.

Как только соединение будет установлено, в действие вступает высокоуровневый протокол, тип которого зависит от используемого порта. Протокол TCP/IP резервирует первые 1024 порта для отдельных протоколов. Многие из них покажутся вам знакомыми, если вам довелось потратить хотя бы немного времени на блуждание по Интернету. В частности, порт 21 выделен для протокола FTP, порт 23 — для протокола Telnet, порт 25 — для электронной почты, порт 43 — для протокола whois, порт 80 — для протокола HTTP, порт 119 — для протокола netnews и т.д. Каждый сетевой протокол определяет порядок взаимодействия клиента с портом.

Например, протокол HTTP используется серверами и веб-браузерами для передачи гипертекста и графических изображений. Это довольно простой протокол для базового страничного просмотра информации, предоставляемой веб-серверами. Рассмотрим принцип его действия. Когда клиент запрашивает файл у HTTP-сервера, это действие называется *обращением* и состоит в том, чтобы отправить имя файла в специальном формате в предопределенный порт и затем прочитать содержимое этого файла. Сервер также сообщает код состояния, чтобы известить клиента, был ли запрос обслужен, а также причину, по которой он не может быть обслужен.

Главной составляющей Интернета является *адрес*. Каждый компьютер в Интернете обладает своим адресом. Адрес Интернета представляет собой число, однозначно обозначающее каждый компьютер в Интернете. Изначально все адреса Интернета состояли из 32-разрядных значений, организованных по четыре 8-разрядных значения. Адрес такого типа определен в протоколе IPv4 (Протокол Интернета версии 4). Но в последнее время вступила в действие новая схема адресации, называемая IPv6 и предназначенная для поддержки намного большего адресного пространства. Правда, для сетевого программирования на Java обычно не приходится беспокоиться, какого типа адрес используется: IPv4 или IPv6, поскольку эта задача решается в Java автоматически.

Подобно тому, как IP-адрес описывает сетевую иерархию, имя адреса Интернета, называемое *доменным именем*, обозначает местонахождение машины в пространстве имен. Например, адрес `www.HerbSchildt.com` относится к верхнему домену `com`, зарезервированному для коммерческих веб-сайтов в США и имеющему имя `HerbSchildt` (по названию компании), а префикс `www` обозначает веб-сервер, обрабатывающий запросы. Доменное имя Интернета сопоставляется с IP-адресом с помощью *службы доменных имен* (Domain Name Service — DNS). Это дает пользователям возможность обращаться с доменными именами, тогда как Интернет оперирует IP-адресами.

Сетевые классы и интерфейсы

Сетевой протокол TCP/IP поддерживается в Java благодаря расширению уже имеющихся интерфейсов потокового ввода-вывода, представленных в главе 20, а также внедрению новых средств, необходимых для построения объектов ввода-вывода через сеть. В Java поддерживаются оба семейства протоколов — TCP и UDP. Протокол TCP применяется для надежного потокового ввода-вывода через сеть. А протокол UDP поддерживает более простую, а следовательно, и быструю модель передачи дейтаграмм от одной точки сети к другой. Ниже перечислены классы, входящие в пакет `java.net`.

Authenticator	InetAddress	SocketAddress
CacheRequest	InetSocketAddress	SocketImpl
CacheResponse	InterfaceAddress	SocketPermission
ContentHandler	JarURLConnection	StandardSocketOption
CookieHandler	MulticastSocket	URI
CookieManager	NetPermission	URL
DatagramPacket	NetworkInterface	URLClassLoader
DatagramSocket	PasswordAuthentication	URLConnection
DatagramSocketImpl	Proxy	URLDecoder
HttpCookie	ProxySelector	URLEncoder
HttpURLConnection	ResponseCache	URLPermission (добавлен в версии JDK 8)
IDN	SecureCacheResponse	URLStreamHandler
Inet4Address	ServerSocket	
Inet6Address	Socket	

Ниже перечислены интерфейсы из пакета `java.net`.

ContentHandlerFactory	FileNameMap	SocketOptions
CookiePolicy	ProtocolFamily	URLStreamHandlerFactory
CookieStore	SocketImplFactory	
DatagramSocketImplFactory	SocketOption	

В последующих разделах рассматриваются основные сетевые классы и представлен ряд примеров их применения. Как только вам станет понятно внутреннее устройство сетевых классов, вы сможете строить на их основе свои классы.

Класс `InetAddress`

Этот класс служит для инкапсуляции как числового IP-адреса, так и его доменного имени. Для взаимодействия с этим классом используется имя IP-хоста, т.е. узла сети, которое намного удобнее и понятнее, чем IP-адрес. Числовое значение IP-адреса скрывается в классе `InetAddress`. Этот класс может оперировать адресами как по протоколу IPv4, так и по протоколу IPv6.

Фабричные методы

В классе `InetAddress` отсутствуют доступные конструкторы. Чтобы создать объект класса `InetAddress`, следует использовать один из доступных в нем фабричных методов. *Фабричные методы* просто обозначают соглашение, по которому статические методы класса возвращают экземпляр этого класса. Это делается вместо перегрузки конструктора с различными списками параметров, когда наличие однозначных имен методов проясняет результат. Ниже приведены общие формы трех наиболее употребительных фабричных методов из класса `InetAddress`.

```
static InetAddress getLocalHost()
    throws UnknownHostException
static InetAddress getByName(String имя_хоста)
    throws UnknownHostException
static InetAddress[] getAllByName(String имя_хоста)
    throws UnknownHostException
```

Метод `getLocalHost()` возвращает объект типа `InetAddress`, представляющий локальный хост, а метод `getByName()` — объект типа `InetAddress`, представляющий хост, имя которого передается в качестве параметра *имя_хоста*. Если эти методы оказываются не в состоянии получить имя хоста, они генерируют исключение типа `UnknownHostException`.

В Интернете считается обычным явлением, когда одно имя используется для обозначения нескольких машин. Это единственный путь обеспечить в какой-то степени масштабируемость веб-серверов. Фабричный метод `getAllByName()` возвращает массив объектов типа `InetAddress`, представляющих все адреса, в которые преобразуется конкретное имя. Этот метод генерирует также исключение типа `UnknownHostException` в том случае, если он не в состоянии преобразовать имя хотя бы в один адрес.

В состав класса `InetAddress` входит также фабричный метод `getDyAddress()`, который принимает IP-адрес и возвращает объект типа `InetAddress`. Причем могут использоваться адреса как по протоколу IPv4, так и по протоколу IPv6.

В следующем примере программы выводятся адреса и имена локальной машины, а также двух веб-сайтов в Интернете:

```
// Продемонстрировать применение класса InetAddress
import java.net.*;

class InetAddressTest
{
    public static void main(String args[]) throws UnknownHostException {
        InetAddress Address = InetAddress.getLocalHost();
        System.out.println(Address);

        Address = InetAddress.getByName("www.HerbSchildt.com");
        System.out.println(Address);

        InetAddress SW[] = InetAddress.getAllByName("www.nba.com");
        for (int i=0; i<SW.length; i++)
            System.out.println(SW[i]);
    }
}
```

Ниже приведен результат, выводимый этой программой. Безусловно, код, который вы увидите на своей машине, может несколько отличаться от приведенного ниже.

```
default/166.203.115.212
www.HerbSchildt.com/216.92.65.4
www.nba.com/216.66.31.161
www.nba.com/216.66.31.179
```

Методы экземпляра

В классе `InetAddress` имеется также ряд других методов, которые могут вызываться для объектов, возвращаемых упомянутыми выше методами. Некоторые из наиболее употребительных методов перечислены в табл. 22.1.

Таблица 22.1. Наиболее употребительные методы из класса `InetAddress`

Метод	Описание
<code>boolean equals(Object <i>другое</i>)</code>	Возвращает логическое значение true , если объект имеет тот же адрес Интернета, что и объект, обозначаемый параметром <i>другое</i>
<code>byte[] getAddress()</code>	Возвращает массив байтов, представляющий IP-адрес в порядке следования байтов в сети
<code>String getHostAddress()</code>	Возвращает символьную строку, представляющую адрес хоста, связанного с объектом типа InetAddress
<code>String getHostName()</code>	Возвращает символьную строку, представляющую имя хоста, связанного с объектом типа InetAddress
<code>boolean isMulticastAddress()</code>	Возвращает логическое значение true , если адрес является групповым, а иначе — возвращает логическое значение false
<code>String toString()</code>	Возвращает символьную строку, перечисляющую для удобства имя и IP-адрес хоста

Поиск адресов Интернета осуществляется в последовательном ряде иерархически кешированных серверов. Это означает, что локальный компьютер может автоматически сопоставить конкретное имя с его IP-адресом, как в отношении себя, так и в отношении ближайших серверов. А в отношении всех прочих имен он может обращаться к DNS-серверам, откуда получают сведения об IP-адресах. Если на таком сервере отсутствуют сведения об определенном адресе, он может обратиться к следующему удаленному сайту и запросить у него эти сведения. Этот процесс может продолжаться вплоть до корневого сервера и потребовать немало времени. Поэтому структуру прикладного кода следует построить таким образом, чтобы сведения об IP-адресах локально кешировались и их не приходилось искать каждый раз заново.

Классы `Inet4Address` и `Inet6Address`

В состав Java включена поддержка адресов как по протоколу IPv4, так и по протоколу IPv6. В связи с этим были созданы следующие два подкласса, производных от класса `InetAddress`: `Inet4Address` и `Inet6Address`. Класс `Inet4Address`

представляет традиционные адреса по протоколу IPv4, а класс `Inet6Address` инкапсулирует адреса нового стиля по протоколу IPv6. А поскольку оба эти класса являются производными от класса `InetAddress`, то ссылки на класс `InetAddress` могут указывать и на них. Это единственный способ, с помощью которого удастся внедрить в Java функциональные возможности протокола IPv6, не нарушая существующий код и не вводя большое количество новых классов. Как правило, класс `InetAddress` можно применять напрямую, чтобы оперировать IP-адресами, поскольку этот класс приспособлен для обеих разновидностей адресов.

Клиентские сокеты по протоколу TCP/IP

Сокеты по протоколу TCP/IP служат для реализации надежных двунаправленных, постоянных, двухточечных, потоковых соединений между хостами в Интернете. Сокет может служить для подключения системы ввода-вывода в Java к другим программам, которые могут находиться как на локальной машине, так и на любой другой машине в Интернете.

На заметку! Как правило, апплеты могут устанавливать сокетные соединения только с тем хостом, с которого они были загружены. Это ограничение введено в связи с тем, что апплетам, загружаемым через брандмауэр, было бы опасно предоставлять доступ к любой произвольной машине.

В Java поддерживаются две разновидности сокетов по протоколу TCP/IP: один — для серверов, другой — для клиентов. Класс `ServerSocket` служит “приемником”, ожидая подключения клиентов прежде, чем предпринять какие-нибудь действия. Иными словами, класс `ServerSocket` предназначен для серверов, тогда как класс `Socket` — для клиентов. Он служит для подключения к серверным сокетам и инициирования обмена данными по сетевому протоколу. Клиентские сокеты чаще всего применяются в прикладных программах на Java, поэтому они здесь и рассматриваются.

При создании объекта типа `Socket` неявно устанавливается соединение клиента с сервером. Выявить это соединение нельзя никакими методами или конструкторами. В табл. 22.2 перечислены два конструктора класса `Socket`, предназначенные для создания клиентских сокетов.

Таблица 22.2. Конструкторы класса `Socket`

Конструктор	Описание
<code>Socket(String имя_хоста, int порт) throws UnknownHostException, IOException</code>	Создает сокет, подключаемый к указанному имени_хоста и порту
<code>Socket(InetAddress IP-адрес, int порт) throws IOException</code>	Создает сокет, используя уже существующий объект типа <code>InetAddress</code> и указанный порт

В классе `Socket` определяется ряд методов экземпляра. Например, объект типа `Socket` может быть просмотрен в любой момент для извлечения сведений о связанных с ним адресе и порте. Для этого применяются методы, перечисленные в табл. 22.3.

Таблица 22.3. Методы из класса **Socket**

Метод	Описание
InetAddress getInetAddress()	Возвращает объект типа InetAddress , связанный с объектом типа Socket . Если же сокет не подключен, возвращается пустое значение null
int getPort()	Возвращает удаленный порт, к которому привязан вызывающий объект типа Socket . Если же сокет не привязан, возвращается нулевое значение
int getLocalPort()	Возвращает локальный порт, к которому привязан вызывающий объект типа Socket . Если же сокет не привязан, возвращается значение -1

Для доступа к потокам ввода-вывода, связанным с классом **Socket**, можно воспользоваться методами **getInputStream()** и **getOutputStream()**, перечисленными в табл. 22.4. Каждый из этих методов может сгенерировать исключение типа **IOException**, если сокет оказался недействительным из-за потери соединения. Эти потоки ввода-вывода используются для передачи и приема данных таким же образом, как и потоки ввода-вывода, рассмотренные в главе 20.

Таблица 22.4. Методы доступа к потокам ввода-вывода

Метод	Описание
InputStream getInputStream() throws IOException	Возвращает объект типа InetAddress , связанный с вызывающим сокетом
OutputStream getOutputStream() throws IOException	Возвращает объект типа OutputStream , связанный с вызывающим сокетом

Имеется и ряд других методов, в том числе метод **connect()**, позволяющий указать новое соединение; метод **isConnected()**, возвращающий логическое значение **true**, если сокет подключен к серверу; метод **isBound()**, возвращающий логическое значение **true**, если сокет привязан к адресу; а также метод **isClosed()**, возвращающий логическое значение **true**, если сокет закрыт. Чтобы закрыть сокет, достаточно вызвать метод **close()**. Закрытие сокета приводит также к закрытию связанных с ним потоков ввода-вывода. Начиная с версии **JDK 7** класс **Socket** реализует также интерфейс **AutoCloseable**. Это означает, что управление сокетом можно организовать в блоке оператора **try** с ресурсами.

Приведенная ниже программа служит простым примером применения класса **Socket**. В этой программе устанавливается соединение с портом "whois" (номер 43) на **InterNIC**-сервере, посылается аргумент командной строки через сокет, а затем выводятся возвращаемые данные. **InterNIC**-сервер пытается интерпретировать аргумент как зарегистрированное доменное имя Интернета, а затем возвращает IP-адрес и контактную информацию из веб-сайта, найденного по этому доменному имени.

```
// Продемонстрировать обращение с сокетами
import java.net.*;
import java.io.*;

class Whois {
```



```

public static void main(String args[]) throws Exception {
    int c;

    // создать сокетное соединение с веб-сайтом internic.net
    // через порт 43
    Socket s = new Socket("whois.internic.net", 43);

    // получить потоки ввода-вывода
    InputStream in = s.getInputStream();
    OutputStream out = s.getOutputStream();

    // сформировать строку запроса
    String str = (args.length == 0 ? "MHProfessional.com" : args[0])
        + "\n";

    // преобразовать строку в байты
    byte buf[] = str.getBytes();

    // послать запрос
    out.write(buf);

    // прочитать ответ и вывести его на экран
    while ((c = in.read()) != -1) {
        System.out.print((char) c);
    }
    s.close();
}
}

```

Если запросить, например, сведения об адресе MHProfessional.com, то будет получен результат, аналогичный следующему:

Whois Server Version 2.0

Domain names in the .com and .net domains can now be registered with many different competing registrars. Go to <http://www.internic.net> for detailed information.

```

Domain Name: MHPROFESSIONAL.COM
Registrar: MELBOURNE IT, LTD. D/B/A INTERNET NAMES WORLDWIDE
Whois Server: whois.melbourneit.com
Referral URL: http://www.melbourneit.com
Name Server: NS1.MHEDU.COM
Name Server: NS2.MHEDU.COM

```

Эта программа действует следующим образом. Сначала в ней создается объект типа Socket, обозначающий сокет и задающий имя хоста "whois.internic.net" и номер порта 43 (internic.net — это сайт веб-службы InterNIC, обрабатывающей запросы по протоколу whois; а порт 43 предназначен именно для этой службы). Затем в сокете открываются потоки ввода-вывода. Далее формируется символьная строка, содержащая имя веб-сайта, сведения о котором требуется получить. Если веб-сайт не указан в командной строке, то выбирается имя хоста "MHProfessional.com". Эта символьная строка преобразуется в массив байтов и направляется в сеть через сокет. После этого ответ читается из сокета, а результат выводится на экран. И наконец, сокет закрывается, а вместе с ним и потоки ввода-вывода. В данном примере сокет закрывается вручную в результате вызова

метода `close()`. Если же используется комплект версии JDK 7, то для автоматического закрытия сокета можно организовать блок оператора `try` с ресурсами. В качестве примера ниже приведен другой способ написать метод `main()` из рассматриваемой здесь программы, чтобы закрывать сокет автоматически.

```
// Использовать блок оператора try с ресурсами для закрытия сокета
public static void main(String args[]) throws Exception {
    int c;

    // создать сокетное соединение с веб-сайтом internic.net
    // через порт 43. Этим сокетом управляет блок оператора
    // try с ресурсами
    try ( Socket s = new Socket("whois.internic.net", 43) ) {

        // получить потоки ввода-вывода
        InputStream in = s.getInputStream();
        OutputStream out = s.getOutputStream();

        // сформировать строку запроса
        String str = (args.length == 0 ? "MHProfessional.com" : args[0])
            + "\n";
        // преобразовать строку в байты
        byte buf[] = str.getBytes();

        // послать запрос
        out.write(buf);

        // прочитать ответ и вывести его на экран
        while ((c = in.read()) != -1) {
            System.out.print((char) c);
        }
    }

    // Теперь сокет закрыт
}
```

В приведенных далее примерах для демонстрации явного закрытия сетевого ресурса и совместимости кода этих примеров с версиями Java, выпущенными до версии JDK 7, будет вызываться метод `close()`. Но в своем прикладном коде следует уделить внимание автоматическому управлению сетевыми ресурсами, поскольку это более рациональный и гибкий подход. Следует также иметь в виду, что в данной версии программы исключения все еще генерируются в теле метода `main()`, но их можно было бы перехватывать и обрабатывать, добавив операторы `catch` в конце блока оператора `try` с ресурсами.

На заметку! Ради простоты в примерах из этой главы все исключения генерируются в теле метода `main()`. Это позволяет яснее показать логику выполнения сетевого кода. Но в реальном коде исключения должны обрабатываться надлежащим образом.

Класс URL

Предыдущий пример не совсем нагляден, поскольку в настоящее время Интернет не ассоциируется с такими старыми протоколами, как `whois`, `finger` или `FTP`. Здесь царствует Всемирная паутина (World Wide Web), или просто веб, — сла-

бо связанная совокупность высокоуровневых сетевых протоколов и форматов файлов, унифицированным образом применяемых в веб-браузерах. Одна из наиболее важных особенностей веб состоит в том, что ее создатель Тим Бернерс-Ли (Tim Berners-Lea) предложил масштабируемый способ нахождения всех ресурсов в Интернете. Как только удастся однозначно и надежно что-нибудь именовать, это становится очень эффективным принципом. Именно это и делает URL — *унифицированный указатель ресурса*.

URL обеспечивает довольно ясную форму однозначной идентификации адресной информации в веб, и для этой цели они широко применяются практически во всех браузерах. Класс URL из библиотеки сетевых классов в Java предоставляет простой и краткий прикладной программный интерфейс API для доступа к информации в Интернете с помощью URL.

Все URL совместно используют один и тот же основной формат, хотя и с некоторыми возможными вариациями. Рассмотрим следующие два примера URL: `http://www.MHProfessional.com/` и `http://www.MHProfessional.com:80/index.htm`. Спецификация URL основывается на четырех составляющих. Первая составляющая обозначает используемый сетевой протокол, отделяемый двоеточием (:) от остальной части URL. К числу наиболее распространенных сетевых протоколов относятся HTTP, FTP, gopher и file, хотя ныне почти весь обмен данными через Интернет осуществляется по протоколу HTTP (на самом деле большинство браузеров правильно интерпретируют URL, даже если исключить из его спецификации составляющую сетевого протокола "http://"). Вторая составляющая обозначает имя хоста или IP-адрес, используемый хостом. Она отделяется слева двойным знаком косой черты (//), а справа — одним знаком косой черты (/) или двоеточием (:), хотя это и необязательно. Третья составляющая обозначает номер порта. Это необязательный параметр, отделяемый слева от имени хоста двоеточием, а справа — одним знаком косой черты. (Так, если порт 80 выбирается для сетевого протокола HTTP по умолчанию, то указывать параметр ": 80" в URL излишне.) Четвертая составляющая обозначает действительный путь к файлу. Большинство HTTP-серверов присоединяют имя файла `index.html` или `index.htm` к URL для непосредственного указания какого-нибудь ресурса в каталоге. Таким образом, URL типа `http://www.MHProfessional.com/` указывает на тот же самый адрес, что и URL типа `http://www.MHProfessional.com/index.htm`.

У класса URL имеется ряд конструкторов, каждый из которых может сгенерировать исключение типа `MalformedURLException`. В одной из наиболее употребительных форм конструктора этого класса URL определяется в виде символьной строки, похожей на ту, что можно видеть в поле адреса, расположенном в верхней части окна браузера. Эта общая форма конструктора класса URL выглядит следующим образом:

```
URL(String спецификатор_URL) throws MalformedURLException
```

Две приведенные ниже формы конструктора данного класса позволяют разделить URL на отдельные составляющие.

```
URL(String имя_протокола, String имя_хоста, int порт, String путь)
    throws MalformedURLException
URL(String имя_протокола, String имя_хоста, String порт)
    throws MalformedURLException
```

Другой часто употребляемый конструктор данного класса позволяет указать существующий URL в качестве ссылочного контекста, а затем создать из этого контекста новый URL. И хотя это, на первый взгляд, слишком сложно, на самом деле очень просто и удобно. Общая форма такого конструктора выглядит следующим образом:

```
URL(URL объект URL, String спецификатор_URL)
    throws MalformedURLException
```

В следующем примере программы формируется URL, указывающий на страницу веб-сайта HerbSchildt.com, а затем просматриваются его свойства:

```
// Продемонстрировать применение класса URL
import java.net.*;
class URLEDemo {
    public static void main(String args[]) throws MalformedURLException {
        URL hp = new URL("http://www.HerbSchildt.com/Articles");

        System.out.println("Протокол: " + hp.getProtocol());
        System.out.println("Порт: " + hp.getPort());

        System.out.println("Хост: " + hp.getHost());
        System.out.println("Файл: " + hp.getFile());
        System.out.println("Полная форма: " + hp.toExternalForm());
    }
}
```

Эта программа выводит следующий результат:

```
Протокол: http
Порт: -1
Хост: www.HerbSchildt.com
Файл: /Articles
Полная форма: http://www.HerbSchildt.com/Articles
```

Обратите внимание на то, что порт имеет номер **-1**, а это означает, что он явно не установлен. Имея объект типа URL, можно извлечь из него связанные с ним данные. Чтобы получить доступ к конкретным битам данных или информационному наполнению объекта типа URL, следует создать из него объект типа URLConnection, вызвав его метод `openConnection()`, как показано ниже.

```
urlc = url.openConnection()
```

Метод `openConnection()` имеет следующую общую форму:

```
URLConnection openConnection() throws IOException
```

Этот метод возвращает объект типа URLConnection, связанный с вызывающим объектом типа URL. Следует, однако, иметь в виду, что метод `openConnection()` может сгенерировать исключение типа IOException.

Класс URLConnection

Класс URLConnection является классом общего назначения и предназначен для доступа к атрибутам удаленного ресурса. Как только будет установлено соединение с удаленным сервером, класс URLConnection можно использовать для про-

смотря свойств удаленного объекта, прежде чем переносить его локально. Эти атрибуты раскрываются в спецификации сетевого протокола HTTP и как таковые имеют смысл только для объектов типа URL, использующих протокол HTTP.

В классе `URLConnection` определяется несколько методов. Некоторые из них перечислены в табл. 22.5.

Таблица 22.5. Избранные методы из класса `URLConnection`

Метод	Описание
<code>int getContenLength()</code>	Возвращает длину в байтах содержимого, связанного с ресурсом. Если длина недоступна, возвращается значение <code>-1</code>
<code>long getContenLengthLong()</code>	Возвращает длину в байтах содержимого, связанного с ресурсом. Если длина недоступна, возвращается значение <code>-1</code>
<code>String getContenType()</code>	Возвращает тип содержимого, обнаруженного в ресурсе. Это значение поля заголовка <code>content-type</code> . Возвращает пустое значение <code>null</code> , если тип содержимого недоступен
<code>long getDate()</code>	Возвращает время и дату ответа в миллисекундах, прошедших с 1 января 1970 г.
<code>long getExpiration()</code>	Возвращает время и дату срока действия ресурса в миллисекундах, прошедших с 1 января 1970 г. Если дата срока действия ресурса недоступна, возвращается нуль
<code>String getHeaderField(int индекс)</code>	Возвращает значение поля заголовка по указанному <i>индексу</i> . (Индексы полей заголовков нумеруются, начиная с нуля.) Если значение параметра <i>индекс</i> превышает количество полей, то возвращается пустое значение <code>null</code>
<code>String getHeaderField(String имя_поля)</code>	Возвращает значение поля заголовка по указанному <i>имени поля</i> . Если указанное поле не найдено, то возвращается пустое значение <code>null</code>
<code>String getHeaderFieldKey(int индекс)</code>	Возвращает ключ поля заголовка по указанному <i>индексу</i> . (Индексы полей заголовков нумеруются, начиная с нуля.) Если значение параметра <i>индекс</i> превышает количество полей, то возвращается пустое значение <code>null</code>
<code>Map<String, List<String>> getHeaderFields()</code>	Возвращает отображение, содержащее все поля заголовков вместе с их значениями
<code>long getLastModified()</code>	Возвращает время и дату последней модификации ресурса в миллисекундах, прошедших с 1 января 1970 г. Если эта информация недоступна, то возвращается нуль
<code>InputStream getInputStream() throws IOException</code>	Возвращает поток ввода типа <code>InputStream</code> , привязанный к ресурсу. Этот поток ввода может использоваться для получения содержимого ресурса

Обратите внимание на то, что в классе `URLConnection` определяется несколько методов, управляющих информацией из заголовков. Заголовок состоит из пар ключей и значений, представленных в виде символьных строк. Используя метод `getHeaderField()`, можно получить значение, связанное с ключом заголовка. Вызывая метод `getHeaderField()`, можно получить отображение, содержащее все заголовки. Несколько стандартных полей заголовков доступны непосредственно через такие методы, как `getDate()` и `getContentType()`.

В следующем примере создается объект типа `URLConnection` с помощью метода `openConnection()`, вызываемого для объекта типа `URL`, а затем он применяется для проверки свойств и содержимого документа:

```
// Продемонстрировать применение класса URLConnection
import java.net.*;
import java.io.*;
import java.util.Date;
class UCDemo
{
    public static void main(String args[]) throws Exception {
        int c;
        URL hp = new URL("http://www.internic.net");
        URLConnection hpCon = hp.openConnection();

        // получить дату
        long d = hpCon.getDate();
        if(d==0)
            System.out.println("Сведения о дате отсутствуют.");
        else
            System.out.println("Дата: " + new Date(d));

        // получить тип содержимого
        System.out.println(
            "Тип содержимого: " + hpCon.getContentType());

        // получить дату срока действия ресурса
        d = hpCon.getExpiration();
        if(d==0)
            System.out.println(
                "Сведения о сроке действия отсутствуют.");
        else
            System.out.println(
                "Срок действия истекает: " + new Date(d));

        // получить дату последней модификации
        d = hpCon.getLastModified();
        if(d==0)
            System.out.println(
                "Сведения о дате последней модификации.");
        else
            System.out.println(
                "Дата последней модификации: " + new Date(d));

        // получить длину содержимого
        long len = hpCon.getContentLengthLong();
        if(len == -1)
            System.out.println("Длина содержимого недоступна.");
        else
            System.out.println("Длина содержимого: " + len);
    }
}
```

```

if(len != 0) {
    System.out.println("=== Содержимое ===");
    InputStream input = hpCon.getInputStream();
    while (((c = input.read()) != -1)) {
        System.out.print((char) c);
    }
    input.close();
} else {
    System.out.println("Содержимое недоступно.");
}
}
}

```

В этой программе сначала устанавливается соединение по протоколу HTTP с сервером `www.internic.net` через порт `80`. Затем в ней выводятся несколько значений из заголовков и извлекается содержимое. Ради интереса попробуйте выполнить эту программу, наблюдая результаты ее выполнения. А затем попробуйте для сравнения выбрать другой веб-сайт.

Класс `URLConnection`

В Java предоставляется подкласс `URLConnection`, производный от класса `URLConnection` и поддерживающий соединения по сетевому протоколу HTTP. Чтобы получить объект класса `URLConnection`, следует вызвать метод `openConnection()` для объекта типа `URL`, как описано выше, но результат нужно привести к типу `URLConnection`. (Разумеется, при этом необходимо убедиться, что соединение по протоколу HTTP действительно установлено.) Получив ссылку на объект класса `URLConnection`, можно вызвать любые методы, унаследованные от класса `URLConnection`, а также любые методы, определенные в самом классе `URLConnection`. Некоторые методы из этого класса перечислены в табл. 22.6.

Таблица 22.6. Избранные методы из класса `URLConnection`

Метод	Описание
static boolean getFollowRedirects()	Возвращает логическое значение true , если автоматически следует переадресация, а иначе — логическое значение false
String getRequestMethod()	Возвращает строковое представление метода, которым выполняется запрос по URL. По умолчанию запрос выполняется методом GET . Доступны и другие методы, в том числе POST
int getResponseCode() throws IOException	Возвращает код ответа по протоколу HTTP. Если код ответа не может быть получен, возвращается значение -1 . При разрыве соединения генерируется исключение типа IOException
String getResponseMessage() throws IOException	Возвращает ответное сообщение, связанное с кодом ответа. Если ответное сообщение отсутствует, то возвращает пустое значение null

Окончание табл. 22.6

Метод	Описание
static void setFollowRedirects (boolean способ)	Если параметр <i>способ</i> принимает логическое значение true , то переадресация выполняется автоматически. Если же он принимает логическое значение false , то переадресация не происходит. По умолчанию переадресация выполняется автоматически
void setRequestMethod (String способ) throws ProtocolException	Задаёт метод, которым делаются запросы по протоколу HTTP, в соответствии со значением параметра <i>способ</i> . По умолчанию принят метод GET , но доступны и другие методы, в том числе POST . Если в качестве параметра <i>способ</i> указано неправильное значение, то генерируется исключение типа ProtocolException

В приведенном ниже примере программы демонстрируется применение класса `URLConnection`. Сначала в этой программе устанавливается соединение с веб-сайтом `www.google.com`, а затем выводится метод запроса, код ответа и ответное сообщение. И наконец, выводятся ключи и значения из заголовка ответа.

```
// Продемонстрировать применение класса HttpURLConnection
import java.net.*;
import java.io.*;
import java.util.*;
class HttpURLDemo
{
    public static void main(String args[]) throws Exception {
        URL hp = new URL("http://www.google.com");
        HttpURLConnection hpCon =
            (HttpURLConnection) hp.openConnection();

        // вывести метод запроса
        System.out.println("Метод запроса: " + hpCon.getRequestMethod());

        // вывести код ответа
        System.out.println("Код ответа: " + hpCon.getResponseCode());

        // вывести ответное сообщение
        System.out.println("Ответное сообщение: " +
            hpCon.getResponseMessage());

        // получить список полей и множество ключей из заголовка
        Map<String, List<String>> hdrMap = hpCon.getHeaderFields();
        Set<String> hdrField = hdrMap.keySet();
        System.out.println("\nДалее следует заголовок:");

        // вывести все ключи и значения из заголовка
        for(String k : hdrField) {
            System.out.println("Ключ: " + k + " Значение: " +
                hdrMap.get(k));
        }
    }
}
```

Ниже приведен результат, выводимый данной программой (разумеется, точный ответ, возвращаемый сайтом `www.google.com`, будет меняться с течением времени).

Метод запроса: GET
 Код ответа: 200
 Сообщение ответа: OK

Далее следует заголовок:

Ключ: Transfer-Encoding Value: [chunked]
 Ключ: X-Frame-Options Value: [SAMEORIGIN]
 Ключ: null Value: [HTTP/1.1 200 OK]
 Ключ: Server Value: [gws]
 Ключ: Cache-Control Value: [private, max-age=0]
 Ключ: Set-Cookie Value:
 [NID=67=rMTQWvn5eVIYA2d8F5Iu 8L-68wiMACyaXYqeSelbvR8SzQQ_PaDCy5mNbxuw5XtdcjY
 KIwmy3oVJmly0qZdibB0kQfJmtHpAtO6lGVwumQlApgSXWjZ67yHxQX3g3-h; expires=Wed,
 23-Apr-2014 18:31:09 GMT; path=/; domain=.google.com; HttpOnly,
 PREF=ID=463b5df7b9ced9d8:FF=0:TM=1382466669:LM=1382466669:S=3LI-oT-Dzi46U1On
 ; expires=Thu, 22-Oct-2015 18:31:09 GMT; path=/; domain=.google.com]
 Ключ: Expires Value: [-1]
 Ключ: X-XSS-Protection Value: [1; mode=block]
 Ключ: P3P Value: [CP="This is not a P3P policy! See
 http://www.google.com/support/accounts/bin/answer.py?hl=en&answer=151657
 formore info."]
 Ключ: Date Value: [Tue, 22 Oct 2013 18:31:09 GMT]
 Ключ: Content-Type Value: [text/html; charset=ISO-8859-1]

Обратите внимание на порядок вывода ключей и значений из заголовка. Сначала вызывается метод `getHeaderFields()`, унаследованный от класса `URLConnection`, чтобы получить отображение ключей и значений из заголовка. Затем для этого отображения вызывается метод `keySet()`, чтобы извлечь множество ключей из заголовка. Далее полученное множество ключей перебирается в цикле `for` в стиле `for each`, где вызывается также метод `get()`, чтобы получить значение, связанное с каждым ключом.

Класс URI

Класс `URI` инкапсулирует *универсальный идентификатор ресурса* (URI), очень похожий на URL. На самом деле URL является подмножеством URI. Если URI обозначает стандартный способ идентификации ресурсов, то URL описывает также доступ к ресурсу.

Cookie-файлы

В состав пакета `java.net` входят классы и интерфейсы, помогающие управлять cookie-файлами, которые можно использовать для организации сеансов связи по сетевому протоколу HTTP с сохранением состояния, в отличие от сеансов связи без сохранения состояния. К их числу относятся классы `CookieHandler`, `CookieManager` и `HttpCookie`, а также интерфейсы `CookiePolicy` и `CookieStore`. Рассмотрение сеансов связи по сетевому протоколу HTTP с сохранением состояния выходит за рамки данной книги.

На заметку! Подробнее о применении cookie-файлов вместе с сервлетами см. в главе 38.

Серверные сокеты по протоколу TCP/IP

Как упоминалось ранее, в Java имеются различные классы сокетов, которые должны применяться при разработке серверных приложений. В частности, класс `ServerSocket` применяется для создания серверов, которые принимают запросы как от локальных, так и от удаленных клиентских программ, желающих установить соединение с ними через открытые порты. Класс `ServerSocket` заметно отличается от обычных классов типа `Socket`. Когда создается объект класса `ServerSocket`, он регистрируется в системе как заинтересованный в соединении с клиентами. Конструкторы класса `ServerSocket` отражают номер порта, через который требуется принимать запросы на соединение, а также (хотя и необязательно) длину очереди, которая организуется для данного порта. Длина очереди сообщает системе, сколько клиентских соединений можно поддерживать, прежде чем отказать в соединении. По умолчанию задается длина очереди 50 соединений. При определенных условиях конструкторы класса `ServerSocket` могут сгенерировать исключение типа `IOException`. Конструкторы этого класса перечислены в табл. 22.7.

Таблица 22.7. Конструкторы класса `ServerSocket`

Конструктор	Описание
<code>ServerSocket(int порт) throws IOException</code>	Создает серверный сокет в указанном <i>порте</i> с длиной очереди 50 соединений
<code>ServerSocket(int порт, int максимум) throws IOException</code>	Создает серверный сокет в указанном <i>порте</i> с максимальной длиной очереди, определяемой параметром <i>максимум</i>
<code>ServerSocket(int порт, int максимум, InetAddress локальный_адрес) throws IOException</code>	Создает серверный сокет в указанном <i>порте</i> с максимальной длиной очереди, определяемой параметром <i>максимум</i> . На многоканальном хосте параметр <i>локальный_адрес</i> обозначает IP-адрес, к которому привязан сокет

В состав класса `ServerSocket` входит метод `accept()`. Он реализует блокирующий вызов, чтобы ожидать от клиента начала соединения, а затем вернуть обычный объект типа `Socket`, который может далее служить для взаимодействия с клиентом.

Дейтаграммы

Сетевое взаимодействие по протоколу TCP/IP подходит для большинства сетевых нужд. Оно обеспечивает сериализуемые, предсказуемые и надежные потоки ввода-вывода пакетов данных. Но все это обходится совсем не даром. Протокол TCP включает в себя немало сложных алгоритмов адаптации к перегруженности сетей, а также самые пессимистические предположения относительно потери пакетов. Это в какой-то степени делает неэффективным способ переноса данных по сети. Альтернативой ему служат дейтаграммы.

Дейтаграммы — это порции данных, передаваемых между машинами. В некотором отношении они подобны сильным броскам тренированного, но подслеповатого принимающего в сторону третьего бейсмена в бейсболе. Даже если дейтаграмма и передается в нужном направлении, нет никаких гарантий, что она достигнет цели или кто-нибудь окажется на месте, чтобы ее перехватить. Аналогично, когда дейтаграмма принимается, нет никакой гарантии, что она не была повреждена при передаче или что ее отправитель все еще ожидает ответа.

Дейтаграммы реализуются в Java поверх сетевого протокола UDP с помощью двух классов: `DatagramPacket` (контейнер данных) и `DatagramSocket` (механизм для передачи и приема пакетов типа `DatagramPacket`). Каждый из этих классов рассматривается далее по отдельности.

Класс `DatagramSocket`

В классе `DatagramSocket` определяются четыре открытых конструктора. Их общие формы приведены ниже.

```
DatagramSocket() throws SocketException  
DatagramSocket(int порт) throws SocketException  
DatagramSocket(int порт, InetAddress IP адрес) throws SocketException  
DatagramSocket(SocketAddress адрес) throws SocketException
```

Первый конструктор создает объект класса `DatagramSocket`, связанный с любым незанятым портом локального компьютера; второй конструктор — объект класса `DatagramSocket`, связанный с указанным *портом*; третий конструктор — объект класса `DatagramSocket`, связанный с указанным *портом* и объектом класса `InetAddress`. Четвертый конструктор — объект класса `DatagramSocket`, связанный с заданным объектом класса `SocketAddress`. Класс `SocketAddress` является абстрактным и реализуется конкретным классом `InetSocketAddress`, который инкапсулирует IP-адрес с номером порта. Все конструкторы класса `DatagramSocket` могут генерировать исключение типа `SocketException`, если при создании сокета возникают ошибки.

В классе `DatagramSocket` определяется немало методов. Наиболее важными из них являются методы `send()` и `receive()`, общие формы которых представлены ниже.

```
void send(DatagramPacket пакет) throws IOException  
void receive(DatagramPacket пакет) throws IOException
```

Метод `send()` отправляет в порт указанный *пакет*. А метод `receive()` ожидает приема через порт указанного *пакета* и возвращает полученный результат.

В классе `DatagramSocket` определяется также метод `close()`, закрывающий сокет данного типа. Начиная с версии JDK 7 класс `DatagramSocket` реализует интерфейс `AutoCloseable`, что позволяет управлять сокетом типа `DatagramSocket` в блоке оператора `try` с ресурсами. Другие методы из данного класса предоставляют доступ к различным атрибутам, связанным с сокетом типа `DatagramSocket`. Эти методы перечислены в табл. 22.8.

Таблица 22.8. Методы из класса `DatagramSocket`

Метод	Описание
<code>InetAddress getAddress()</code>	Возвращает адрес, если сокет подключен, а иначе — логическое значение null
<code>int getLocalPort()</code>	Возвращает номер локального порта
<code>int getPort()</code>	Возвращает номер порта, к которому подключен сокет. Если же сокет не подключен ни к одному из портов, то возвращается значение -1
<code>boolean isBound()</code>	Возвращает логическое значение true , если сокет привязан к адресу, а иначе — логическое значение false
<code>boolean isConnected()</code>	Возвращает логическое значение true , если сокет подключен к серверу, а иначе — логическое значение false
<code>void setSoTimeout(int миллисекунд) throws SocketException</code>	Устанавливает период ожидания, равный заданному количеству <i>миллисекунд</i>

Класс `DatagramPacket`

В классе `DatagramPacket` определяется несколько конструкторов. Ниже приведены четыре конструктора данного класса.

```
DatagramPacket(byte данные[], int размер)
DatagramPacket(byte данные[], int смещение, int размер)
DatagramPacket(byte данные[], int размер, InetAddress IP-адрес,
                int порт)
DatagramPacket(byte данные[], int смещение, int размер,
                InetAddress IP-адрес, int порт)
```

Первый конструктор определяет буфер, который будет принимать данные, и размер пакета. Он служит для приема данных через сокет типа `DatagramSocket`. Второй конструктор позволяет указать смещение в буфере, где должны быть размещены данные. Третий конструктор позволяет указать целевой адрес и порт, используемые в соquete типа `DatagramSocket` для отправки пакета по месту назначения. Четвертый конструктор организует передачу пакетов, начиная с указанного смещения в буфере данных. Два первых конструктора следует рассматривать как запечатывание письма в конверт, а два других — как запечатывание письма в конверт и написание адреса на нем.

В классе `DatagramPacket` определяется несколько методов, включая перечисленные в табл. 22.9. Эти методы предоставляют доступ к адресу и номеру порта отдельного пакета, а также к исходным данным и их длине.

Таблица 22.9. Методы из класса `DatagramPacket`

Метод	Описание
<code>InetAddress getAddress()</code>	Возвращает адрес источника (для принимаемых дейтаграмм) или места назначения (для отправляемых дейтаграмм)

Метод	Описание
byte[] getData()	Возвращает массив байтов данных, содержащихся в дейтаграмме. Используется в основном для извлечения данных из дейтаграммы после ее приема
int getLength()	Возвращает длину достоверных данных, содержащихся в массиве байтов, который должен быть возвращен из метода getData() . Эта длина может не полностью совпадать с длиной массива байтов
int getOffset()	Возвращает начальный индекс данных
int getPort()	Возвращает номер порта
void setAddress(InetAddress IP-адрес)	Устанавливает адрес, по которому отправляется пакет. Адрес обозначается параметром <i>IP-адрес</i>
void setData(byte[] данные)	Устанавливает буфер в виде заданного массива <i>данные</i> , нулевое смещение и длину, равную количеству байтов в массиве <i>данные</i>
void setData(byte[] данные, int индекс, int размер)	Устанавливает буфер в виде заданного массива <i>данные</i> , смещение — по указанному <i>индексу</i> , а длину — по заданному <i>размеру</i>
void setLength(int размер)	Устанавливает длину пакета по заданному <i>размеру</i>
void setPort(int порт)	Устанавливает заданный <i>порт</i>

Пример обработки дейтаграмм

В приведенном ниже примере реализуется очень простое сетевое взаимодействие клиента с сервером. Сообщения вводятся в окне на сервере и передаются по сети на сторону клиента, где они выводятся на экран.

```
// Продемонстрировать обработку дейтаграмм
import java.net.*;

class WriteServer {
    public static int serverPort = 998;
    public static int clientPort = 999;
    public static int buffer_size = 1024;
    public static DatagramSocket ds;
    public static byte buffer[] = new byte[buffer_size];

    public static void TheServer() throws Exception {
        int pos=0;
        while (true) {
            int c = System.in.read();
            switch (c) {
                case -1:
                    System.out.println("Сервер завершает сеанс связи.");
                    ds.close();
                    return;
                case '\r':
                    break;
                case '\n':
```

```

        ds.send(new DatagramPacket(buffer, pos,
                                   InetAddress.getLocalHost(), clientPort));
        pos=0;
        break;
    default:
        buffer[pos++] = (byte) c;
    }
}

public static void TheClient() throws Exception {
    while(true) {
        DatagramPacket p = new DatagramPacket(buffer, buffer.length);
        ds.receive(p);
        System.out.println(new String(p.getData(), 0, p.getLength()));
    }
}

public static void main(String args[]) throws Exception {
    if(args.length == 1) {
        ds = new DatagramSocket(serverPort);
        TheServer();
    } else {
        ds = new DatagramSocket(clientPort);
        TheClient();
    }
}
}

```

Этот пример программы ограничивается обменом данными между портами локальной машины с помощью конструктора класса `DatagramSocket`. Чтобы воспользоваться этой программой, выполните в одном окне следующую команду:

```
java WriteServer
```

Это будет клиент. Затем выполните в другом окне такую команду:

```
java WriteServer 1
```

Это будет сервер. Все, что вы введете в окне сервера, будет отправлено в окно клиента после ввода символа перевода строки.

На заметку! Применение дейтаграмм на вашем компьютере может быть запрещено, например, установленным на нем брандмауэром. В таком случае вам не удастся воспользоваться программой из приведенного выше примера. Кроме того, номера портов, указанные в программе, подходят для системы автора книги, но в вашей системе их, возможно, придется дополнительно настроить.

Класс Applet

В этой главе рассматривается класс `Applet`, который служит основанием для создания апплетов. Этот класс входит в состав пакета `java.applet` и содержит ряд методов, обеспечивающих полный контроль над выполнением апплета. Кроме того, в пакете `java.applet` определяются три интерфейса: `AppletContext`, `AudioClip` и `AppletStub`.

Два типа апплетов

Следует особо подчеркнуть, что имеются два типа апплетов. Первый тип основывается непосредственно на классе `Applet`, описываемом в этой главе. В апплетах данного типа используются средства из библиотеки `Abstract Window Toolkit (AWT)` для предоставления графического пользовательского интерфейса (ГПИ, если такой интерфейс вообще используется). Этот тип апплетов доступен с самого начала существования `Java`.

Второй тип апплетов основывается на классе `JApplet` из библиотеки `Swing`. В апплетах типа `Swing` применяются классы из библиотеки `Swing` для построения ГПИ. Библиотека `Swing` предоставляет более богатый и зачастую более простой в употреблении ГПИ, чем библиотека `AWT`. Поэтому апплеты, построенные на основе библиотеки `Swing`, наиболее распространены в настоящее время. Но и традиционные апплеты, построенные на основе библиотеки `AWT`, по-прежнему применяются, особенно когда требуется построить очень простой пользовательский интерфейс. Поэтому применение апплетов, построенных на основе обеих библиотек `AWT` и `Swing`, вполне обосновано.

В этой главе описываются апплеты, создаваемые на основе библиотеки `AWT`. Но поскольку класс `JApplet` наследует от класса `Applet`, то все средства последнего доступны и в классе `JApplet`. Следовательно, большая часть материала этой главы распространяется на оба упомянутых выше типа апплетов. И даже если вас интересуют только апплеты типа `Swing`, то вам все равно будет полезно и даже нужно ознакомиться с материалом этой главы. Следует, однако, иметь в виду, что создание апплетов на основе библиотеки `Swing` связано с рядом дополнительных ограничений, которые будут обсуждаться далее в этой книге, когда пойдет речь о библиотеке `Swing`.

Основы разработки апплетов

В главе 13 была рассмотрена общая форма апплета, а также действия, которые требуется предпринять для его компиляции и запуска на выполнение. Прежде всего напомним вкратце основные положения об апплетах.

Все классы апплетов являются производными от класса `Applet`. Апплеты не являются самостоятельными программами, они выполняются в окне веб-браузера или средства просмотра апплетов. Иллюстрации, приведенные в этой главе, были получены с помощью средства просмотра апплетов, которое называется **appletviewer** и входит в комплект JDK.

Выполнение апплета не начинается с метода `main()`. На самом деле лишь немногие апплеты имеют метод `main()`. Вместо этого запуск апплета и управление его выполнением осуществляется с помощью совершенно другого, описываемого ниже механизма. Кроме того, вывод данных в окно апплета не осуществляется методом `System.out.println()`. Вместо этого в апплетах типа AWT вывод осуществляется различными методами из библиотеки AWT, в том числе методом `drawString()`, который выводит символьную строку в точку с указанными координатами X,Y. Ввод данных в апплетах также осуществляется иначе, чем в консольных приложениях.

Прежде чем использовать апплет, следует выбрать стратегию его развертывания. Для этого имеются два основных подхода. Первый подход подразумевает применение *сетевого протокола запуска приложений на Java* (Java Network Launch Protocol – JNLP). Такой подход обеспечивает больше удобств, особенно в отношении функционально усовершенствованных приложений для Интернета. Для разработки реальных апплетов сетевой протокол JNLP зачастую является наилучшим выбором, но подробное обсуждение этого протокола выходит за рамки данной книги. (Подробнее о сетевом протоколе JNLP можно узнать из документации на комплект JDK.) Правда, протокол JNLP для примеров апплетов, представленных в этой главе, не требуется.

Второй подход к развертыванию апплетов подразумевает его определение непосредственно в HTML-файле, не прибегая к сетевому протоколу JNLP. Это первоначальный способ запуска апплетов, который применялся с момента создания Java и до сих пор употребляется для запуска простых апплетов. Кроме того, благодаря унаследованной простоте этот подход применяется в примерах апплетов, представленных в данной книге. На момент написания данной книги корпорация Oracle рекомендовала дескриптор `APPLET`, поэтому в данной книге используется именно он. (Однако в настоящее время применение дескриптора `APPLET` не рекомендуется по спецификации HTML. Альтернативой ему служит дескриптор `OBJECT`. Поэтому за последними рекомендациями в этом отношении имеет смысл обратиться к документации на комплект JDK.) Когда в HTML-файле встречается дескриптор `APPLET`, веб-браузер, поддерживающий Java, выполняет указанный апплет.

Использование дескриптора `APPLET` дает еще одно преимущество при разработке апплетов, поскольку он позволяет легко просматривать и проверять апплет. Для этого достаточно ввести комментарий, содержащий дескриптор `APPLET`, в заголовок файла исходного кода Java. Таким образом, исходный код будет документирован элементами HTML-разметки, необходимыми апплету, и это дает возможность проверить скомпилированный апплет, запустив средство просмотра апплетов с заданным файлом исходного кода Java. Ниже приведен пример такого комментария.

```

/*
<applet code="MyApplet" width=200 height=60>
</applet>
*/

```

Этот комментарий включает дескриптор `APPLET`, который запускает апплет `MyApplet` в окне размером 200 пикселей в ширину и 60 пикселей в высоту. Включение команды `APPLET` облегчает проверку апплетов, поэтому все апплеты, примеры которых представлены в этой книге, будут содержать соответствующий дескриптор `APPLET`, введенный в комментарий.

На заметку! Как отмечалось в главе 13, начиная с обновления 21 версии Java 7 апплеты Java должны быть подписаны для предотвращения предупреждений о нарушении защиты при их выполнении в браузере. На самом деле выполнение апплета может быть иногда запрещено. Апплеты, которые сохраняются в локальной файловой системе, например, после компиляции примеров из данной книги, особенно чувствительны к подобному изменению. Так, для выполнения локального апплета в браузере, возможно, потребуется настройка параметров безопасности на панели управления Java. На момент написания данной книги компания Oracle рекомендовала пользоваться не локальными апплетами, а апплетами, выполняемыми через веб-сервер. Более того, выполнение неподписанных локальных апплетов может быть заблокировано в будущем. В общем, подписание апплетов, которые будут распространяться через Интернет, например для коммерческого применения, безусловно необходимо. Обсуждение принципов и методик подписания апплетов (и других видов прикладных программ на Java) выходит за рамки данной книги. Впрочем, обширную информацию по данному вопросу можно найти на веб-сайте компании Oracle. И наконец, как упоминалось ранее, простейший способ опробовать представленные здесь примеры апплетов — воспользоваться средством их просмотра `appletviewer`.

Класс Applet

В этом классе определяются методы, перечисленные в табл. 23.1. Класс `Applet` обеспечивает всю необходимую поддержку для выполнения апплета, в том числе для его запуска и остановки. Кроме того, этот класс предоставляет методы для загрузки и воспроизведения графических изображений, а также методы для загрузки и воспроизведения аудиоклипов. Класс `Applet` расширяет класс `Panel` из библиотеки `AWT`. В свою очередь, класс `Panel` расширяет класс `Container`, а тот — класс `Component`. Все эти классы обеспечивают поддержку графического оконного интерфейса на базе Java. Краткий обзор библиотеки `AWT` представлен в последующих главах.

Таблица 23.1. Методы из класса Applet

Метод	Описание
<code>void destroy()</code>	Вызывается браузером непосредственно перед уничтожением апплета. Этот метод переопределяется в апплете, если он нуждается в выполнении ряда действий по очистке перед уничтожением апплета
<code>AccessibleContext getAccessibleContext()</code>	Возвращает контекст доступности для вызывающего объекта
<code>AppletContext getAppletContext()</code>	Возвращает контекст, связанный с апплетом

Метод	Описание
String getAppletInfo()	Переопределенные версии этого метода должны возвращать символьную строку, описывающую апплет. Реализация этого метода по умолчанию возвращает пустое значение null
AudioClip getAudioClip(URL url)	Возвращает объект типа AudioClip , инкапсулирующий аудиоклип, находящийся в месте, обозначаемом параметром <i>url</i>
AudioClip getAudioClip(URL url, String имя_клипа)	Возвращает объект типа AudioClip , инкапсулирующий аудиоклип, находящийся в месте, обозначаемом параметром <i>url</i> , и имеющим указанное <i>имя_клипа</i>
URL getCodeBase()	Возвращает URL, связанный с вызывающим апплетом
URL getDocumentBase()	Возвращает URL, указывающий на HTML-документ, вызывающий апплет
Image getImage(URL url)	Возвращает объект типа Image , инкапсулирующий графическое изображение, находящееся в месте, обозначаемом параметром <i>url</i>
Image getImage(URL url, String имя_изображения)	Возвращает объект типа Image , инкапсулирующий графическое изображение, находящееся в месте, обозначаемом параметром <i>url</i> и имеющим указанное <i>имя_изображения</i>
Locale getLocale()	Возвращает объект типа Locale , используемый различными классами и методами, учитывающими региональные настройки
String getParameter(String имя_параметра)	Возвращает параметр по указанному <i>имени_параметра</i> . Если указанный параметр не найден, то возвращается значение пустое null
String[] [] getParameterInfo()	Метод, переопределяющий данный метод, должен возвращать таблицу объектов типа String , описывающую параметры, распознаваемые апплетом. Каждая запись в таблице должна состоять из трех символьных строк, содержащих имя параметра, описание его типа и/или пределы его значений, а также все необходимые пояснения. Реализация данного метода по умолчанию возвращает пустое значение null
void init()	Вызывается при запуске апплета на выполнение. Это первый метод, вызываемый при выполнении апплета
boolean isActive()	Возвращает логическое значение true , если апплет запущен на выполнение. Возвращает логическое значение false , если выполнение апплета остановлено
boolean isValidateRoot()	Возвращает логическое значение true , указывающее на то, что апплет проверяет корневой каталог
static final AudioClip newAudioClip(URL url)	Возвращает объект типа AudioClip , инкапсулирующий аудиоклип, находящийся в месте, обозначаемом параметром <i>url</i> . Этот метод похож на метод getAudioClip() , за исключением того, что он является статическим и может быть выполнен без объекта типа Applet
void play(URL url)	Если аудиоклип найден в месте, обозначаемом параметром <i>url</i> , то он выполняется
void play(URL url, String имя_клипа)	Если аудиоклип, имеющий указанное <i>имя_клипа</i> , найден в месте, обозначаемом параметром <i>url</i> , то он выполняется
void resize(Dimension размеры)	Изменяет указанные <i>размеры</i> апплета, где Dimension — это класс, входящий в состав пакета java.awt . Он содержит два целочисленных поля: width и height

Окончание табл. 23.1

Метод	Описание
void resize(int ширина, int высота)	Изменяет размеры апплета, обозначаемые параметрами <i>ширина</i> и <i>высота</i>
final void setStub(AppletStub загрузчик)	Устанавливает заданную <i>загрузку</i> для апплета. Этот метод применяется исполняющей системой и никогда не вызывается непосредственно из апплета. <i>Загрузчик</i> — это небольшой фрагмент кода, обеспечивающего связь апплета с браузером
void showStatus(String строка)	Выводит указанную <i>строку</i> в строке состояния окна браузера или средства просмотра апплетов. Если же в окне браузера не поддерживается строка состояния, то ничего не происходит
void start()	Вызывается браузером, когда апплет должен начать (или возобновить) выполнение. Автоматически вызывается после метода init() в начале работы апплета
void stop()	Вызывается браузером для приостановки выполнения апплета. Будучи остановленным, апплет перезапускается в результате вызова метода start()

Структура апплетов

Как правило, *апплет* — это оконная программа с ГПИ, и поэтому его структура отличается от структуры консольных программ, примеры которых были представлены в части I данной книги. Если у вас имеется некоторый опыт программирования ГПИ, то при написании апплетов вы будете чувствовать себя, как дома. Если же у вас нет такого опыта, то вам придется усвоить некоторые понятия, которые поясняются ниже.

Во-первых, апплеты управляются событиями. И хотя рассмотрение обработки событий придется отложить до следующей главы, важно хотя в общих чертах понимать, каким образом управляемая событиями архитектура оказывает влияние на разработку апплетов. Апплет напоминает ряд служебных процедур, обрабатывающих прерывания, действуя по следующему принципу. Апплет ожидает до тех пор, пока не произойдет какое-нибудь событие. Исполняющая система извещает апплет о событии, вызывая обработчик событий, предоставляемый апплетом. Как только это произойдет, апплет должен предпринять соответствующие действия и немедленно возвратить управление. И это очень важный момент. Как правило, апплет не должен входить в режим работы, в котором он будет удерживать управление в течение длительного периода времени. Вместо этого он должен выполнить определенные действия в ответ на происходящие события, а затем возвратить управление исполняющей системе. В тех случаях, когда апплету требуется выполнить какое-нибудь повторяющееся действие (например, отображать в окне прокручивающееся сообщение), следует запустить дополнительный поток исполнения. (Соответствующий пример будет представлен ниже.) Во-вторых, только пользователь инициирует взаимодействие с апплетом, но никак иначе! Как известно, если в консольной программе требуется ввести данные, то сначала пользователю выводится соответствующее приглашение, а затем вызывается некоторый метод ввода, например `readLine()`. Совсем иначе дело обстоит в апплетах. Пользователь взаимодействует с апплетами так и тогда, как и когда он пожелает. Подобные взаимодействия передаются апплету

в виде событий, на которые тот должен отреагировать. Так, если пользователь щелкнет кнопкой мыши в окне апплета, передается событие от щелчка кнопкой мыши. Если пользователь нажимает клавишу, когда окно апплета имеет фокус ввода, то передается событие от нажатия клавиши. Как будет показано в последующих главах, апплеты могут содержать различные элементы управления, в том числе нажимаемые кнопки и устанавливаемые флажки. Всякий раз, когда пользователь взаимодействует с одним из этих элементов управления, происходят определенные события.

Структуру апплетов понять труднее, чем структуру консольных программ, но Java позволяет сделать ее как можно более простой. Если у вас имеется некоторый опыт написания программ под Windows (или других ОС с ГПИ), то вам должно быть известно, насколько путающе может выглядеть эта исполняющая среда. К счастью, Java предоставляет в ваше распоряжение намного более ясный и понятный подход, которым можно овладеть гораздо быстрее.

Скелет апплета

Во всех апплетах, кроме самых тривиальных, переопределяются методы, обеспечивающие основные механизмы взаимодействия браузера или средства просмотра апплетов с самим апплетом и управляющие его выполнением. К их числу относятся следующие четыре метода: `init()`, `start()`, `stop()` и `destroy()`. Они применяются во всех апплетах и определяются в классе `Applet`. Предоставляются также реализации всех этих методов по умолчанию, но переопределять их не требуется лишь в очень простых апплетах.

И в апплетах, создаваемых на основе библиотеки AWT (т.е. тех, что обсуждаются в этой главе), нередко переопределяется метод `paint()`, определенный в классе `Component` из библиотеки AWT. Этот метод вызывается в том случае, если требуется повторно вывести данные из апплета. (В апплетах, создаваемых на основе библиотеки Swing, для решения этой задачи применяется другой механизм.) Все пять упомянутых выше методов могут быть собраны в приведенный ниже скелет апплета.

```
// Скелет апплета
import java.awt.*;
import java.applet.*;
/*
  <applet code="AppletSkel" width=300 height=100>
  </applet>
*/

public class AppletSkel extends Applet {
    // Этот метод вызывается первым
    public void init() {
        // инициализация
    }

    /* Этот метод вызывается вторым, после метода init().
       Вызывается также при перезапуске апплета. */
    public void start() {
        // начать или возобновить выполнение апплета
    }

    // Этот метод вызывается при остановке апплета
    public void stop() {
```

```
        // приостановить выполнение апплета
    }

    /* Этот метод вызывается перед уничтожением апплета.
       Это последний выполняемый метод. */
    public void destroy() {
        // выполнить завершающие действия
    }

    // Этот метод вызывается, когда окно апплета
    // должно быть восстановлено.
    public void paint(Graphics g) {
        // повторно воспроизвести содержимое окна
    }
}
```

Несмотря на то что этот шаблон практически ничего не делает, его можно откомпилировать и запустить на выполнение. Если он запускается на выполнение в средстве просмотра апплетов `appletviewer`, то создает окно, показанное на рис. 23.1. На этом и последующих рисунках вида апплета в окне средства просмотра апплетов `appletviewer` может отличаться в зависимости от конкретной исполняющей среды. С целью проиллюстрировать этот факт для получения моментальных снимков экрана были выбраны разные исполняющие среды.

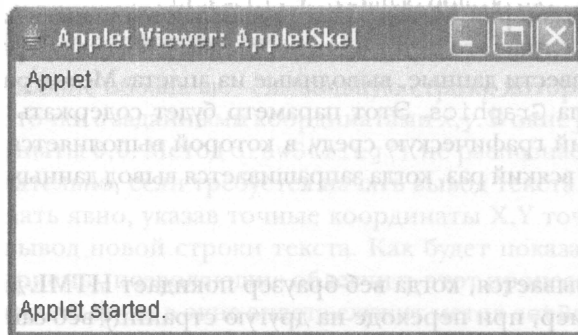


Рис. 23.1. Окно, воспроизводимое скелетом апплета

Инициализация и прекращение работы апплета

Важно понимать порядок вызова различных методов, представленных выше в скелете апплета. При запуске апплета на выполнение по порядку вызываются следующие методы:

- `init()`
- `start()`
- `paint()`

Когда выполнение апплета прекращается, по порядку вызываются следующие методы:

- `stop()`
- `destroy()`

Рассмотрим эти методы более подробно.

Метод `init()`

Это первый метод в цепочке вызовов. Именно в нем следует инициализировать переменные. Во время выполнения апплета этот метод вызывается лишь один раз.

Метод `start()`

После метода `init()` вызывается метод `start()`, который также вызывается для перезапуска апплета после его остановки. Если метод `init()` вызывается лишь один раз при первоначальной загрузке апплета, то метод `start()` вызывается каждый раз, когда HTML-документ, содержащий апплет, воспроизводится на экране. Следовательно, если пользователь покидает веб-страницу, а затем возвращается обратно, апплет возобновляет каждый раз свою работу с метода `start()`.

Метод `paint()`

Метод `paint()` вызывается всякий раз, когда выводимые из апплета данные должны быть воспроизведены повторно. Эта ситуация возникает в нескольких случаях. Например, окно, в котором выполняется апплет, может быть перекрыто другим окном, а затем вновь открыто. Или же окно апплета может быть свернуто, а затем восстановлено до нормальных размеров. Метод `paint()` вызывается также в начале выполнения апплета. Но в любом случае его метод `paint()` вызывается всякий раз, когда требуется повторно воспроизвести данные, выводимые из апплета. Метод `paint()` принимает один параметр типа `Graphics`. Этот параметр будет содержать графический контекст, описывающий графическую среду, в которой выполняется апплет. Такой контекст используется всякий раз, когда запрашивается вывод данных из апплета.

Метод `stop()`

Этот метод вызывается, когда веб-браузер покидает HTML-документ, содержащий апплет, например, при переходе на другую страницу веб-сайта. Когда вызывается метод `stop()`, апплет может еще работать. Поэтому метод `stop()` следует вызывать для приостановки потоков исполнения, которые не должны выполняться, когда апплет недоступен. Эти потоки исполнения можно перезапустить, когда вызывается метод `start()`, если пользователь возвращается на страницу.

Метод `destroy()`

Метод `destroy()` вызывается, когда исполняющая среда определяет, что апплет должен быть полностью удален из оперативной памяти. В этот момент следует освободить все ресурсы, используемые апплетом. Вызову метода `destroy()` всегда предшествует вызов метода `stop()`.

Переопределение метода `update()`

Иногда в апплете требуется переопределить еще один метод, определенный в библиотеке AWT и называемый `update()`. Этот метод вызывается, когда апплет запрашивает повторное воспроизведение определенной части окна. В версии ме-

тогда `update()` по умолчанию просто вызывается метод `paint()`. Но метод `update()` можно переопределить таким образом, чтобы выполнить повторное воспроизведение более изощренным способом. В общем, переопределение метода `update()` — это специальный прием, который применяется не во всех апплетах, и поэтому в примерах апплетов из этой главы метод `update()` не переопределяется.

Простые методы воспроизведения апплетов

Как упоминалось выше, апплеты воспроизводятся в окне, и для выполнения операций ввода-вывода апплеты, построенные на основе библиотеки AWT, пользуются ее средствами. Методы, процедуры и приемы, связанные с библиотекой AWT, будут обсуждаться в последующих главах, но некоторые из них следует рассмотреть теперь, поскольку они понадобятся для разработки примеров апплетов. (Напомним, что апплеты, создаваемые на основе библиотеки Swing, будут рассматриваться далее в этой книге.)

Как пояснялось в главе 13, для вывода символьной строки из апплета служит метод `drawString()` из класса `Graphics`. Как правило, этот метод вызывается из метода `update()` или `paint()`. Он имеет следующую общую форму:

```
void drawString(String сообщение, int x, int y)
```

где параметр *сообщение* обозначает символьную строку, которая должна быть выведена, начиная с точки с заданными координатами *x, y*. В окне Java левый верхний угол имеет координаты 0,0. Метод `drawString()` не распознает символы переноса строки. Следовательно, если требуется начать вывод текста с новой строки, то это придется сделать явно, указав точные координаты *X, Y* точки, с которой должен начинаться вывод новой строки текста. Как будет показано в последующих главах, имеются приемы, позволяющие облегчить этот процесс.

Для установки цвета фона в окне апплета служит метод `setBackground()`, а для установки цвета переднего плана (например, цвета выводимого на экран текста) — метод `setForeground()`. Эти методы определены в классе `Component` и имеют следующие общие формы:

```
void setBackground(Color новый_цвет)
void setForeground(Color новый_цвет)
```

где параметр *новый_цвет* обозначает устанавливаемый новый цвет. В классе `Color` определяются перечисленные ниже константы, которые могут быть использованы для указания цвета.

<code>Color.black</code>	<code>Color.magenta</code>
<code>Color.blue</code>	<code>Color.orange</code>
<code>Color.cyan</code>	<code>Color.pink</code>
<code>Color.darkGray</code>	<code>Color.red</code>
<code>Color.gray</code>	<code>Color.white</code>
<code>Color.green</code>	<code>Color.yellow</code>
<code>Color.lightGray</code>	

Определены также версии этих констант в верхнем регистре. В следующем примере кода устанавливается зеленый цвет фона и красный цвет текста:

```
setBackground(Color.green);
setForeground(Color.red);
```

Подходящим местом для установки цветов заднего плана (т.е. фона) и переднего плана является метод `init()`. Впрочем, во время выполнения апплета эти цвета можно изменять как угодно часто.

Для того чтобы получить текущие установки цветов переднего и заднего плана, достаточно вызвать методы `getBackground()` и `getForeground()` соответственно. Они также определены в классе `Component`, а их общие формы приведены ниже.

```
Color getBackground()
Color getForeground()
```

Рассмотрим пример очень простого апплета, где в качестве цвета фона устанавливается голубой, а в качестве цвета переднего плана — красный. Далее выводится сообщение, иллюстрирующее порядок вызова методов `init()`, `start()` и `paint()` при запуске апплета на выполнение. Ниже приведен исходный код из этого примера апплета.

```
/* Простой апплет, устанавливающий цвета фона
   и переднего плана и выводящий символьную строку
*/
import java.awt.*;
import java.applet.*;
/*
   <applet code="Sample" width=300 height=50>
   </applet>
*/
public class Sample extends Applet{
    String msg;

    // установить цвета фона и переднего плана
    public void init() {
        setBackground(Color.cyan);
        setForeground(Color.red);
        msg = "Inside init( ) --";
    }

    // инициализировать выводимую символьную строку
    public void start() {
        msg += " Inside start() --";
    }

    // вывести символьную строку msg в окне апплета
    public void paint(Graphics g) {
        msg += " Inside paint().";
        g.drawString(msg, 10, 30);
    }
}
```

Этот апплет создает окно, показанное на рис. 23.2. Методы `stop()` и `destroy()` не переопределены, поскольку в таком простом апплете это не требуется.

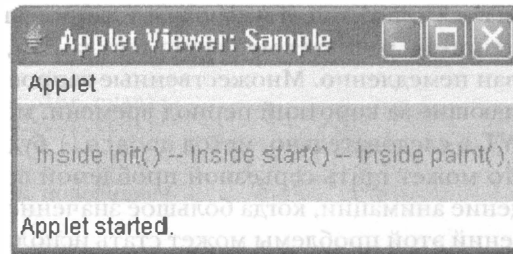


Рис. 23.2. Окно апплета, в котором выводится символьная строка

Запрос на повторное воспроизведение

Запомните главное правило: апплет выводит данные в свое окно только тогда, когда его метод `update()` или `paint()` вызывается из библиотеки AWT. В связи с этим возникает следующий интересный вопрос: каким образом апплет может инициировать собственное обновление, когда в нем изменяется информация? Так, если апплет выводит движущийся баннер (т.е. крупный заголовок), то какой механизм вынудит апплет обновлять окно на каждом шаге прокрутки этого баннера? Напомним, что одно из самых важных ограничений, накладываемых на апплет, состоит в том, что он должен быстро возвращать управление исполняющей системе. Например, апплет не может создавать в теле метода `paint()` циклы, в которых будет непрерывно прокручиваться баннер. Это помешало бы передаче управления обратно библиотеке AWT. Имея такое ограничение, можно подумать, что вывод данных в окно апплета, как минимум, существенно затруднен. К счастью, это не так. Всякий раз, когда апплет нуждается в обновлении отображаемой в его окне информации, он просто вызывает метод `repaint()`.

Метод `repaint()` определен в библиотеке AWT. Он вынуждает исполняющую систему библиотеки AWT осуществлять вызов метода `update()` из апплета, который в реализации по умолчанию обращается к методу `paint()`. Таким образом, чтобы вывод данных можно было выполнять из другой части апплета в его окно, достаточно сохранить выводимые данные и вызвать метод `repaint()`. Затем библиотека AWT выполнит вызов метода `paint()`, который может воспроизвести сохраненные данные. Так, если часть апплета нуждается в выводе символьной строки, ее можно сначала сохранить в переменной типа `String`, а затем вызвать метод `repaint()`. В методе `paint()` символьная строка выводится с помощью метода `drawString()`.

У метода `repaint()` имеются четыре общие формы. Рассмотрим их по порядку. В следующей общей форме определяется область, подлежащая повторному воспроизведению:

```
void repaint(int слева, int сверху, int ширина, int высота)
```

Здесь координаты правого верхнего угла области обозначаются параметрами *слева* и *сверху*, а ширина и высота области — параметрами *ширина* и *высота*. Эти размеры указаны в пикселях. Указывая область для повторного воспроизведения, можно сэкономить время, которое требуется для обновления окна. Если же требуется обновить только небольшую часть окна, то эффективнее будет обновить только эту конкретную область, а не всю площадь окна.

Вызов метода `repaint()`, по существу, означает запрос апплета на скорейшее обновление. Но если система работает медленно или занята, то метод `update()` может и не быть вызван немедленно. Множественные запросы на повторное воспроизведение, поступающие за короткий период времени, могут быть не обработаны библиотекой AWT, а следовательно, метод `update()` будет вызываться лишь время от времени. Это может стать серьезной проблемой во многих ситуациях, включая воспроизведение анимации, когда большое значение имеет время обновления. Одним из решений этой проблемы может стать использование следующих форм метода `repaint()`:

```
void repaint(long максимальная_задержка)
void repaint(long максимальная_задержка, int x, int y,
             int ширина, int высота)
```

где параметр *максимальная_задержка* обозначает максимальное количество миллисекунд, которые могут пройти, прежде чем будет вызван метод `update()`. Следует, однако, иметь в виду, что если время истечет прежде, чем системе удастся вызвать метод `update()`, этот метод не будет вызван. Никакого возвращаемого значения или передаваемого исключения не появится, поэтому следует соблюдать особую осторожность.

На заметку! Существует возможность вывода данных в окно апплета другими методами, кроме `paint()` или `update()`. Для вывода в окно такой метод должен получить графический контекст, вызвав метод `getGraphics()`, определенный в классе `Component`, а затем использовать этот контекст для вывода данных в окно. Но для большинства приложений лучше и проще направлять вывод данных в окно через метод `paint()` и вызывать метод `repaint()`, когда изменяется содержимое окна.

Простой апплет с баннером

Чтобы продемонстрировать метод `repaint()`, обратимся к простому примеру апплета, воспроизводящего баннер. Этот апплет должен прокручивать выводимое в виде баннера сообщение слева направо в своем окне. Прокрутка такого сообщения поперек окна является повторяющейся операцией, поэтому она должна выполняться в отдельном потоке, создаваемом при инициализации апплета. Ниже приведен исходный код апплета, воспроизводящего баннер.

```
/* Простой апплет, воспроизводящий баннер
   Этот апплет создает поток исполнения, прокручивающий
   сообщение, содержащееся в переменной msg, справа
   налево в окне апплета.
*/
import java.awt.*;
import java.applet.*;
/*
   <applet code="SimpleBanner" width=300 height=50>
   </applet>
*/

public class SimpleBanner extends Applet implements Runnable {
    String msg = " A Simple Moving Banner."; // Простой движущийся баннер
    Thread t = null;
    int state;
```

```

volatile boolean stopFlag;

// установить цвета и инициализировать поток исполнения
public void init() {
    setBackground(Color.cyan);
    setForeground(Color.red);
}

// запустить поток исполнения
public void start() {
    t = new Thread(this);
    stopFlag = false;
    t.start();
}

// точка входа в поток исполнения, прокручивающий баннер
public void run() {

    // воспроизвести баннер повторно
    for( ; ; ) {
        try {
            repaint();
            Thread.sleep(250);
            if(stopFlag)
                break;
        } catch (InterruptedException e) {}
    }
    // приостановить воспроизведение баннера
    public void stop() {
        stopFlag = true;
        t = null;
    }
    // воспроизвести баннер
    public void paint(Graphics g) {
        char ch;

        ch = msg.charAt(0);
        msg = msg.substring(1, msg.length());
        msg += ch;

        g.drawString(msg, 50, 30);
    }
}

```

На рис. 23.3 показано окно этого апплета, воспроизводящего баннер.

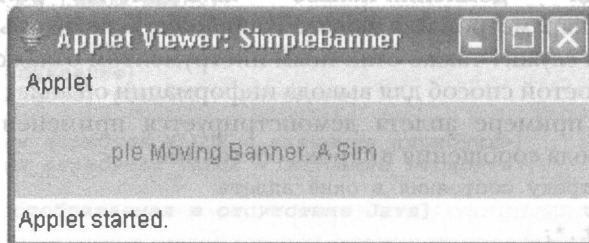


Рис. 23.3. Окно апплета с прокруткой сообщения в виде баннера

Рассмотрим подробнее, каким образом действует этот апплет. Прежде всего обратите внимание на то, что класс SimpleBanner расширяет класс Applet, как и следовало ожидать. Но, кроме того, он реализует интерфейс Runnable. Это необходимо

для создания в апплете второго потока исполнения, в котором должен прокручиваться баннер. В методе `init()` устанавливаются цвета фона и переднего плана апплета.

После инициализации исполняющая система вызывает метод `start()` для запуска апплета на выполнение. В методе `start()` создается новый поток исполнения, который присваивается переменной `t` типа `Thread`. Затем переменной `stopFlag` типа `boolean`, которая управляет выполнением апплета, присваивается логическое значение `false`. После этого вызывается метод `t.start()` для запуска потока исполнения. Напомним, что по ссылке `t.start()` вызывается метод `start()`, определенный в классе `Thread`, а следовательно, начинает выполняться метод `run()`. И это не приводит к вызову варианта метода `start()`, определенно-го в классе `Applet`. Ведь это два отдельных метода.

В методе `run()` вызывается метод `repaint()`. Это, в свою очередь, приводит к вызову метода `paint()`, а в конечном итоге — к прокрутке содержимого переменной `msg`. В промежутке между последовательными шагами цикла выполнение метода `run()` приостанавливается на четверть секунды. Таким образом, содержимое переменной `msg` прокручивается слева направо в непрерывном движении. Переменная `stopFlag` проверяется на каждом шаге цикла. Когда она принимает логическое значение `true`, выполнение цикла в частности и метода `run()` вообще прерывается.

Если браузер воспроизводит апплет во время просмотра новой страницы, вызывается метод `stop()`, в котором переменной `stopFlag` присваивается логическое значение `true`, тем самым прерывая выполнение метода `run()`. Этот механизм служит для остановки потока исполнения, когда страница больше не просматривается. Когда же апплет снова становится видимым, опять вызывается его метод `start()`, который запускает новый поток исполнения для прокрутки баннера.

Применение строки состояния

Кроме воспроизведения информации в своем окне, апплет может также выводить сообщения в строке состояния браузера или средства просмотра апплетов, в котором он запускается. Для этого достаточно вызвать метод `showStatus()` со строкой сообщения, которое требуется вывести. Строка состояния служит удобным местом для уведомления пользователя о том, что происходит в апплете, для отображения параметров или вывода сообщения о некоторых видах ошибок. Строка состояния служит также отличным инструментом отладки, поскольку она предоставляет простой способ для вывода информации об апплете.

В следующем примере апплета демонстрируется применение метода `showStatus()` для вывода сообщения в строке состояния:

```
// Использовать строку состояния в окне апплета
import java.awt.*;
import java.applet.*;
/*
  <applet code="StatusWindow" width=300 height=50>
  </applet>
*/

public class StatusWindow extends Applet {
    public void init() {
        setBackground(Color.cyan);
    }
}
```

```

}

// вывести сообщение в окне апплета и в строке состояния
public void paint(Graphics g) {
    g.drawString("This is in the applet window.", 10, 20);
    showStatus("This is shown in the status window.");
}
}

```

Окно апплета с выводимыми сообщениями показано на рис. 23.4.

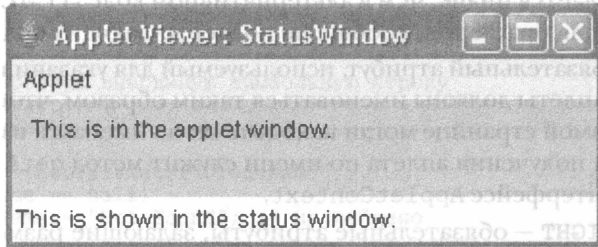


Рис. 23.4. Окно апплета со строкой состояния

HTML-дескриптор APPLET

Как упоминалось ранее, на момент написания данной книги компания Oracle рекомендовала употреблять дескриптор APPLET для запуска апплета вручную, если сетевой протокол JNLP не применяется. Средство просмотра апплетов выполнит каждый дескриптор APPLET, который оно обнаружит, в отдельном окне, в то время как веб-браузеры позволяют выполнять многие апплеты на одной странице. В рассмотренных до сих пор примерах апплетов употреблялась лишь упрощенная форма дескриптора APPLET. А теперь пришло время рассмотреть его более основательно.

Ниже приведен синтаксис полной формы дескриптора APPLET. Элементы, заключенные в квадратные скобки, являются необязательными.

```

<APPLET
  [CODEBASE = URL_кодировой_базы]
  CODE = файл_апплета
  [ALT = альтернативный_текст]
  [NAME = имя_экземпляра_апплета]
  WIDTH = пикселей HEIGHT = пикселей
  [ALIGN = выравнивание]
  [VSPACE = пикселей] [HSPACE = пикселей]
>
[<PARAM NAME = имя_атрибута VALUE = значение_атрибута>]
[<PARAM NAME = имя_атрибута2 VALUE = значение_атрибута2>]
...
[HTML-разметка, отображаемая в отсутствие Java]
</APPLET>

```

Рассмотрим каждую составляющую этого дескриптора по отдельности.

- **CODEBASE** — необязательный атрибут, задающий URL кодовой базы апплета, т.е. каталог для поиска исполняемого файла класса, обозначаемого дескриптором CODE. Если атрибут CODEBASE не задан явно, в качестве этого атрибута используется URL, указывающий на каталог текущего HTML-документа.

- **CODE** — обязательный атрибут, задающий имя файла с расширением `.class`, содержащего скомпилированный код апплета. Имя этого файла задается относительно URL кодовой базы апплета, т.е. того каталога, где находится текущий HTML-документ, или каталога, указанного в атрибуте `CODEBASE`.
- **ALT** — необязательный атрибут, используемый для указания краткого текстового сообщения, которое должно быть отображено, если браузер распознает дескриптор `APPLET`, но в данный момент не сможет выполнять апплеты Java. Это делается иначе, чем в альтернативном коде HTML, который представляется для браузеров, вообще не поддерживающих апплеты.
- **NAME** — необязательный атрибут, используемый для указания имени экземпляра апплета. Апплеты должны именоваться таким образом, чтобы другие апплеты на той же самой странице могли находить их по именам и взаимодействовать с ними. Для получения апплета по имени служит метод `getApplet()`, определяемый в интерфейсе `AppletContext`.
- **WIDTH** и **HEIGHT** — обязательные атрибуты, задающие размеры (в пикселях) отображаемой области апплета.
- **ALIGN** — необязательный атрибут, задающий выравнивание апплета. Этот атрибут трактуется точно так же, как и в HTML-дескрипторе `IMG`. Он имеет следующие значения: `LEFT`, `RIGHT`, `TOP`, `BOTTOM`, `MIDDLE`, `BASELINE`, `TEXTTOP`, `ABSMIDDLE` и `ABSBOTTOM`.
- **VSPACE** и **HSPACE** — необязательные атрибуты. Атрибут `VSPACE` определяет пространство в пикселях выше и ниже области, занимаемой апплетом, тогда как атрибут `HSPACE` — пространство в пикселях по бокам от области, занимаемой апплетом. Они интерпретируются таким же образом, как и атрибуты `VSPACE` и `HSPACE` дескриптора `IMG`.
- **PARAM NAME** и **VALUE** — дескриптор `PARAM` позволяет указать характерные для апплета аргументы. Апплеты получают доступ к этим атрибутам с помощью метода `getParameter()`.

Среди прочих допустимых атрибутов следует также упомянуть атрибут `ARCHIVE`, позволяющий задать один или несколько архивных файлов, а также атрибут `OBJECT`, обозначающий сохраняемую версию апплета. В общем, дескриптор `APPLET` должен включать только атрибут `CODE` или `OBJECT`, но не оба атрибута сразу.

Передача параметров апплетам

Как упоминалось выше, HTML-дескриптор `APPLET` позволяет передавать параметры апплету. Для извлечения параметра служит метод `getParameter()`. Он возвращает значение указанного параметра в форме символьной строки. Таким образом, строковые представления числовых и логических значений придется преобразовывать во внутренние форматы. Ниже приведен пример, демонстрирующий передачу параметров апплету.

```
// Использовать параметры, передаваемые апплету
import java.awt.*;
import java.applet.*;
/*
```

```
<applet code="ParamDemo" width=300 height=80>
<param name=fontName value=Courier>
<param name=fontSize value=14>
<param name=leading value=2>
<param name=accountEnabled value=true>
</applet>
*/

public class ParamDemo extends Applet {
    String fontName;
    int fontSize;
    float leading;
    boolean active;

    // инициализировать выводимую символьную строку
    public void start() {
        String param;

        fontName = getParameter("fontName");
        if(fontName == null)
            fontName = "Not Found"; // не найдено

        param = getParameter("fontSize");
        try {
            if(param != null)
                fontSize = Integer.parseInt(param);
            else
                fontSize = 0;
        } catch(NumberFormatException e) {
            fontSize = -1;
        }

        param = getParameter("leading");
        try {
            if(param != null)
                leading = Float.valueOf(param).floatValue();
            else
                leading = 0;
        } catch(NumberFormatException e) {
            leading = -1;
        }

        param = getParameter("accountEnabled");
        if(param != null)
            active = Boolean.valueOf(param).booleanValue();
    }

    // вывести параметры
    public void paint(Graphics g) {
        g.drawString("Font name: " + fontName, 0, 10); // шрифт
        g.drawString("Font size: " + fontSize, 0, 26); // кегль
        g.drawString("Leading: " + leading, 0, 42); // интерлиньяж
        g.drawString("Account Active: " + active, 0, 58); // активная
        // учетная запись
    }
}
```

Пример вывода параметров в окне данного апплета показан на рис. 23.5.

Как следует из приведенного выше примера апплета, следует проверять значения, возвращаемые методом `getParameter()`. Если параметр недоступен, метод `getParameter()` возвратит пустое значение `null`. Должна быть также предпринята попытка преобразования в числовые типы в блоке оператора `try`, где пере-


```

        msg = " " + msg;
        t = new Thread(this);
        stopFlag = false;
        t.start();
    }

    // точка входа в поток исполнения, воспроизводящего баннер
    public void run() {
        // воспроизвести баннер повторно
        for( ; ; ) {
            try {
                repaint();
                Thread.sleep(250);
                if(stopFlag)
                    break;
            } catch(InterruptedException e) {}
        }
    }

    // приостановить воспроизведение баннера
    public void stop() {
        stopFlag = true;
        t = null;
    }

    // воспроизвести баннер
    public void paint(Graphics g) {
        char ch;

        ch = msg.charAt(0);
        msg = msg.substring(1, msg.length());
        msg += ch;

        g.drawString(msg, 50, 30);
    }
}

```

Методы `getDocumentBase()` и `getCodeBase()`

Нередко приходится создавать апплеты, которые должны явным образом загружать мультимедийные данные и текст. Апплетам в Java разрешается загружать данные из каталога, содержащего HTML-файл, который запускает апплет (*базу документов*), а также из каталога, из которого загружается класс апплета (*кодovou базу*). Эти каталоги возвращаются как объекты класса `URL` (см. главу 22) из методов `getDocumentBase()` и `getCodeBase()`. Они могут быть соединены с символьной строкой, содержащей имя файла, который требуется загрузить. Для того чтобы загрузить другой файл, достаточно вызвать метод `showDocument()`, определяемый в интерфейсе `AppletContext`, который рассматривается в следующем разделе. Применение рассматриваемых здесь методов демонстрируется в приведенном ниже примере апплета.

```

// Отобразить базу документов и кодовую базу
import java.awt.*;
import java.applet.*;
import java.net.*;
/*
<applet code="Bases" width=300 height=50>

```

```

</applet>
*/

public class Bases extends Applet {
    // вывести в окне апплета базу документов и кодовую базу
    public void paint(Graphics g) {
        String msg;

        URL url = getCodeBase(); // получить кодовую базу
        msg = "Code base: " + url.toString(); // кодовая база
        g.drawString(msg, 10, 20);

        url = getDocumentBase(); // получить базу документов
        msg = "Document base: " + url.toString(); // база документа
        g.drawString(msg, 10, 40);
    }
}

```

Примерный результат, выводимый при выполнении этого апплета, показан на рис. 23.6.

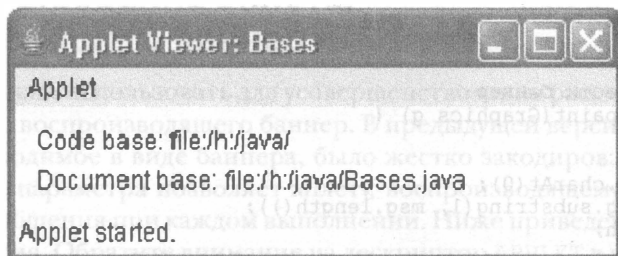


Рис. 23.6. Окно апплета, отображающего базу документов и кодовую базу

Интерфейс `AppletContext` и метод `showDocument()`

Одной из областей приложения Java является применение активных изображений и анимации для обеспечения графических средств навигации в веб как более интересный вариант, чем простые текстовые ссылки. Чтобы предоставить апплету возможность передавать управление другому URL, следует вызвать метод `showDocument()`, определяемый в интерфейсе `AppletContext`. Интерфейс `AppletContext` позволяет получать данные из исполняющей среды апплета. Методы, определяемые в интерфейсе `AppletContext`, перечислены в табл. 23.2. Контекст апплета, выполняемого в текущий момент, можно получить, вызвав метод `getAppletContext()`, определяемый в классе `Applet`.

Как только будет получен контекст апплета, можно сразу же отобразить другой документ, вызвав метод `showDocument()`. Этот метод не возвращает никакого значения и не генерирует никаких исключений при неудачном исходе своего выполнения, поэтому пользоваться им следует осторожно. Имеются две разновидности метода `showDocument()`. В частности, метод `showDocument(URL)` отображает документ, находящийся по указанному URL, а метод `showDocument(URL, String)` — до-

кумент, находящийся в указанном месте окна браузера. Достоверными значениями параметра *где* второй разновидности метода `showDocument()` являются `"_self"` (отобразить в текущем фрейме), `"_parent"` (отобразить в родительском фрейме), `"_top"` (отобразить в самом верхнем фрейме) и `"_blank"` (отобразить в новом окне браузера). Имеется также возможность задать имя документа, чтобы отобразить документ по его имени в новом окне браузера.

Таблица 23.2. Методы из интерфейса `AppletContext`

Метод	Описание
Applet <code>getApplet(String имя_аплета)</code>	Возвращает апплет по указанному <i>имени_аплета</i> , если он находится в контексте текущего апплета, а иначе — пустое значение <code>null</code>
Enumeration < Applet > <code>getApplets()</code>	Возвращает перечисление, содержащее все апплеты из контекста текущего апплета
AudioClip <code>getAudioClip(URL url)</code>	Возвращает объект типа AudioClip , инкапсулирующий аудиоклип, находящийся в месте, обозначаемом параметром <i>url</i>
Image <code>getImage(URL url)</code>	Возвращает объект типа Image , инкапсулирующий графическое изображение, находящееся в месте, обозначаемом параметром <i>url</i>
InputStream <code>getStream(String ключ)</code>	Возвращает поток ввода, с которым связан заданный <i>ключ</i> . Ключи привязываются к потокам ввода-вывода методом <code>setStream()</code> . Если же поток ввода, с которым связан заданный <i>ключ</i> , отсутствует, то возвращается пустое значение <code>null</code>
Iterator < String > <code>getStreamKeys()</code>	Возвращает итератор для ключей, связанных с вызывающим объектом. Ключи связаны с потоками ввода-вывода. См. также методы <code>getStream()</code> и <code>setStream()</code>
void <code>setStream(String ключ, InputStream поток)</code> throws IOException	Связывает указанный <i>поток</i> ввода с заданным <i>ключом</i> . Ключ удаляется из вызывающего объекта, если параметр <i>поток</i> принимает пустое значение <code>null</code>
void <code>showDocument(URL url)</code>	Отображает документ по URL, обозначаемому параметром <i>url</i> . Этот метод может не поддерживаться средствами просмотра апплетов
void <code>showDocument(URL url, String где)</code>	Отображает документ по URL, обозначаемому параметром <i>url</i> . Этот метод может не поддерживаться средствами просмотра апплетов. Расположение документа определяется параметром <i>где</i> , как описано далее
void <code>showStatus(String строка)</code>	Отображает содержимое указанной <i>строки</i> в строке состояния

В приведенном ниже примере апплета демонстрируется применение интерфейса `AppletContext` и метода `showDocument()`. Когда этот апплет выполняется, он получает контекст текущего апплета и использует его для передачи управления файлу `Test.html`. Этот файл должен находиться в том же каталоге, что и апплет. Файл `Test.html` может содержать любой допустимый гипертекст.

```

/* Использование контекста апплета, методов getCodeBase()
и showDocument() для отображения HTML-документа
*/

import java.awt.*;
import java.applet.*;
import java.net.*;
/*
    <applet code="ACDemo" width=300 height=50>
    </applet>
*/

public class ACDemo extends Applet {
    public void start() {
        AppletContext ac = getAppletContext();
        URL url = getCodeBase(); // получить URL данного апплета

        try {
            ac.showDocument(new URL(url+"Test.html"));
        } catch (MalformedURLException e) {
            showStatus("URL not found"); // URL не найден
        }
    }
}

```

Интерфейс AudioClip

В интерфейсе `AudioClip` определяются следующие три метода: `play()` (воспроизведение аудиоклипа с начала), `stop()` (остановка воспроизведения) и `loop()` (циклическое воспроизведение). После загрузки аудиоклипа с методом `getAudioClip()` эти методы можно применять для его воспроизведения.

Интерфейс AppletStub

Интерфейс `AppletStub` предоставляет средства, с помощью которых взаимодействуют апплет и браузер (или средство просмотра апплетов). Как правило, этот интерфейс не реализуется в прикладном коде.

Консольный вывод

Несмотря на то что вывод данных в окно апплета должен осуществляться методами ГПИ, например `drawString()`, в апплетах можно организовать и консольный вывод, в частности, для целей отладки. Когда в апплете вызывается такой метод, как `System.out.println()`, выводимые им данные не направляются в окно апплета. Вместо этого они выводятся на консоль, из которой было запущено средство просмотра апплетов, или же на консоль Java, доступную в некоторых браузерах. Пользоваться консольным выводом в других целях, кроме отладки, не рекомендуется, поскольку он нарушает принципы графического оформления пользовательского интерфейса, к которым уже привыкли большинство пользователей апплетов.

Эта глава посвящена событиям — важнейшему аспекту Java. Обработка событий имеет решающее значение для всего программирования на Java в целом, поскольку она является неотъемлемой частью разработки апплетов и прочих видов прикладных программ с *графическим пользовательским интерфейсом* (ГПИ). Как упоминалось в главе 23, апплеты представляют собой управляемые событиями прикладные программы, использующие ГПИ для взаимодействия с пользователем. Более того, любая прикладная программа с ГПИ, например, написанная на Java для Windows, выполняется под управлением событий. Иными словами, такую программу нельзя написать не имея ясного представления об обработке событий. События поддерживаются в целом ряде пакетов, включая `java.util`, `java.awt` и `java.event`.

Большинство событий, на которые будет реагировать прикладная программа с ГПИ, происходят при взаимодействии пользователя с этой программой. Именно события такого рода и рассматриваются в этой главе. События передаются прикладной программе самыми разными способами, каждый из которых зависит от конкретного события. Существует несколько типов событий, включая генерируемые мышью, клавиатурой и различными элементами управления ГПИ, в том числе кнопками, полосами прокрутки или флажками.

Эта глава начинается с обзора механизма управления событиями в Java. Затем рассматриваются основные классы и интерфейсы событий, используемые библиотекой AWT, а также ряд примеров, демонстрирующих основы обработки событий. В этой главе поясняется также, как пользоваться классами адаптеров, внутренними и анонимными вложенными классами, чтобы упростить код обработки событий. Рассматриваемые здесь приемы будут использоваться в примерах, приведенных далее в данной книге.

На заметку! В этой главе основное внимание уделяется событиям, имеющим отношение к программам с ГПИ. А иногда события также используются в целях, не имеющих прямого отношения к подобного рода программам. Но во всех случаях применяются одни и те же приемы обработки событий.

Два механизма обработки событий

Прежде чем приступить к обсуждению обработки событий, нужно сделать одно важное замечание. Способ обработки событий существенно изменился с момента

выпуска первоначальной версии Java 1.0 до появления более современных версий, начиная с Java 1.1. Способ обработки событий, принятый в версии Java 1.0, поддерживается до сих пор, хотя и не рекомендован для употребления в новых программах. К тому же многие методы, поддерживающие старую модель обработки событий из версии Java 1.0, теперь объявлены не рекомендованными к употреблению. Современный подход состоит в том, что события должны обрабатываться в новых программах так, как описано в данной книге.

Модель делегирования событий

Современный подход к обработке событий основан на модели делегирования *событий*, определяющей стандартные и согласованные механизмы для генерирования и обработки событий. Принцип действия этой модели довольно прост: *источник* генерирует событие и извещает о нем один или несколько *приемников*. В этой модели приемник просто ожидает до тех пор, пока не получит извещение о событии. Как только извещение о событии получено, приемник обрабатывает его и возвращает управление. Преимущество такой модели заключается в том, что логика прикладной программы, обрабатывающей события, четко отделена от логики пользовательского интерфейса, извещающего об этом событии. Элемент пользовательского интерфейса может делегировать обработку события отдельному фрагменту кода.

В модели делегирования событий приемники должны регистрироваться источником, чтобы получать извещения о событиях. Это дает следующее важное преимущество: уведомления посылаются только тем приемникам, которым требуется их получать. Это более эффективный способ обработки событий, чем тот, что был принят в первоначальной модели из версии Java 1.0. Раньше извещение о событии распространялось по всей иерархии вложенности до тех пор, пока оно не было обработано каким-нибудь компонентом. Это вынуждало все компоненты получать извещения о событиях, которые они могли и не обрабатывать, что приводило к напрасной трате времени. Модель делегирования событий исключила подобную расточительность.

В последующих разделах рассматриваются отдельные события и поясняется роль источников и приемников в обработке событий.

События

В рассматриваемой здесь модели делегирования *событие* представляет собой объект, описывающий изменение состояния источника. Этот объект может быть создан в результате взаимодействия пользователя с элементом управления ГПИ. К событиям приводят такие действия, как щелчок на экранной кнопке, ввод символа с клавиатуры, выбор элемента из списка и щелчок кнопкой мыши. Имеется немало и других пользовательских операций, которые могут служить примерами для наступления событий.

События могут происходить не только в результате прямого взаимодействия с пользовательским интерфейсом. Например, событие может произойти по истечении времени срабатывания таймера, а также в результате превышения счетчи-

ком некоторого значения, программного или аппаратного сбоя или завершения некоторой операции. Имеется возможность определять и собственные события, отвечающие характеру прикладной программы.

Источники событий

Источник — это объект, генерирующий событие. Событие происходит при изменении некоторым образом внутреннего состояния объекта. Источники могут генерировать события нескольких типов.

Приемники должны быть зарегистрированы в источнике, чтобы получать извещения о событиях соответствующего типа. У события каждого типа имеется свой метод регистрации. Его общая форма выглядит так:

```
public void addТипListener(ТипListener приемник_событий)
```

где *Тип* обозначает имя объекта события, а *приемник_событий* — ссылку на конкретный приемник событий. Например, метод, регистрирующий приемник событий от клавиатуры, называется `addKeyListener()`, а метод, регистрирующий приемник событий от перемещения мыши, — `addMouseMotionListener()`. Когда наступает событие, все зарегистрированные приемники получают копию объекта события. Этот процесс называется *групповой рассылкой* событий. Но в любом случае уведомления отправляются только тем приемникам, которые зарегистрированы на их получение.

Некоторые источники допускают регистрацию только одного приемника. Ниже приведена общая форма такого метода.

```
public void addТипListener(ТипListener приемник_событий)  
    throws java.util.TooManyListenersException
```

где *Тип* обозначает имя объекта события, а *приемник_событий* — ссылку на конкретный приемник событий. Когда такое событие наступает, зарегистрированный приемник получает уведомление об этом событие. Этот процесс называется *целевой рассылкой* событий.

Источник должен также предоставлять метод, позволяющий снять приемник с регистрации определенного типа событий. Общая форма этого метода следующая:

```
public void removeТипListener(ТипListener приемник_событий)
```

Здесь *Тип* обозначает имя объекта события, а *приемник_событий* — ссылку на конкретный приемник событий. Например, чтобы снять с регистрации приемник событий от клавиатуры, следует вызвать метод `removeKeyListener()`.

Методы, которые добавляют или удаляют регистрируемые источники событий, предоставляются источником, генерирующим событие. Например, в классе `Component` предоставляются методы для добавления и удаления регистрируемых приемников событий от клавиатуры и мыши.

Приемники событий

Приемник — это объект, уведомляемый о событии. К нему предъявляются два основных требования. Во-первых, он должен быть зарегистрирован одним или не-

сколькими источниками событий, чтобы получать уведомления о событиях определенного типа. И во-вторых, он должен реализовать методы для получения и обработки таких уведомлений.

Методы, принимающие и обрабатывающие события, определены в ряде интерфейсов, входящих в состав пакета `java.awt.event`. Например, в интерфейсе `MouseMotionListener` определяются два метода для получения уведомлений о перетаскивании объекта или перемещении мыши. Любой объект может принимать и обрабатывать одно или оба этих события, если его класс предоставляет реализацию данного интерфейса. В этой и последующих главах речь еще не раз пойдет о данном и ряде других интерфейсов приемников событий.

Классы событий

Классы, представляющие события, находятся в самой сердцевине механизма обработки событий в Java. Поэтому обсуждение обработки событий должно начинаться с классов событий. Следует, однако, иметь в виду, что в Java определяется несколько типов событий и что не все классы событий будут упомянуты в этой главе. Наиболее широко употребительными являются те события, которые определены в библиотеках AWT и Swing. В этой главе основное внимание уделяется событиям из библиотеки AWT. (Многие из них относятся также и к библиотеке Swing.) Несколько характерных для библиотеки Swing событий будут описаны в главе 31, когда речь пойдет о библиотеке Swing.

В основу иерархии классов событий в Java положен класс `EventObject`, входящий в пакет `java.util`. Он служит суперклассом для всех событий. Ниже приведен единственный конструктор этого класса, где *источник* обозначает объект, сгенерировавший событие.

`EventObject(Object источник)`

Класс `EventObject` содержит два метода: `getSource()` и `toString()`. Метод `getSource()` возвращает источник событий. Ниже приведена его общая форма. Как и следовало ожидать, метод `toString()` возвращает символьную строку, которая служит эквивалентом события.

`Object getSource()`

Класс `AWTEvent`, определенный в пакете `java.awt`, является производным от класса `EventObject`. Кроме того, класс `AWTEvent` является (прямо или косвенно) суперклассом для всех событий из библиотеки AWT, применяемых в модели делегирования событий. С помощью его метода `getID()` можно определить тип события. Его общая форма приведена ниже.

`int getID()`

Классы всех остальных событий из библиотеки AWT являются производными от класса `AWTEvent`. Итак, подводя итог, можно сказать следующее.

- Класс `EventObject` служит суперклассом для классов всех событий.
- Класс `AWTEvent` служит суперклассом для классов всех событий из библиотеки AWT, обрабатываемых по модели делегирования событий.

В пакете `java.awt.event` определяются многие типы событий, генерируемых различными элементами пользовательского интерфейса. В табл. 24.1 перечислены наиболее употребительные классы событий вместе с кратким описанием причин, по которым они наступают. Наиболее употребительные конструкторы и методы каждого из этих классов описаны в последующих разделах.

Таблица 24.1. Основные классы событий из пакета `java.awt.event`

Класс события	Описание
ActionEvent	Наступает после щелчка на кнопке, двойного щелчка на элементе списка или выбора пункта меню
AdjustmentEvent	Наступает при манипулировании полосой прокрутки
ComponentEvent	Наступает, когда компонент скрывается, перемещается, изменяет свои размеры или становится доступным
ContainerEvent	Наступает после добавления или удаления компонента из контейнера
FocusEvent	Наступает, когда компонент получает или теряет фокус ввода с клавиатуры
InputEvent	Абстрактный суперкласс для всех классов событий, связанных с вводом данных в компонентах
ItemEvent	Наступает после щелчка на флажке или элементе списка, а также на отмечаемом пункте меню
KeyEvent	Наступает при получении данных, введенных с клавиатуры
MouseEvent	Наступает при перетаскивании, перемещении, щелчках, нажатии и отпуске кнопок мыши, а также в том случае, когда курсор мыши наводится на компонент или отводится от него
MouseWheelEvent	Наступает при прокрутке колесика мыши
TextEvent	Наступает при изменении значения в текстовой области или текстовом поле
WindowEvent	Наступает, когда окно активизируется, деактивизируется, сворачивается, разворачивается или закрывается

Класс `ActionEvent`

Событие типа `ActionEvent` генерируется после щелчка на кнопке, двойного щелчка на элементе списка или выбора пункта меню. В классе `ActionEvent` определяются следующие целочисленные константы, которые могут быть использованы для идентификации любых модификаторов, связанных с событием действия: `ALT_MASK`, `CTRL_MASK`, `META_MASK`, `SHIFT_MASK` и `ACTION_PERFORMED`.

В классе `ActionEvent` имеются следующие конструкторы:

```

ActionEvent(Object источник, int тип, String команда)
ActionEvent(Object источник, int тип, String команда,
              int модификаторы)
ActionEvent(Object источник, int тип, String команда,
              long момент, int модификаторы)

```

где параметр *источник* обозначает ссылку на объект, сгенерировавший событие; параметр *тип* — конкретный тип события; параметр *команда* — командную строку;

параметр *момент* — конкретный момент наступления события. Кроме того, параметр *модификаторы* обозначает нажатие модифицирующих клавиш (<Alt>, <Ctrl>, <Meta> и/или <Shift>) в момент наступления события.

После вызова метода `getActionCommand()` можно получить имя команды для вызывающего объекта типа `ActionEvent`. Ниже приведена общая форма этого метода.

```
String getActionCommand()
```

Например, когда производится щелчок на кнопке, наступает событие действия, которое имеет имя команды, соответствующее метке данной кнопки.

Метод `getModifiers()` возвращает значение, обозначающее нажатие модифицирующих клавиш (<Alt>, <Ctrl>, <Meta> и/или <Shift>) в момент события. Ниже приведена общая форма этого метода.

```
int getModifiers()
```

Метод `getWhen()` возвращает момент наступления события, называемый *временной меткой* события. Общая форма этого метода выглядит следующим образом:

```
long getWhen()
```

Класс `AdjustmentEvent`

Событие класса `AdjustmentEvent` генерируется полосой прокрутки. Имеется пять типов событий настройки. В классе `AdjustmentEvent` определяются целочисленные константы, которые могут использоваться для идентификации событий. Константы и их описание представлены в табл. 24.2.

Таблица 24.2. Константы, определенные в классе `AdjustmentEvent`

Константа	Описание
<code>BLOCK_DECREMENT</code>	Пользователь щелкнул на полосе прокрутки для уменьшения прокручиваемого значения
<code>BLOCK_INCREMENT</code>	Пользователь щелкнул на полосе прокрутки для увеличения прокручиваемого значения
<code>TRACK</code>	Перемещен ползунок
<code>UNIT_DECREMENT</code>	Произведен щелчок на кнопке в конце полосы для уменьшения прокручиваемого значения
<code>UNIT_INCREMENT</code>	Произведен щелчок на кнопке в конце полосы для увеличения прокручиваемого значения

Имеется также целочисленная константа `ADJUSTMENT_VALUE_CHANGED`, обозначающая, что произошло изменение.

Ниже приведен единственный конструктор класса `AdjustmentEvent`, где *источник* обозначает ссылку на объект, сгенерировавший событие; *идентификатор* — событие настройки; *тип* — конкретный тип события настройки; *значение* — связанное с этим событием значение.

```
AdjustmentEvent(Adjustable источник, int идентификатор,
                int тип, int значение)
```

Метод `getAdjustable()` возвращает объект, сгенерировавший событие. Ниже приведена его общая форма.

```
Adjustable getAdjustable()
```

Тип события настройки можно получить, вызвав метод `getAdjustmentType()`. Он возвращает одну из констант, определенных в классе `AdjustmentEvent`. Его общая форма следующая:

```
int getAdjustmentType()
```

Величину настройки можно получить, вызвав метод `getValue()`. Ниже приведена его общая форма.

```
int getValue()
```

Например, когда выполняется манипулирование полосой прокрутки, этот метод возвращает значение, обозначающее произведенное изменение.

Класс `ComponentEvent`

Объект класса `ComponentEvent` создается при изменении размеров, положения или видимости компонента. Имеются четыре типа событий в компонентах. Для их идентификации в классе `ComponentEvent` определяются целочисленные константы, представленные в табл. 24.3.

Таблица 24.3. Константы, определенные в классе `ComponentEvent`

Константы	Описание
<code>COMPONENT_HIDDEN</code>	Компонент скрыт
<code>COMPONENT_MOVED</code>	Компонент перемещен
<code>COMPONENT_RESIZED</code>	Изменен размер компонента
<code>COMPONENT_SHOWN</code>	Компонент стал видимым

В классе `ComponentEvent` имеется следующий конструктор:

```
ComponentEvent(Component источник, int тип)
```

где параметр *источник* обозначает ссылку на объект, сгенерировавший событие, а параметр *тип* — конкретный тип события. Класс `ComponentEvent` служит (прямо или косвенно) суперклассом для классов `ContainerEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent`, `WindowEvent` и др.

Метод `getComponent()` возвращает компонент, сгенерировавший событие. Его общая форма выглядит следующим образом:

```
Component getComponent()
```

Класс `ContainerEvent`

Объект класса `ContainerEvent` создается при добавлении компонента в контейнер или его удалении оттуда. Имеются два типа событий в контейнере. В классе `ContainerEvent` определяются следующие целочисленные константы, кото-

рые могут использоваться для идентификации этих событий: `COMPONENT_ADDED` и `COMPONENT_REMOVED`. Они определяют, введен ли компонент в контейнер или же удален из него.

Класс `ContainerEvent` является производным от класса `ComponentEvent` и имеет следующий конструктор:

```
ContainerEvent(Component источник, int тип, Component компонент)
```

где параметр *источник* обозначает ссылку на контейнер, который уведомляет о наступившем событии, параметр *тип* — конкретный тип события, а параметр *компонент* — вводимый в контейнер или удаляемый из него компонент.

Вызвав метод `getContainer()`, можно получить ссылку на контейнер, создавший это событие. Ниже приведена общая форма этого метода.

```
Container getContainer()
```

Метод `getChild()` возвращает ссылку на компонент, который был добавлен или удален из контейнера. Его общая форма следующая:

```
Component getChild()
```

Класс `FocusEvent`

Событие типа `FocusEvent` генерируется при получении или потере компонентом фокуса ввода. Такие события определяются целочисленными константами `FOCUS_GAINED` и `FOCUS_LOST`.

Класс `FocusEvent` является производным от класса `ComponentEvent` и имеет следующие конструкторы:

```
FocusEvent(Component источник, int тип)  
FocusEvent(Component источник, int тип, boolean временный_признак)  
FocusEvent(Component источник, int тип, boolean временный_признак,  
            Component другое)
```

где параметр *источник* обозначает ссылку на компонент, сгенерировавший событие, а параметр *тип* — конкретный тип события. Если событие фокуса ввода оказывается временным, то параметр *временный_признак* должен принимать логическое значение `true`, а иначе — логическое значение `false`. (Событие временного фокуса ввода происходит в результате другой операции в пользовательском интерфейсе. Допустим, фокус ввода находится на текстовом поле. Если пользователь передвигает мышь, чтобы переместить полосу прокрутки, то фокус ввода временно теряется.)

Другой компонент, участвующий в изменении фокуса ввода и называемый *противоположным компонентом*, определяется параметром *другое*. Следовательно, если наступает событие `FOCUS_GAINED`, то параметр *другое* будет ссылаться на компонент, теряющий фокус ввода. А если наступает событие `FOCUS_LOST`, то параметр *другое* ссылается на компонент, получающий фокус.

Вызвав метод `getOppositeComponent()`, можно определить другие компоненты. Ниже приведена общая форма этого метода. Этот метод возвращает противоположный компонент.

```
Component getOppositeComponent()
```

Метод `isTemporary()` позволяет выяснить, является ли изменение фокуса временным. Его общая форма следующая:

```
boolean isTemporary()
```

Этот метод возвращает логическое значение `true`, если изменение является временным, а иначе — логическое значение `false`.

Класс `InputEvent`

Абстрактный класс `InputEvent` является производным от класса `ComponentEvent` и служит суперклассом для событий, связанных с вводом данных в компонентах. У него имеются подклассы `KeyEvent` и `MouseEvent`.

В классе `InputEvent` определяется несколько целочисленных констант, представляющих любые модификаторы клавиш, в том числе признак нажатия управляющих клавиш, которые могут быть связаны с событием. Изначально в классе `InputEvent` определены следующие восемь констант для представления модификаторов клавиш:

ALT_MASK	BUTTON2_MASK	META_MASK
ALT_GRAPH_MASK	BUTTON3_MASK	SHIFT_MASK
BUTTON1_MASK	CTRL_MASK	

Но поскольку существует вероятность конфликтов среди модификаторов клавиш, используемых в событиях от клавиатуры и мыши, а также других осложнений, то класс `InputEvent` был дополнен приведенными ниже расширенными значениями модификаторов клавиш. При написании нового кода рекомендуется использовать новые расширенные модификаторы клавиш вместо первоначальных.

ALT_DOWN_MASK	BUTTON2_DOWN_MASK	META_DOWN_MASK
ALT_GRAPH_DOWN_MASK	BUTTON3_DOWN_MASK	SHIFT_DOWN_MASK
BUTTON1_DOWN_MASK	CTRL_DOWN_MASK	

Чтобы проверить, какая именно модифицирующая клавиша была нажата в момент, когда наступило событие, можно воспользоваться методами `isAltDown()`, `isAltGraphDown()`, `isControlDown()`, `isMetaDown()` и `isShiftDown()`. Ниже приведены общие формы этих методов.

```
boolean isAltDown()  
boolean isAltGraphDown()  
boolean isControlDown()  
boolean isMetaDown()  
boolean isShiftDown()
```

Вызвав метод `getModifiers()`, можно получить значение, содержащее все признаки первоначальных модификаторов клавиш. Ниже приведена общая форма этого метода.

```
int getModifiers()
```

Расширенные модификаторы клавиш можно получить, вызвав метод `getModifiersEx()`. Ниже приведена общая форма этого метода.

```
int getModifiersEx()
```

Класс `ItemEvent`

Событие типа `ItemEvent` генерируется, когда производится щелчок на флажке, элементе списка или отмечаемом пункте меню. (Флажки, списки, меню и прочие элементы ГПИ рассматриваются далее в этой книге.) Имеются два типа событий от выбираемых элементов, которые обозначаются целочисленными константами, перечисленными в табл. 24.4.

Таблица 24.4. Константы, определенные в классе `ItemEvent`

Константа	Описание
<code>DESELECTED</code>	Пользователь отменил выбор элемента
<code>SELECTED</code>	Пользователь выбрал элемент

В классе `ItemEvent` определяется еще одна целочисленная константа `ITEM_STATE_CHANGED`, которая сигнализирует об изменении состояния выбираемого элемента.

В классе `ItemEvent` имеется следующий конструктор:

```
ItemEvent(ItemSelectable источник, int тип, Object элемент,
          int состояние)
```

где параметр *источник* обозначает ссылку на компонент, известивший о наступившем событии. Это может быть, например, список или выбираемый элемент. Параметр *тип* обозначает конкретный тип события, параметр *элемент* — тот элемент, который сгенерировал событие, а параметр *состояние* — текущее состояние выбираемого элемента.

Метод `getItem()` может быть использован для получения ссылки на выбираемый элемент, сгенерировавший событие. Его общая форма выглядит следующим образом:

```
Object getItem()
```

Метод `getItemSelectable()` может быть использован для получения ссылки на объект типа `ItemSelectable`, сгенерировавший событие. Его общая форма приведена ниже.

```
ItemSelectable getItemSelectable()
```

Списки и флажки являются примерами выбираемых элементов ГПИ, реализующих интерфейс `ItemSelectable`. Метод `getStateChanged()` возвращает измененное состояние события (т.е. значение `SELECTED` или `DESELECTED`). Его общая форма приведена ниже.

```
int getStateChanged()
```

Класс `KeyEvent`

Событие типа `KeyEvent` генерируется при вводе с клавиатуры. Имеются три типа клавиатурных событий, обозначаемых следующими целочисленными константами: `KEY_PRESSED`, `KEY_RELEASED` и `KEY_TYPED`. События первых двух типов наступают

при нажатии и отпускании клавиши на клавиатуре, а событие третьего типа – при вводе символа. Следует, однако, иметь в виду, что нажатие не всех клавиш приводит к вводу символа с клавиатуры. Так, при нажатии клавиши <Shift> символ не вводится.

В классе KeyEvent определяется целый ряд других целочисленных констант. Например, константы VK_0–VK_9 и VK_A–VK_Z обозначают эквиваленты чисел и букв в коде ASCII. Ниже перечислены другие константы, определяемые в классе KeyEvent.

VK_ALT	VK_DOWN	VK_LEFT	VK_RIGHT
VK_CANCEL	VK_ENTER	VK_PAGE_DOWN	VK_SHIFT
VK_CONTROL	VK_ESCAPE	VK_PAGE_UP	VK_UP

Константы типа VK обозначают *виртуальные коды клавиш*, не зависящие от таких модифицирующих клавиш, как <Control>, <Alt> или <Shift>.

Класс KeyEvent является производным от класса InputEvent. Ниже приведен один из его конструкторов.

```
KeyEvent(Component источник, int тип, long момент, int модификаторы,
          int код, char символ)
```

Здесь параметр *источник* обозначает ссылку на компонент, сгенерировавший событие; параметр *тип* – конкретный тип события; параметр *момент* – тот момент системного времени, когда была нажата клавиша, параметр *модификаторы* – те модифицирующие клавиши, которые были нажаты при наступлении события от клавиатуры. Виртуальный код клавиши (например, VK_UP, VK_A и т.д.) передается в качестве параметра *код*, а символьный эквивалент нажатой клавиши, если таковой существует, – в качестве параметра *символ*. В отсутствие достоверного символа параметр *символ* принимает значение константы CHAR_UNDEFINED. Для событий типа KEY_TYPED параметр *код* будет принимать значение константы VK_UNDEFINED.

В классе KeyEvent определяется несколько методов, но к наиболее употребительным среди них относится метод getKeyChar(), возвращающий введенный с клавиатуры символ, а также метод getKeyCode(), возвращающий код клавиши. Общие формы этих методов приведены ниже.

```
char getKeyChar()
int getKeyCode()
```

Если никаких корректных символов при нажатии клавиши не вводится, то метод getKeyChar() возвращает значение константы CHAR_UNDEFINED. При наступлении события KEY_TYPED метод getKeyCode() возвращает значение константы VK_UNDEFINED.

Класс MouseEvent

Имеется восемь типов событий от мыши. Для их обозначения в классе MouseEvent определяется ряд целочисленных констант, перечисленных в табл. 24.5.

Класс MouseEvent является производным от класса InputEvent. Ниже приведен один из его конструкторов.

```
MouseEvent(Component источник, int тип, long момент, int модификаторы,
            int x, int y, int щелчок, boolean мызов_меню)
```


Таблица 24.5. Константы, определенные в классе `MouseEvent`

Константа	Описание
<code>MOUSE_CLICKED</code>	Пользователь щелкнул кнопкой мыши
<code>MOUSE_DRAGGED</code>	Пользователь перетащил мышь при нажатой кнопке
<code>MOUSE_ENTERED</code>	Курсор мыши наведен на компонент
<code>MOUSE_EXITED</code>	Курсор мыши отведен от компонента
<code>MOUSE_MOVED</code>	Мышь перемещена
<code>MOUSE_PRESSED</code>	Кнопка мыши нажата
<code>MOUSE_RELEASED</code>	Кнопка мыши отпущена
<code>MOUSE_WHEEL</code>	Произведена прокрутка колесика мыши

Здесь параметр *источник* обозначает ссылку на компонент, сгенерировавший событие; параметр *тип* — конкретный тип события; параметр *момент* — тот момент системного времени, когда была нажата клавиша; параметр *модификаторы* — те модифицирующие клавиши, которые были нажаты при наступлении события от мыши. Координаты курсора мыши передаются в качестве параметров *x* и *y*, а подсчет произведенных щелчков — в качестве параметра *щелчки*. Признак *вызов_меню* обозначает, должно ли данное событие вызывать появление всплывающего меню на данной платформе.

К числу наиболее употребительных в этом классе относятся методы `getX()` и `getY()`. Они возвращают координаты *X* и *Y* курсора мыши на момент наступления события. Их общие формы следующие:

```
int getX()
int getY()
```

В качестве альтернативы для получения координат курсора мыши можно вызывать метод `getPoint()`. Ниже приведена его общая форма.

```
Point getPoint()
```

Этот метод возвращает объект типа `Point`, содержащий координаты *X*, *Y* в своих целочисленных членах *x* и *y*.

Метод `translatePoint()` изменяет местоположение события. Ниже приведена его общая форма, где аргументы *x* и *y* добавляются к первоначальным координатам события.

```
void translatePoint(int x, int y)
```

Метод `getClickCount()` получает количество произведенных щелчков мыши для данного события. Ниже приведена его общая форма.

```
int getClickCount()
```

Метод `isPopupTrigger()` проверяет, вызывает ли наступившее событие всплывающее меню на данной платформе. Его общая форма следующая:

```
boolean isPopupTrigger()
```

Имеется также метод `getButton()`, общая форма которого приведена ниже.

```
int getButton()
```

Этот метод возвращает значение, представляющее кнопку, вызвавшую событие. Как правило, этот метод возвращает значение одной из следующих констант, определенных в классе `MouseEvent`:

NOBUTTON	BUTTON1	BUTTON2	BUTTON3
-----------------	----------------	----------------	----------------

Значение константы `NOBUTTON` обозначает, что ни одна из кнопок не была нажата или отпущена. В классе `MouseEvent` имеются также три метода, получающие координаты мыши относительно экрана, а не компонента. Ниже приведены общие формы этих методов.

```
Point getLocationOnScreen()
int getXOnScreen()
int getYOnScreen()
```

Метод `getLocationOnScreen()` возвращает объект типа `Point`, содержащий обе координаты X и Y местоположения курсора мыши. Два других метода возвращают по одной координате соответственно.

Класс `MouseWheelEvent`

Этот класс инкапсулирует событие от колесика мыши. Он является производным класса `MouseEvent`. Не все мыши оснащены колесиками, но если оно есть, то располагается между левой и правой кнопками. Колесики служат для прокрутки содержимого (изображения, текста, таблиц и т.п.). В классе `MouseWheelEvent` определяются целочисленные константы, перечисленные в табл. 24.6.

Таблица 24.6. Константы, определенные в классе `MouseWheelEvent`

Константа	Описание
WHEEL_BLOCK_SCROLL	Произошло событие прокрутки содержимого на страницу вверх или вниз
WHEEL_UNIT_SCROLL	Произошло событие прокрутки содержимого на строку вверх или вниз

Ниже приведен один из конструкторов, определенных в классе `MouseWheelEvent`.

```
MouseWheelEvent(Component источник, int тип, long момент,
                 int модификаторы, int x, int y, int щелчки,
                 boolean вызов_меню, int способ_прокрутки,
                 int количество, int подсчет)
```

Здесь параметр *источник* обозначает ссылку на компонент, сгенерировавший событие; параметр *тип* — конкретный тип события; параметр *момент* — тот момент системного времени, когда была нажата клавиша; параметр *модификаторы* — те модифицирующие клавиши, которые были нажаты при наступлении события прокрутки содержимого. Координаты курсора мыши передаются в качестве параметров *x* и *y*, а подсчет произведенных щелчков — в качестве параметра *щелчки*. Признак *вызов_меню* обозначает, должно ли данное событие вызывать появление всплывающего меню на данной платформе. Параметр *способ_прокрутки* может принимать значение константы `WHEEL_UNIT_SCROLL` или `WHEEL_BLOCK_SCROLL`.

Количество единиц прокрутки передается в качестве параметра *количество*, а количество единиц вращения колесика — в качестве параметра *подсчет*.

В классе `MouseEvent` определяются методы, предоставляющие доступ к событию от колесика мыши. Чтобы получить количество единиц вращения колесика, следует вызвать метод `getWheelRotation()`, общая форма которого приведена ниже.

Этот метод возвращает количество единиц вращения колесика. Если возвращаемое значение положительно, то колесико повернуто против часовой стрелки, а если это значение отрицательно — по часовой стрелке. В версии JDK 7 внедрен метод `getPreciseWheelRotation()`, поддерживающий колесико с высокой разрешающей способностью. Он действует таким же образом, как и метод `getWheelRotation()`, но возвращает значение типа `double`.

Чтобы получить тип прокрутки, следует вызвать метод `getScrollType()`, общая форма которого приведена ниже.

```
int getScrollType()
```

Этот метод возвращает значение константы `WHEEL_UNIT_SCROLL` или `WHEEL_BLOCK_SCROLL`. Если тип прокрутки обозначается константой `WHEEL_UNIT_SCROLL`, то для получения количества единиц прокрутки можно далее вызвать метод `getScrollAmount()`. Общая форма этого метода выглядит следующим образом:

```
int getScrollAmount()
```

Класс `TextEvent`

Экземпляры этого класса описывают текстовые события. Такие события генерируются текстовыми полями и областями, когда в них вводится текст вручную или программно. В классе `TextEvent` определяется целочисленная константа `TEXT_VALUE_CHANGED`.

Ниже приведен единственный конструктор этого класса, где параметр *источник* обозначает ссылку на компонент, сгенерировавший событие, а параметр *тип* — конкретный тип события.

```
TextEvent(Object источник, int тип)
```

Объект класса `TextEvent` не содержит символы, находящиеся в данный момент в текстовом компоненте, сгенерировавшем событие. Вместо этого для извлечения подобной информации в прикладной программе должны использоваться другие методы, связанные с текстовым компонентом. Этим событие данного типа отличается от других рассматриваемых здесь событий. Уведомление о текстовом событии следует считать сигналом приемнику, что он должен извлечь информацию из указанного текстового компонента.

Класс `WindowEvent`

Имеется десять типов оконных событий. Для их обозначения в классе `WindowEvent` определяются целочисленные константы, перечисленные в табл. 24.7.

Таблица 24.7. Константы, определенные в классе `WindowEvent`

Константы	Описание
<code>WINDOW_ACTIVATED</code>	Окно активизировано
<code>WINDOW_CLOSED</code>	Окно закрыто
<code>WINDOW_CLOSING</code>	Пользователь запросил закрытие окна
<code>WINDOW_DEACTIVATED</code>	Окно деактивизировано
<code>WINDOW_DEICONIFIED</code>	Окно развернуто
<code>WINDOW_GAINED_FOCUS</code>	Окно получило фокус ввода
<code>WINDOW_ICONIFIED</code>	Окно свернуто
<code>WINDOW_LOST_FOCUS</code>	Окно утратило фокус ввода
<code>WINDOW_OPENED</code>	Окно открыто
<code>WINDOW_STATE_CHANGED</code>	Состояние окна изменилось

Класс `WindowEvent` является производным от класса `ComponentEvent`. В нем определяется несколько конструкторов. Ниже приведен первый из них.

```
WindowEvent(Window источник, int тип)
```

где параметр *источник* обозначает ссылку на компонент, сгенерировавший событие, а параметр *тип* — конкретный тип события. Следующие три конструктора обеспечивают более точный контроль оконных событий:

```
WindowEvent(Window источник, int тип, Window другое)
WindowEvent(Window источник, int тип, int исходное_состояние,
            int конечное_состояние)
WindowEvent(Window источник, int тип, Window другое,
            int исходное_состояние, int конечное_состояние)
```

Здесь параметр *другое* определяет противоположное окно при наступлении события, связанного с получением фокуса ввода или активизацией окна. Параметр *исходное_состояние* определяет предыдущее состояние окна, а параметр *конечное_состояние* — новое состояние, которое окно получает при смене состояния.

Наиболее употребительным в этом классе является метод `getWindow()`. Он возвращает объект типа `Window`, сгенерировавший событие. Ниже приведена общая форма этого метода.

```
Window getWindow()
```

В классе `WindowEvent` определяются также методы, возвращающие противоположное окно (при наступлении событий, связанных с получением фокуса ввода или активизацией окна), предыдущее состояние окна, а также его текущее состояние. Эти методы имеют следующие общие формы:

```
Window getOppositeWindow()
int getOldState()
int getNewState()
```

Источники событий

В табл. 24.8 перечислены некоторые компоненты пользовательского интерфейса, способные генерировать события, описанные в предыдущем разделе. Помимо этих компонентов ГПИ, любой класс, наследующий от класса `Component`, например класс `Applet`, способен генерировать события в других компонентах. Например, события от клавиатуры и мыши можно получать от апплета, а также создавать свои компоненты, генерирующие события. В этой главе обсуждается обработка только событий от клавиатуры и мыши, а в последующих главах речь пойдет об обработке событий от источников, перечисленных в табл. 24.8.

Таблица 24.8. Примеры источников событий

Источник событий	Описание
Кнопка	Извещает о событиях действия, наступающих после щелчка на кнопке
Флажок	Извещает о событиях от элементов, наступающих после установки и сбросе флажка
Переключатель	Извещает о событиях от элементов, наступающих при изменении выбора
Список	Извещает о событиях действия, наступающих после двойного щелчка на элементе; генерирует событие от элемента после выделения или отмены выделения элемента
Пункт меню	Извещает о событиях действия, наступающих после выбора пункта меню; генерирует событие от элемента после установки и сброса флажка непосредственно в пункте меню
Полоса прокрутки	Извещает о событиях настройки при манипулировании с полосой прокрутки
Текстовые компоненты	Извещает о текстовых событиях, когда пользователь вводит символ
Окно	Извещает об оконных событиях при активизации, закрытии, деактивизации, развертывании, сворачивании, открытии окна или выходе из него

Интерфейсы приемников событий

Как пояснялось ранее, модель делегирования событий состоит из двух частей: источников и приемников. Приемники создаются при реализации одного или нескольких интерфейсов, определенных в пакете `java.awt.event`. Когда наступает событие, его источник вызывает соответствующий метод, определенный в приемнике, и передает объект события в качестве аргумента. В табл. 24.9 перечислены наиболее употребительные интерфейсы приемников событий и представлено краткое описание определяемых в них методов. А в последующих разделах рассматриваются отдельные методы, доступные в каждом из этих интерфейсов.

Таблица 24.9. Наиболее употребительные интерфейсы приемников событий

Интерфейс	Описание
ActionListener	Определяет один метод для приема событий действия
AdjustmentListener	Определяет один метод для приема событий настройки
ComponentListener	Определяет четыре метода для распознавания момента, когда происходит сокрытие, перемещение, изменение размера или отображение компонента
ContainerListener	Определяет два метода для распознавания момента, когда компонент добавляется в контейнер либо исключается из него
FocusListener	Определяет два метода для распознавания момента, когда компонент получает или теряет фокус ввода с клавиатуры
ItemListener	Определяет один метод, распознающий момент, когда изменяется состояние элемента
KeyListener	Определяет три метода, распознающих момент, когда происходит нажатие, отпускание клавиши или ввод символа
MouseListener	Определяет пять методов, распознающих момент, когда производится щелчок кнопкой мыши, курсор мыши наводится на компонент, отводится от компонента, нажимается и отпускается кнопка мыши
MouseMotionListener	Определяет два метода для распознавания момента, когда выполняется перетаскивание курсора или перемещение мыши
MouseWheelListener	Определяет один метод, распознающий момент, когда прокручивается колесико мыши
TextListener	Определяет один метод, распознающий момент, когда изменяется текстовое значение
WindowFocusListener	Определяет два метода для распознавания момента, когда окно получает или теряет фокус ввода
WindowListener	Определяет семь методов, распознающих момент, когда окно активизируется, деактивизируется, разворачивается, сворачивается, открывается или закрывается

Интерфейс ActionListener

В этом интерфейсе определяется метод `actionPerformed()`, вызываемый при наступлении события действия. Ниже приведена его общая форма.

```
void actionPerformed(ActionEvent событие_действия)
```

Интерфейс AdjustmentListener

В этом интерфейсе определяется метод `adjustmentValueChanged()`, вызываемый при наступлении события настройки. Ниже приведена его общая форма.

```
void adjustmentValueChanged(AdjustmentEvent событие_настройки)
```

Интерфейс `ComponentListener`

В этом интерфейсе определяются четыре метода, вызываемых при изменении размеров компонента, его перемещении, отображении или сокрытии. Ниже приведены их общие формы.

```
void componentResized(ComponentEvent событие_от_компонента)
void componentMoved(ComponentEvent событие_от_компонента)
void componentShown(ComponentEvent событие_от_компонента)
void componentHidden(ComponentEvent событие_от_компонента)
```

Интерфейс `ContainerListener`

В этом интерфейсе определяются два метода. Когда компонент вводится в контейнер, вызывается метод `componentAdded()`. А когда компонент удаляется из контейнера, вызывается метод `componentRemoved()`. Общие формы этих методов выглядят следующим образом:

```
void componentAdded(ContainerEvent событие_от_компонента)
void componentRemoved(ContainerEvent событие_от_компонента)
```

Интерфейс `FocusListener`

В этом интерфейсе определяются два метода. Когда компонент получает фокус ввода с клавиатуры, вызывается метод `focusGained()`. А когда компонент теряет фокус ввода с клавиатуры, вызывается метод `focusLost()`. Ниже приведены общие формы этих методов.

```
void focusGained(FocusEvent событие_фокуса_ввода)
void focusLost(FocusEvent событие_фокуса_ввода)
```

Интерфейс `ItemListener`

В этом интерфейсе определяется метод `itemStateChanged()`, вызываемый при изменении состояния элемента. Его общая форма следующая:

```
void itemStateChanged(ItemEvent событие_от_элемента)
```

Интерфейс `KeyListener`

В этом интерфейсе определяются три метода. Методы `keyPressed()` и `keyReleased()` вызываются при нажатии и отпускании клавиш соответственно. А метод `keyTyped()` вызывается при вводе символа.

Так, если пользователь нажимает и отпускает клавишу <A>, последовательно наступают следующие три события: нажатие клавиши, ввод символа, отпускание клавиши. Если же пользователь нажимает и отпускает клавишу <Home>, последовательно наступают только два события: нажатие клавиши и ее отпускание. Общие формы этих методов выглядят следующим образом:

```
void keyPressed(KeyEvent событие_от_клавиатуры)
void keyReleased(KeyEvent событие_от_клавиатуры)
void keyTyped(KeyEvent событие_от_клавиатуры)
```

Интерфейс `MouseListener`

В этом интерфейсе определяются пять методов. Если кнопка мыши нажата и отпущена в одной и той же точке, то вызывается метод `mouseClicked()`. Когда курсор мыши наводится на компонент, вызывается метод `mouseEntered()`. А когда курсор отводится от компонента, вызывается метод `mouseExited()`. И наконец, методы `mousePressed()` и `mouseReleased()` вызываются, когда кнопка мыши нажимается и отпускается соответственно. Ниже приведены общие формы этих методов.

```
void mouseClicked(MouseEvent событие_от_мышь)  
void mouseEntered(MouseEvent событие_от_мышь)  
void mouseExited(MouseEvent событие_от_мышь)  
void mousePressed(MouseEvent событие_от_мышь)  
void mouseReleased(MouseEvent событие_от_мышь)
```

Интерфейс `MouseMotionListener`

В этом интерфейсе определяются два метода. В частности, метод `mouseDragged()` вызывается неоднократно при перетаскивании объекта мышью, а метод `mouseMoved()` — при перемещении курсора мыши. Их общие формы следующие:

```
void mouseDragged(MouseEvent событие_от_мышь)  
void mouseMoved(MouseEvent событие_от_мышь)
```

Интерфейс `MouseWheelListener`

В этом интерфейсе определяется метод `mouseWheelMoved()`, вызываемый при прокрутке колесика мыши. Его общая форма показана ниже.

```
void mouseWheelMoved(MouseWheelEvent событие_перемещения_колесика)
```

Интерфейс `TextListener`

В этом интерфейсе определяется метод `textChanged()`, вызываемый при изменении содержимого текстовой области или текстового поля. Его общая форма выглядит следующим образом:

```
void textChanged(TextEvent текстовое_событие)
```

Интерфейс `WindowFocusListener`

В этом интерфейсе определяются два метода: `windowGainedFocus()` и `windowLostFocus()`. Они вызываются в тот момент, когда окно получает и теряет фокус ввода. Ниже приведены общие формы этих методов.

```
void windowGainedFocus(WindowEvent оконное_событие)  
void windowLostFocus(WindowEvent оконное_событие)
```

Интерфейс `WindowListener`

В этом интерфейсе определяются семь методов. Методы `windowActivated()` и `windowDeactivated()` вызываются, когда окно активизируется и деактивизируется соответственно.

Если окно сворачивается в пиктограмму, то вызывается метод `windowIconified()`. Когда же окно разворачивается, вызывается метод `windowDeIconified()`. А когда окно открывается и закрывается, вызываются методы `windowOpened()` и `windowClosed()` соответственно. Метод `windowClosing()` вызывается при закрытии окна. Общие формы этих методов выглядят следующим образом:

```
void windowActivated(WindowEvent оконное_событие)
void windowClosed(WindowEvent оконное_событие)
void windowClosing(WindowEvent оконное_событие)
void windowDeactivated(WindowEvent оконное_событие)
void windowDeiconified(WindowEvent оконное_событие)
void windowIconified(WindowEvent оконное_событие)
void windowOpened(WindowEvent оконное_событие)
```

Применение модели делегирования событий

Итак, рассмотрев принципы, по которым действует модель делегирования событий, а также представив различные ее компоненты, можно перейти от теории к практике. Пользоваться моделью делегирования событий совсем не трудно. Для этого достаточно выполнить два действия.

- Реализовать соответствующий интерфейс в приемнике событий, чтобы он мог принимать события требуемого типа.
- Реализовать код регистрации и снятия с регистрации, если требуется, приемника как получателя уведомлений о событиях.

Следует, однако, иметь в виду, что источник может извещать о нескольких типах событий. Каждое событие должно быть зарегистрировано отдельно. К тому же объект может подписаться на получение нескольких типов событий и должен реализовать все интерфейсы, необходимые для получения этих событий.

Чтобы показать, каким образом модель делегирования событий функционирует на практике, рассмотрим примеры обработки событий от двух наиболее употребительных устройств ввода: мыши и клавиатуры.

Обработка событий от мыши

Чтобы обработать события от мыши, следует реализовать интерфейсы `MouseListener` и `MouseMotionListener`. (Можно было бы также реализовать интерфейс `MouseWheelListener`, но мы не станем здесь этого делать.) Весь процесс обработки событий от мыши демонстрируется в приведенном ниже примере аплета. В строке состояния окна этого аплета выводятся текущие координаты мыши. Всякий раз, когда нажимается кнопка мыши, на месте курсора мыши появляется слово "Down" (Нажато). И всякий раз, когда кнопка мыши отпускается, — слово "Up" (Отпущено). А если производится щелчок кнопкой мыши, то в левом верхнем углу области отображения аплета выводится сообщение "Mouse clicked" (Произведен щелчок кнопкой мыши).

Когда же курсор мыши наводится на окно аплета или отводится от него, то в левом верхнем углу области отображения аплета также выводится соответствующее

сообщение. При перетаскивании курсора мыши выводится символ *, сопровождающий курсор мыши. Обратите внимание на то, что в двух переменных, `mouseX` и `mouseY`, сохраняются координаты местоположения курсора мыши, когда происходят события нажатия и отпускания кнопки мыши, а также события перетаскивания. Эти координаты затем используются методом `paint()` для вывода соответствующего сообщения в той точке, где возникает событие.

```
// Прдемонстрировать обработчики событий от мыши
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="MouseEvents" width=300 height=100>
  </applet>
*/

public class MouseEvents extends Applet
implements MouseListener, MouseMotionListener {

    String msg = "";
    int mouseX = 0, mouseY = 0; // координаты курсора мыши

    public void init() {
        addMouseListener(this);
        addMouseMotionListener(this);
    }

    // обработать событие от щелчка кнопкой мыши
    public void mouseClicked(MouseEvent me) {
        // сохранить координаты
        mouseX = 0;
        mouseY = 10;
        msg = "Mouse clicked."; // Произведен щелчок кнопкой мыши
        repaint();
    }

    // обработать событие наведения курсора мыши
    public void mouseEntered(MouseEvent me) {
        // сохранить координаты
        mouseX = 0;
        mouseY = 10;
        msg = "Mouse entered."; // Курсор наведен
        repaint();
    }

    // обработать событие отведения курсора мыши
    public void mouseExited(MouseEvent me) {
        // сохранить координаты
        mouseX = 0;
        mouseY = 10;
        msg = "Mouse exited."; // Курсор отведен
        repaint();
    }

    // обработать событие нажатия кнопки мыши
    public void mousePressed(MouseEvent me) {
        // сохранить координаты
        mouseX = me.getX();
        mouseY = me.getY();
        msg = "Down"; // Кнопка мыши нажата
        repaint();
    }
}
```

```

// обработать событие отпускания кнопки мыши
public void mouseReleased(MouseEvent me) {
    // сохранить координаты
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Up"; // Кнопка мыши отпущена
    repaint();
}

// обработать событие перетаскивания курсора мыши
public void mouseDragged(MouseEvent me) {
    // сохранить координаты
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "**";
    showStatus("Dragging mouse at " + mouseX + ", " + mouseY);
    // Перетаскивание курсора мыши в точку с указанными координатами
    repaint();
}

// обработать событие перемещения мыши
public void mouseMoved(MouseEvent me) {
    // показать состояние
    showStatus("Moving mouse at " + me.getX() + ", " + me.getY());
    // Перемещение курсора мыши в точку с указанными координатами
}

// вывести сообщение из переменной msg на текущей позиции
// с координатами X,Y в окне апплета
public void paint(Graphics g) {
    g.drawString(msg, mouseX, mouseY);
}
}

```

Пример выполнения данного апплета приведен на рис. 24.1.

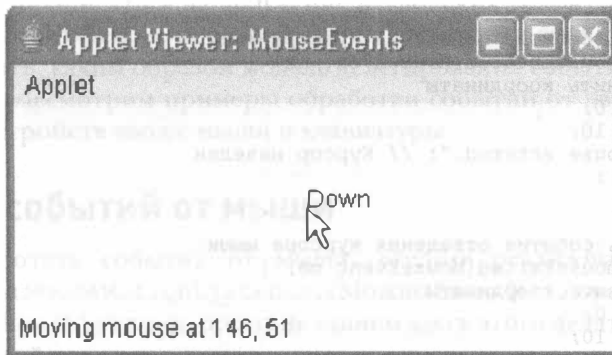


Рис. 24.1. Пример выполнения апплета, в котором обрабатываются события от мыши

Рассмотрим данный пример апплета более подробно. В частности, класс `MouseEvents` расширяет класс `Applet` и реализует интерфейсы `MouseListener` и `MouseMotionListener`. Оба эти интерфейса содержат методы, принимающие и обрабатывающие различные типы событий от мыши. Обратите внимание на то, что апплет одновременно является источником и приемником этих событий. И это

вполне допустимо, поскольку класс `Component`, предоставляющий методы `addMouseListener()` и `addMouseMotionListener()`, служит суперклассом для класса `Applet`. Вообще, использование одного и того же объекта в качестве источника и приемника событий характерно для апплетов.

В методе `init()` апплет регистрирует себя в качестве приемника событий от мыши. Это делается с помощью методов `addMouseListener()` и `addMouseMotionListener()`, которые, как упоминалось ранее, являются членами класса `Component`. Ниже приведены общие формы этих методов.

```
void addMouseListener(MouseListener ml)
void addMouseMotionListener(MouseMotionListener mml)
```

Здесь параметр `ml` обозначает ссылку на объект, принимающий события от мыши, а параметр `mml` — ссылку на объект, принимающий события перемещения мыши. В рассматриваемом здесь примере апплета оба эти метода принимают в качестве параметра один и тот же объект.

Далее в данном примере апплета реализуются все методы, определенные в интерфейсах `MouseListener` и `MouseMotionListener`, т.е. обработчики разных событий от мыши. Каждый из них обрабатывает свое событие, а затем возвращает управление.

Обработка событий от клавиатуры

Для обработки событий от клавиатуры используется та же общая архитектура, что и для событий от мыши, как показано в примере апплета в предыдущем разделе. Отличие, разумеется, состоит в том, что в данном случае требуется реализовать интерфейс `KeyListener`.

Прежде чем обратиться к конкретному примеру, имеет смысл еще раз рассмотреть процесс генерирования событий от клавиатуры. При нажатии клавиши на клавиатуре наступает событие типа `KEY_PRESSED`. Это приводит к вызову обработчика событий `keyPressed()`. А при отпускании клавиши наступает событие типа `KEY_RELEASED` и вызывается обработчик событий `keyReleased()`. Если в результате нажатия клавиши клавиатура формирует символ, то наступает также событие типа `KEY_TYPED` и вызывается обработчик событий `keyTyped()`. Таким образом, всякий раз, когда пользователь нажимает клавишу, наступают как минимум два, а то и три события. Если главный интерес представляют только вводимые с клавиатуры символы, то сведения о нажатии и отпускании клавиш можно проигнорировать. Но если прикладная программа должна обрабатывать нажатие специальных клавиш (например, клавиш со стрелками или функциональных клавиш), то их нажатие следует отслеживать в обработчике событий `keyPressed()`.

В приведенном ниже примере апплета демонстрируется процесс ввода с клавиатуры. Введенные с клавиатуры символы выводятся в окне апплета, а в строке состояния показывается, нажата ли клавиша или отпущена.

```
// Продемонстрировать обработчики событий от клавиатуры
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="SimpleKey" width=300 height=100>
</applet>
```

```

*/

public class SimpleKey extends Applet
implements KeyListener {

    String msg = "";
    int X = 10, Y = 20; // координаты вывода

    public void init() {
        addKeyListener(this);
    }

    public void keyPressed(KeyEvent ke) {
        showStatus("Key Down"); // Клавиша нажата
    }

    public void keyReleased(KeyEvent ke) {
        showStatus("Key Up"); // Клавиша отпущена
    }

    public void keyTyped(KeyEvent ke) {
        msg += ke.getKeyChar();
        repaint();
    }
    // вывести символы, введенные с клавиатуры
    public void paint(Graphics g) {
        g.drawString(msg, X, Y);
    }
}

```

Пример выполнения данного апплета приведен на рис. 24.2.

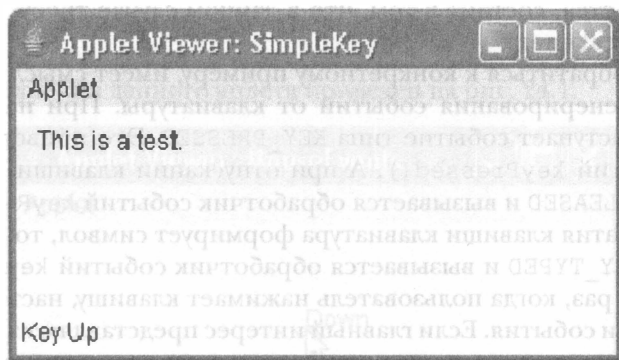


Рис. 24.2. Пример выполнения апплета, в котором обрабатываются события от клавиатуры

Если же требуется обрабатывать нажатие специальных клавиш (например, клавиш со стрелками и функциональных клавиш), то на их нажатие следует реагировать в обработчике событий `keyPressed()`, поскольку обработчик событий `keyTyped()` не реагирует на нажатие таких клавиш. Чтобы вывести нажатие подобных клавиш, следует воспользоваться виртуальными кодами клавиш. В следующем примере апплета выводятся наименования некоторых специальных клавиш:

```

// Продемонстрировать некоторые виртуальные коды клавиш
import java.awt.*;
import java.awt.event.*;

```

```
import java.applet.*;
/*
  <applet code="KeyEvents" width=300 height=100>
</applet>
*/

public class KeyEvents extends Applet
implements KeyListener {

    String msg = "";
    int X = 10, Y = 20; // координаты вывода

    public void init() {
        addKeyListener(this);
    }

    public void keyPressed(KeyEvent ke) {
        showStatus("Key Down"); // Клавиша нажата
        int key = ke.getKeyCode();

        switch(key) {
            case KeyEvent.VK_F1:
                msg += "<F1>";
                break;
            case KeyEvent.VK_F2:
                msg += "<F2>";
                break;
            case KeyEvent.VK_F3:
                msg += "<F3>";
                break;
            case KeyEvent.VK_PAGE_DOWN:
                msg += "<PgDn>";
                break;
            case KeyEvent.VK_PAGE_UP:
                msg += "<PgUp>";
                break;
            case KeyEvent.VK_LEFT:
                msg += "<Left Arrow>"; // Клавиша <←>
                break;
            case KeyEvent.VK_RIGHT:
                msg += "<Right Arrow>"; // Клавиша <→>
                break;
        }
        repaint();
    }

    public void keyReleased(KeyEvent ke) {
        showStatus("Key Up"); // Клавиша отпущена
    }

    public void keyTyped(KeyEvent ke) {
        msg += ke.getKeyChar();
        repaint();
    }

    // вывести символы, введенные с клавиатуры
    public void paint(Graphics g) {
        g.drawString(msg, X, Y);
    }
}
```

Пример выполнения данного апплета приведен на рис. 24.3.

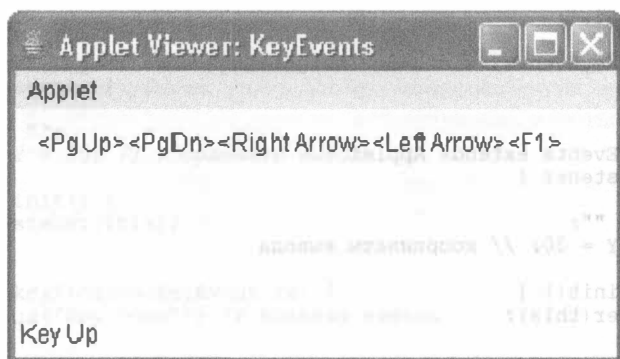


Рис. 24.3. Пример выполнения апплета, в котором используются виртуальные коды некоторых клавиш

Процедуры, продемонстрированные в приведенных выше примерах обработки событий от мыши и клавиатуры, могут быть обобщены для обработки событий любого типа, включая события от элементов управления. В последующих главах будут представлены многочисленные примеры обработки событий других типов, но все они следуют той же основной схеме, что и в рассмотренных выше примерах апплетов.

Классы адаптеров

В Java имеется специальное средство, которое называется *классом адаптера* и в некоторых случаях позволяет упростить реализацию обработчиков событий. В классе адаптера предоставляется пустая реализация всех методов интерфейса приемника событий. Классы адаптеров удобны в тех случаях, когда требуется принимать только те события, которые обрабатываются конкретным интерфейсом приемника событий. В качестве приемника событий можно определить новый класс, расширив один из классов адаптеров и реализовав только те события, обработка которых представляет особый интерес.

Например, в классе `MouseMotionAdapter` имеются два метода, `mouseDragged()` и `mouseMoved()`, которые определены в интерфейсе `MouseMotionListener`. Если интерес представляют только события перетаскивания курсора мыши, то для их обработки достаточно расширить класс `MouseMotionAdapter` и переопределить метод `mouseDragged()`. Пустая реализация метода `mouseMoved()` будет автоматически обрабатывать события перемещения курсора мыши.

В табл. 24.10 перечислены наиболее употребительные классы адаптеров из пакета `java.awt.event`, а также интерфейсы приемников событий, реализуемые каждым из них.

В приведенном ниже примере демонстрируется применение класса адаптера для вывода сообщения в строке состояния, находящейся в окне средства просмотра апплетов или браузера, когда производится щелчок кнопкой мыши или перетаскивание ее курсора, а все прочие события от мыши игнорируются. Апплет из данного примера состоит из трех классов. В частности, класс `AdapterDemo` расширяет класс `Applet`. В его методе `init()` создается экземпляр класса `MyMouseAdapter`,

который регистрируется для получения уведомлений о событиях от мыши, а также экземпляр класса `MyMouseMotionAdapter`, который регистрируется для получения уведомлений о событиях перемещения мыши. Оба конструктора принимают ссылку на апплет в качестве аргумента.

Таблица 24.10. Наиболее употребительные интерфейсы приемников событий, реализуемые классами адаптеров

Класс адаптера	Интерфейс приемника событий
<code>ComponentAdapter</code>	<code>ComponentListener</code>
<code>ContainerAdapter</code>	<code>ContainerListener</code>
<code>FocusAdapter</code>	<code>FocusListener</code>
<code>KeyAdapter</code>	<code>KeyListener</code>
<code>MouseAdapter</code>	<code>MouseListener</code>
<code>MouseMotionAdapter</code>	<code>MouseMotionListener</code>
<code>WindowAdapter</code>	<code>WindowListener</code>

Класс `MyMouseAdapter` расширяет класс `MouseAdapter` и переопределяет метод `mouseClicked()`. Все прочие события от мыши игнорируются кодом, наследуемым из класса `MouseAdapter`. Класс `MyMouseMotionAdapter` расширяет класс `MouseMotionAdapter` и переопределяет метод `mouseDragged()`. Другое событие перемещения мыши негласно игнорируется кодом, наследуемым из класса `MouseAdapter`. (В классе `MouseAdapter` предоставляется также пустая реализация интерфейса `MouseMotionListener`. Но ради наглядности примера каждый тип событий от мыши обрабатывается по отдельности.)

Обратите внимание на то, что оба класса приемников событий сохраняют ссылку на апплет. Эти сведения предоставляются в виде аргумента и используются в дальнейшем для вызова метода `showStatus()`.

```
// Продемонстрировать применение класса адаптера
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="AdapterDemo" width=300 height=100>
  </applet>
*/

public class AdapterDemo extends Applet {
    public void init() {
        addMouseListener(new MyMouseAdapter(this));
        addMouseMotionListener(new MyMouseMotionAdapter(this));
    }
}

class MyMouseAdapter extends MouseAdapter {

    AdapterDemo adapterDemo;
    public MyMouseAdapter(AdapterDemo adapterDemo) {
        this.adapterDemo = adapterDemo;
    }
    // обработать событие от щелчка кнопкой мыши
    public void mouseClicked(MouseEvent me) {
        adapterDemo.showStatus("Mouse clicked");
    }
}
```



```

        // Произведен щелчок кнопкой мыши
    }
}

class MyMouseMotionAdapter extends MouseMotionAdapter {
    AdapterDemo adapterDemo;
    public MyMouseMotionAdapter(AdapterDemo adapterDemo) {
        this.adapterDemo = adapterDemo;
    }

    // обработать событие перетаскивания курсора мыши
    public void mouseDragged(MouseEvent me) {
        adapterDemo.showStatus("Mouse dragged");
        // Перетаскивание курсора мыши
    }
}

```

Как видите, благодаря тому, что отпадает необходимость реализовывать все методы, определенные в интерфейсах `MouseMotionListener` и `MouseListener`, экономится немало труда, а код избавляется от нагромождения пустых методов. В качестве упражнения можете попробовать переписать один из приведенных ранее примеров, где обрабатываются события от ввода с клавиатуры, воспользовавшись классом `KeyAdapter`.

Внутренние классы

Основные положения о внутренних классах были представлены в главе 7. А здесь поясняется, почему они так важны. Напомним, что *внутренним* называется класс, определенный в другом классе или даже в выражении. В этом разделе показано, каким образом внутренние классы упрощают код, в котором применяются классы адаптеров событий.

Чтобы стала понятнее выгода, которую приносят внутренние классы, рассмотрим приведенный ниже пример апплета. В этом примере внутренние классы *не* применяются. Главное назначение апплета из этого примера — вывести сообщение "Mouse Pressed" (Кнопка мыши нажата) в строке состояния, находящейся в окне средства просмотра апплетов или браузера, когда нажата кнопка мыши. В данном апплете имеются еще два класса верхнего уровня. Класс `MousePressedDemo` расширяет класс `Applet`, а класс `MyMouseAdapter` — класс `MouseAdapter`. Метод `init()` из класса `MousePressedDemo` создает экземпляр класса `MyMouseAdapter` и предоставляет этот объект в качестве аргумента методу `addMouseListener()`.

Обратите внимание на то, что ссылка на апплет передается в качестве аргумента конструктору класса `MyMouseAdapter`. Эта ссылка сохраняется в переменной экземпляра для последующего использования в методе `mousePressed()`. Когда нажимается кнопка мыши, вызывается метод `showStatus()` по сохраненной ссылке на апплет. Иными словами, метод `showStatus()` вызывается относительно ссылки на апплет, сохраненной в переменной экземпляра класса `MyMouseAdapter`.

```

// В этом апплете внутренний класс НЕ применяется
import java.applet.*;
import java.awt.event.*;
/*
    <applet code="MousePressedDemo" width=200 height=100>

```

```

</applet>
*/

public class MousePressedDemo extends Applet {
    public void init() {
        addMouseListener(new MyMouseAdapter(this));
    }
}

class MyMouseAdapter extends MouseAdapter {
    MousePressedDemo mousePressedDemo;
    public MyMouseAdapter(MousePressedDemo mousePressedDemo) {
        this.mousePressedDemo = mousePressedDemo;
    }
    public void mousePressed(MouseEvent me) {
        mousePressedDemo.showStatus("Mouse Pressed");
        // Кнопка мыши нажата
    }
}

```

В приведенном ниже примере показано, как усовершенствовать апплет из предыдущего примера, используя внутренний класс. В данном примере `InnerClassDemo` — класс верхнего уровня, расширяющий класс `Applet`, а `MyMouseAdapter` — внутренний класс, расширяющий класс `MouseAdapter`. А поскольку класс `MyMouseAdapter` определен в области действия класса `InnerClassDemo`, то он имеет доступ ко всем переменным и методам, находящимся в контексте этого класса. Таким образом, метод `showStatus()` можно вызвать непосредственно из метода `mousePressed()`. И теперь это больше не нужно делать через сохраняемую ссылку на апплет, а следовательно, и передавать конструктору `MyMouseAdapter()` ссылку на вызывающий объект.

```

// Продемонстрировать применение внутреннего класса
import java.applet.*;
import java.awt.event.*;
/*
<applet code="InnerClassDemo" width=200 height=100>
</applet>
*/

public class InnerClassDemo extends Applet {
    public void init() {
        addMouseListener(new MyMouseAdapter());
    }
    class MyMouseAdapter extends MouseAdapter {
        public void mousePressed(MouseEvent me) {
            showStatus("Mouse Pressed"); // Кнопка мыши нажата
        }
    }
}

```

Анонимные внутренние классы

Анонимным называется такой внутренний класс, которому не присвоено имя. В этом разделе показывается, как анонимный внутренний класс позволяет упростить написание обработчиков событий. Рассмотрим в качестве примера приведенный ниже апплет. Как и прежде, назначение этого апплета — вывести сообщение "Mouse Pressed" в строке состояния, находящейся в окне средства просмотра апплетов или браузера, когда нажата кнопка мыши.

```
// Продемонстрировать применение анонимного внутреннего класса
import java.applet.*;
import java.awt.event.*;
/*
  <applet code="AnonymousInnerClassDemo" width=200 height=100>
  </applet>
*/

public class AnonymousInnerClassDemo extends Applet {
    public void init() {
        addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent me) {
                showStatus("Mouse Pressed"); // Кнопка мыши нажата
            }
        });
    }
}
```

В данном апплете присутствует только один класс верхнего уровня `AnonymousInnerClassDemo`. Метод `addMouseListener()` вызывается из метода `init()`. Его аргументом служит выражение, определяющее и создающее экземпляр анонимного внутреннего класса. Проанализируем это выражение подробнее.

Синтаксис `new MouseAdapter() { ... }` указывает компилятору, что в коде, заключенном в фигурные скобки, определяется анонимный внутренний класс. Более того, этот класс расширяет класс `MouseAdapter`. И хотя этот новый класс не имеет имени, его экземпляр автоматически создается при выполнении данного выражения.

Анонимный внутренний класс определяется в контексте класса `AnonymousInnerClassDemo` и поэтому имеет доступ ко всем переменным и методам, находящимся в контексте данного класса. Следовательно, в нем можно вызвать метод `showStatus()` непосредственно.

Как видите, именованные и анонимные внутренние классы разрешают некоторые неприятные затруднения простым и удобным способом. Они также позволяют создавать более эффективный код.

Введение в библиотеку AWT: работа с окнами, графикой и текстом

Библиотека Abstract Window Toolkit (AWT) стала первым каркасом ГПИ, начиная с версии Java 1.0. Эта библиотека включает многочисленные классы и методы, позволяющие создавать окна с простыми элементами управления. Библиотека AWT уже была представлена в главе 23, где она использовалась в нескольких кратких примерах апплетов. А в этой главе она рассматривается более подробно. В частности, в ней будет показано, как создавать окна и управлять ими, обращаться со шрифтами, выводимым текстом и применять графику. Различные элементы управления из библиотеки AWT, в том числе полосы прокрутки и экранные кнопки, описываются в главе 26, где также поясняются дополнительные особенности механизма обработки событий в Java. А в главе 27 представлено введение в подсистему AWT для формирования и обработки изображений.

Следует особо подчеркнуть, что новые прикладные программы с ГПИ редко приходится разрабатывать только средствами библиотеки AWT, поскольку для этой цели в Java внедрены более эффективные библиотеки Swing и JavaFX. Несмотря на это, библиотека по-прежнему остается важным компонентом Java. Ниже поясняются веские на то основания.

На момент написания данной книги наиболее употребительной для построения ГПИ считалась библиотека Swing, поскольку она предоставляет для этой цели более развитые и удобные средства, чем библиотека AWT. Из этого можно сделать поспешный вывод, что библиотека AWT больше не отвечает требованиям разработки современных прикладных программ с ГПИ, поскольку явно уступает в этом отношении библиотеке Swing. Но это ложный вывод. Напротив, ясное представление о функциональных возможностях библиотеки AWT не теряет своей актуальности до сих пор, потому что она положена в основу библиотеки Swing, причем многие классы AWT применяются (прямо или косвенно) в Swing. Таким образом, для эффективного применения библиотеки Swing по-прежнему требуются прочные знания и навыки обращения с библиотекой AWT.

Для построения ГПИ в Java появилась новейшая библиотека JavaFX. Предполагается, что в какой-то момент в будущем она полностью заменит библиотеку Swing, когда станет наиболее употребительной для построения ГПИ прикладных программ на Java. Но даже если это и произойдет, то все равно останется еще немало унаследованного кода, опирающегося на библиотеку Swing, а следовательно, и на библиотеку AWT. И этот код еще какое-то время придется сопровождать. И наконец, библиотека AWT оказывается по-прежнему удобной для разработки

мелких прикладных программ (особенно апплетов), где предъявляются минимальные требования к ГПИ. Таким образом, основательные знания и навыки работы с библиотекой AWT по-прежнему требуются, несмотря на то, что она является самой старой библиотекой Java для построения ГПИ.

Чаще всего библиотека AWT применяется для разработки апплетов. Но с ее помощью можно также создавать автономные оконные программы, выполняющиеся в такой среде с ГПИ, как, например, Windows. Ради удобства изложения материала этой главы большинство примеров представлены в ней в виде апплетов, которые нетрудно выполнить в средстве просмотра апплетов. И лишь в некоторых примерах демонстрируется создание автономных оконных программ, которые можно запускать на выполнение непосредственно.

И последнее предварительное замечание: библиотека AWT довольно обширна, и для полного ее описания потребуется отдельная книга. Поэтому в данной книге просто невозможно описать во всех подробностях каждый класс, метод или переменную экземпляра из библиотеки AWT. Тем не менее в этой и последующих главах поясняются основные приемы, которые следует освоить, чтобы пользоваться библиотекой AWT. Опираясь на эти приемы как на прочное основание, можно самостоятельно изучить остальные части библиотеки AWT, чтобы затем перейти к библиотеке Swing.

На заметку! Если вы еще не читали главу 24, сделайте это теперь. В ней представлен краткий обзор механизма обработки событий, который будет применяться во многих примерах из этой главы.

Классы библиотеки AWT

В пакете `java.awt`, одном из наиболее крупных в Java, содержатся классы библиотеки AWT. Правда, благодаря логической организации в виде нисходящей иерархии классов в этом пакете понять его и пользоваться им намного легче, чем может показаться на первый взгляд. В табл. 25.1 приведены лишь некоторые из многих классов AWT.

Таблица 25.1. Избранные классы из библиотеки AWT

Класс	Описание
AWTEvent	Инкапсулирует события из библиотеки AWT
AWTEventMulticaster	Доставляет события многим приемникам
BorderLayout	Диспетчер граничной компоновки с помощью следующих пяти компонентов: North , South , East , West и Center
Button	Создает элемент управления в виде экранной кнопки
Canvas	Пустое, свободное от семантики окно
CardLayout	Диспетчер карточной компоновки, эмулирующий индексированные карты. Отображается только одна, самая верхняя карта
Checkbox	Создает элемент управления в виде флажка
CheckboxGroup	Создает группу флажков
CheckboxMenuItem	Создает переключаемый пункт меню
Choice	Создает раскрывающийся список

Продолжение табл. 25.1

Класс	Описание
Color	Управляет цветом переносимым и не зависящим от платформы способом
Component	Абстрактный суперкласс для различных компонентов AWT
Container	Подкласс, который является производным от класса Component и может содержать другие компоненты
Cursor	Инкапсулирует растровый курсор
Dialog	Создает диалоговое окно
Dimension	Определяет размеры объекта. Ширина сохраняется в поле width , высота — в поле height
Event	Инкапсулирует события
EventQueue	Организует очередь событий
FileDialog	Создает окно, в котором можно выбрать файл
FlowLayout	Диспетчер поточной компоновки, размещающий компоненты слева направо и сверху вниз
Font	Инкапсулирует печатный шрифт
FontMetrics	Инкапсулирует типографские параметры шрифтового оформления текста. Эти параметры помогают отображать текст в окне
Frame	Создает стандартное окно, снабженное строкой заголовка, элементами управления размерами и строкой меню
Graphics	Инкапсулирует графический контекст. Этот контекст используется различными методами вывода данных в окне
GraphicsDevice	Описывает графическое устройство вывода, например экран монитора или принтер
GraphicsEnvironment	Описывает коллекцию доступных объектов типа Font и GraphicsDevice
GridBagConstraints	Описывает различные ограничения, накладываемые на класс GridBagLayout
GridBagLayout	Диспетчер сеточно-контейнерной компоновки, отображающий компоненты в соответствии с ограничениями, накладываемыми средствами класса GridBagConstraints
GridLayout	Диспетчер сеточной компоновки, отображающий компоненты в виде двухмерной сетки
Image	Инкапсулирует графические изображения
Insets	Инкапсулирует границы контейнера
Label	Создает метку, отображающую символьную строку
List	Создает список, из которого пользователь может выбирать отдельные элементы. Аналогичен стандартному списковому окну в Windows
MediaTracker	Управляет мультимедийными объектами
Menu	Создает ниспадающее меню
MenuBar	Создает строку меню
MenuComponent	Абстрактный класс, реализуемый различными классами меню

Класс	Описание
MenuItem	Создает пункт меню
MenuShortcut	Инкапсулирует “горячую клавишу” для быстрого вызова пункта меню
Panel	Простейший конкретный подкласс, производный от класса Container
Point	Инкапсулирует пару декартовых координат, хранящихся в его членах x и y
Polygon	Инкапсулирует многоугольник
PopupMenu	Создает всплывающее меню
PrintJob	Абстрактный класс, представляющий задание на печать
Rectangle	Инкапсулирует прямоугольник
Robot	Поддерживает автоматическую проверку прикладных программ на основе библиотеки AWT
Scrollbar	Создает элемент управления в виде полосы прокрутки
ScrollPane	Контейнер, предоставляющий горизонтальную и вертикальную полосы прокрутки для другого компонента
SystemColor	Содержит цвета для таких виджетов (т.е. элементов) ГПИ, как окна, полосы прокрутки, текстовые поля и пр.
TextArea	Создает элемент управления в виде многострочного текстового редактора
TextComponent	Суперкласс для классов TextArea и TextField
TextField	Создает элемент управления в виде однострочного текстового редактора
Toolkit	Абстрактный класс, реализуемый в библиотеке AWT
Window	Создает окно без рамки, строк меню и заголовка

Базовая структура библиотеки AWT не менялась еще с версии Java 1.0, и поэтому некоторые первоначальные методы из этой версии уже устарели и заменены новыми, хотя они все еще поддерживаются в Java ради обратной совместимости. Но поскольку эти методы не предназначены для применения в новом коде, то они в данной книге не рассматриваются.

Основные положения об окнах

Окна определяются в библиотеке AWT в соответствии с иерархией классов, которые обеспечивают функциональные и конкретные возможности на каждом уровне. Два наиболее употребительных класса для формирования окон происходят от класса **Panel**, применяемого в апплетах, или же от класса **Frame**, создающего стандартное окно прикладной программы. Большую часть функциональных возможностей эти окна наследуют от своих родительских классов. Поэтому описание иерархии классов, связанных в этими двумя классами, является основополагающим для их понимания. На рис. 25.1 показана иерархия классов **Panel** и **Frame**. Рассмотрим каждый класс из этой иерархии в отдельности.

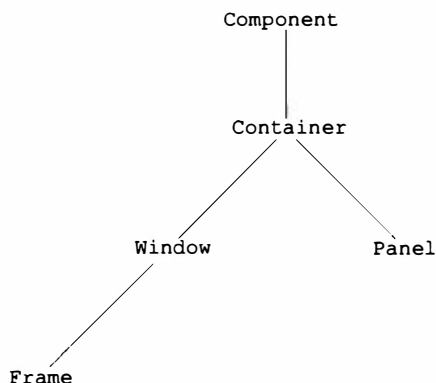


Рис. 25.1. Иерархия классов Panel и Frame

Класс Component

На вершине рассматриваемой здесь иерархии находится класс `Component`. Этот класс является абстрактным, инкапсулируя все атрибуты визуального компонента. За исключением меню, все элементы пользовательского интерфейса являются подклассами, производными от класса `Component`. В этом классе определяется больше сотни открытых методов, отвечающих за управление событиями, в том числе от клавиатуры и мыши, перемещения и изменения размеров и перерисовки окон. (Многие из этих методов уже применялись при создании апплетов в примерах глав 23 и 24.) Объект класса `Component` отвечает за запоминание текущего цвета фона и переднего плана, а также шрифта, выбранного для форматирования текста, отображаемого в окне.

Класс Container

Класс `Container` является производным от класса `Component`. Он предоставляет дополнительные методы, позволяющие вкладывать в него другие объекты класса `Component`. В объекте класса `Container` могут храниться и другие объекты этого же класса, поскольку они сами являются экземплярами класса `Component`. Это дает возможность построить многоуровневую систему вложенности. Контейнер отвечает за компоновку (т.е. расположение любых компонентов), которую он содержит. Это достигается с помощью различных диспетчеров компоновки, речь о которых пойдет в главе 26.

Класс Panel

Класс `Panel` является конкретным и производным от класса `Container`. Таким образом, класс `Panel` можно рассматривать как представляющий рекурсивно вложенный, конкретный экранный компонент. Кроме того, класс `Panel` служит суперклассом для класса `Applet`. Когда данные, выводимые на экран, направляются в апплет, они воспроизводятся на поверхности объекта типа `Panel`. По существу, класс `Panel` представляет окно без строк заголовка и меню, а также ограничивающей рамки. Именно

поэтому указанные элементы не видны, когда апплет запускается на выполнение в окне браузера. Когда же апплет запускается в средстве просмотра апплетов, то последнее предоставляет строку заголовка и ограничивающую рамку для окна.

В объект класса `Panel` могут быть добавлены и другие компоненты с помощью метода `add()`, наследуемого от класса `Container`. Как только эти компоненты будут введены, их можно будет размещать и изменять их размеры вручную, используя методы `setLocation()`, `setSize()`, `setPreferredSize()` или `setBounds()`, определенные в классе `Component`.

Класс `Window`

В этом классе создается *окно верхнего уровня*, которое не содержится в другом объекте, а располагается непосредственно на рабочем столе. Как правило, создавать объекты класса `Window` непосредственно не требуется. Вместо этого обычно используется подкласс `Frame`, производный от класса `Window`.

Класс `Frame`

Этот класс инкапсулирует то, что обычно воспринимается как окно. Этот класс является производным от класса `Window` и представляет окно в виде фрейма со строками заголовка и меню, обрамлением и элементами управления размерами окна. Внешний вид фрейма типа `Frame` может отличаться в зависимости от конкретной среды. На рисунках, приведенных в данной книге, представлены моментальные снимки экрана, отражающие особенности разных сред.

Класс `Canvas`

В связи с тем, что класс `Canvas` не является частью иерархии апплетов или обрамляющих окон, он может показаться на первый взгляд не особенно полезным. Этот класс является производным от класса `Component` и инкапсулирует пустое окно, в котором можно рисовать и воспроизводить содержимое. Пример применения класса `Canvas` будет приведен далее в этой книге.

Работа с обрамляющими окнами

После апплета тип окон, которые придется чаще всего создавать на основе библиотеки AWT, является производным от класса `Frame`. Этот класс служит для создания дочерних окон в апплетах, а также окон верхнего уровня или дочерних окон для автономных прикладных программ. Как упоминалось ранее, этот класс позволяет создавать окна в стандартном стиле.

Ниже приведены два конструктора класса `Frame`.

```
Frame() throws HeadlessException  
Frame(String заголовок) throws HeadlessException
```

В первой форме конструктора создается стандартное окно без заголовка, а во второй форме — окно с указанным *заголовком*. Следует, однако, иметь в виду, что задать

конкретные размеры окна нельзя, — их следует устанавливать после его создания. При попытке создать экземпляр класса `Frame` в среде, где не поддерживается взаимодействие с пользователем, генерируется исключение типа `HeadlessException`. Имеется ряд ключевых методов, которые придется вызывать, работая с обрамляющими окнами типа `Frame`. Эти методы рассматриваются в последующих разделах.

Установка размеров окна

Метод `setSize()` используется для установки размеров окна. Ниже приведена общая форма этого метода.

```
void setSize(int новая_ширина, int новая_высота)
void setSize(Dimension новые_размеры)
```

Новые размеры окна обозначаются параметрами *новая_ширина* и *новая_высота* или же в полях `width` и `height` объекта типа `Dimension`, передаваемого в качестве параметра *новые_размеры*. Размеры окна задаются в пикселях.

Метод `getSize()` вызывается для получения текущих размеров окна. Ниже показано, выглядит одна из его общих форм. Этот метод возвращает текущий размер окна в полях `width` и `height` объекта класса `Dimension`.

```
Dimension getSize()
```

Соккрытие и отображение окна

Как только обрамляющее окно будет создано, оно останется невидимым до тех пор, пока не будет вызван метод `setVisible()`. Ниже приведена его общая форма.

```
void setVisible(boolean признак_видимости)
```

Компонент становится видимым, если в качестве аргумента *признак_видимости* методу `setVisible()` передается логическое значение `true`. В противном случае компонент остается скрытым.

Установка заголовка окна

Заголовок обрамляющего окна можно изменить, вызвав метод `setTitle()`, общая форма которого приведена ниже, где параметр *новый_заголовок* обозначает устанавливаемый в окне новый заголовок.

```
void setTitle(String новый_заголовок)
```

Заккрытие обрамляющего окна

При использовании обрамляющего окна прикладная программа должна удалять окно с экрана после его закрытия, вызывая метод `setVisible(false)`. Чтобы перехватить событие закрытия окна, следует реализовать метод `windowClosing()` из интерфейса `WindowListener`. В методе `windowClosing()` следует удалить окно с экрана. Этот прием демонстрируется в примере, приведенном в следующем разделе.

Создание обрамляющего окна в апплете, построенном на основе библиотеки AWT

Получить окно можно, создав экземпляр класса `Frame`, но поступать именно таким образом приходится редко, поскольку пользы от такого окна немного. В частности, нельзя не только принимать или обрабатывать события, происходящие в таком окне, но и просто выводить в него данные. Поэтому на практике придется в основном создавать подклассы, производные от класса `Frame`. Это даст возможность переопределять методы из класса `Frame` и обеспечивать обработку событий.

Создать новое обрамляющее окно в апплете, построенном на основе библиотеки AWT, на самом деле совсем не трудно. Сначала создается подкласс, производный от класса `Frame`. Затем переопределяется любой стандартный метод апплета, например `init()`, `start()` и `stop()`, чтобы отображать или скрывать обрамляющее окно (или фрейм) по мере надобности. И наконец, реализуется метод `windowClosing()`.

Как только будет определен подкласс, производный от класса `Frame`, появится возможность создать объект этого класса. Это приведет к появлению обрамляющего окна, которое, однако, будет первоначально невидимым. Сделать его видимым можно, вызвав метод `setVisible()`. Окно создается с размерами по высоте и ширине, выбираемыми по умолчанию. Размеры окна можно установить и явным образом, вызвав метод `setSize()`.

В приведенном ниже примере апплета создается подкласс `SampleFrame`, производный от класса `Frame`. Экземпляр этого класса создается в виде окна в методе `init()` из класса `AppletFrame`. Обратите внимание на то, что в классе `SampleFrame` вызывается конструктор класса `Frame`. Это позволяет создать стандартное обрамляющее окно с заголовком, передаваемым в качестве параметра *заголовок*. В данном примере апплета методы `start()` и `stop()` переопределяются таким образом, чтобы показывать или скрывать дочернее окно соответственно. Это дает возможность автоматически удалить окно, когда прерывается выполнение апплета, закрывается окно или осуществляется переход на другую страницу, если апплет выполняется в окне браузера. А когда браузер возвращается к апплету, отображается также дочернее окно.

```
// Создать дочернее окно в апплете
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

/*
  <applet code="AppletFrame" width=300 height=50>
  </applet>
*/

// создать подкласс, производный от класса Frame
class SampleFrame extends Frame {
    SampleFrame(String title) {
        super(title);

        // создать объект для обработки событий в окне
        MyWindowAdapter adapter = new MyWindowAdapter(this);

        // зарегистрировать его в качестве приемника событий
        addWindowListener(adapter);
    }
}
```

```

    public void paint(Graphics g) {
        g.drawString("This is in frame window", 10, 40);
        // Это сообщение выводится в обрамляющем окне
    }
}

class MyWindowAdapter extends WindowAdapter {
    SampleFrame sampleFrame;

    public MyWindowAdapter(SampleFrame sampleFrame) {
        this.sampleFrame = sampleFrame;
    }

    public void windowClosing(WindowEvent we) {
        sampleFrame.setVisible(false);
    }
}

// создать обрамляющее окно
public class AppletFrame extends Applet {
    Frame f;

    public void init() {
        f = new SampleFrame("A Frame Window"); // Обрамляющее окно
        f.setSize(250, 250);
        f.setVisible(true);
    }

    public void start() {
        f.setVisible(true);
    }

    public void stop() {
        f.setVisible(false);
    }

    public void paint(Graphics g) {
        g.drawString("This is in applet window ", 10, 20);
        // Это сообщение выводится в окне апплета
    }
}

```

Пример выполнения данного апплета приведен на рис. 25.2.

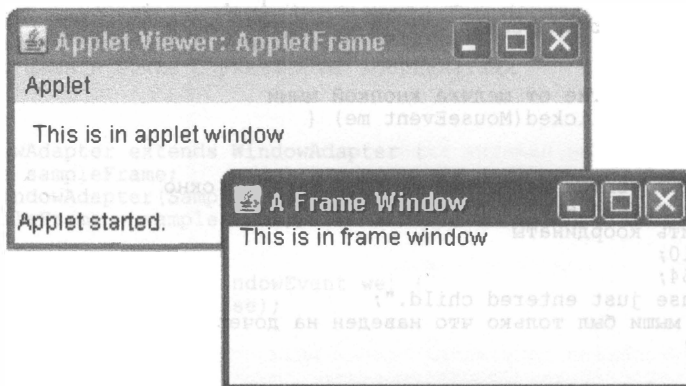


Рис. 25.2. Создание дочернего окна в апплете

Обработка событий в обрамляющем окне

Класс `Frame` является производным от класса `Component`, поэтому наследует все, что определено в классе `Component`. Это означает, что обращаться с обрамляющим окном можно таким же образом, как и с главным окном апплета. В частности, для вывода данных в окне можно переопределить метод `paint()`, а для восстановления окна и добавления к нему обработчика событий — вызвать метод `repaint()`. Всякий раз, когда в окне происходят события, вызываются обработчики событий, определенные для этого окна. В каждом окне обрабатываются свои события. Например, в приведенном ниже апплете создается окно, реагирующее на события от мыши. Главное окно апплета также реагирует на события от мыши. Поэкспериментировав с этой программой, вы обнаружите, что события от мыши посылаются тому окну, в котором они происходят.

```
// обработать события от мыши в дочернем окне и окне апплета
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="WindowEvents" width=300 height=50>
</applet>
*/

// создать подкласс, производный от класса Frame
class SampleFrame extends Frame
implements MouseListener, MouseMotionListener {

    String msg = "";
    int mouseX=10, mouseY=40;
    int movX=0, movY=0;

    SampleFrame(String title) {
        super(title);
        // зарегистрировать этот объект в качестве приемника
        // собственных событий от мыши
        addMouseListener(this);
        addMouseMotionListener(this);
        // создать объект для обработки событий в окне
        MyWindowAdapter adapter = new MyWindowAdapter(this);
        // зарегистрировать этот объект в качестве приемника
        // событий в окне
        addWindowListener(adapter);
    }

    // обработать событие от щелчка кнопкой мыши
    public void mouseClicked(MouseEvent me) {
    }

    // обработать событие наведения курсора мыши на окно
    public void mouseEntered(MouseEvent evtObj) {
        // сохранить координаты
        mouseX = 10;
        mouseY = 54;
        msg = "Mouse just entered child.";
        // Курсор мыши был только что наведен на дочернее окно
        repaint();
    }

    // обработать событие отведения курсора мыши от окна
    public void mouseExited(MouseEvent evtObj) {
```

```

    // сохранить координаты
    mouseX = 10;
    mouseY = 54;
    msg = "Mouse just left child window.";
    // Курсор мыши был только что отведен от дочернего окна
    repaint();
}

// обработать событие нажатия кнопки мыши
public void mousePressed(MouseEvent me) {
    // сохранить координаты
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Down"; // Кнопка мыши нажата
    repaint();
}

// обработать событие отпускания кнопки мыши
public void mouseReleased(MouseEvent me) {
    // сохранить координаты
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Up"; // Кнопка мыши отпущена
    repaint();
}

// обработать событие перетаскивания курсора мыши
public void mouseDragged(MouseEvent me) {
    // сохранить координаты
    mouseX = me.getX();
    mouseY = me.getY();
    movX = me.getX();
    movY = me.getY();
    msg = "**";
    repaint();
}

// обработать событие перемещения мыши
public void mouseMoved(MouseEvent me) {
    // сохранить координаты
    movX = me.getX();
    movY = me.getY();
    repaint(0, 0, 100, 60);
}

public void paint(Graphics g) {
    g.drawString(msg, mouseX, mouseY);
    g.drawString("Mouse at " + movX + ", " + movY, 10, 40);
    // Курсор мыши в точке с указанными координатами
}

}

class MyWindowAdapter extends WindowAdapter {
    SampleFrame sampleFrame;
    public MyWindowAdapter(SampleFrame sampleFrame) {
        this.sampleFrame = sampleFrame;
    }

    public void windowClosing(WindowEvent we) {
        sampleFrame.setVisible(false);
    }
}

// Окно апплета
public class WindowEvents extends Applet
implements MouseListener, MouseMotionListener {

```

```

SampleFrame f;
String msg = "";
int mouseX=0, mouseY=10;
int movX=0, movY=0;

// создать обрамляющее окно
public void init() {
    f = new SampleFrame("Handle Mouse Events");
    // Обработка событий от мыши
    f.setSize(300, 200);
    f.setVisible(true);
    // зарегистрировать объект в качестве приемника
    // собственных событий от мыши
    addMouseListener(this);
    addMouseMotionListener(this);
}

// удалить обрамляющее окно при остановке апплета
public void stop() {
    f.setVisible(false);
}

// показать обрамляющее окно при запуске апплета
public void start() {
    f.setVisible(true);
}

// обработать событие от щелчка кнопкой мыши
public void mouseClicked(MouseEvent me) {
}

// обработать событие наведения курсора мыши на окно
public void mouseEntered(MouseEvent me) {
    // сохранить координаты
    mouseX = 0;
    mouseY = 24;
    msg = "Mouse just entered applet window.";
    // Курсор мыши был только что наведен на окно апплета
    repaint();
}

// обработать событие отведения курсора мыши от окна
public void mouseExited(MouseEvent me) {
    // сохранить координаты
    mouseX = 0;
    mouseY = 24;
    msg = "Mouse just left applet window.";
    // Курсор мыши был только что отведен от окна апплета
    repaint();
}

// обработать событие нажатия кнопки мыши
public void mousePressed(MouseEvent me) {
    // сохранить координаты
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Down"; // Кнопка мыши нажата
    repaint();
}

// обработать событие отпускания кнопки мыши
public void mouseReleased(MouseEvent me) {
    // сохранить координаты
    mouseX = me.getX();

```

```

    mouseY = me.getY();
    msg = "Up"; // Кнопка мыши отпущена
    repaint();
}

// обработать событие перетаскивания курсора мыши
public void mouseDragged(MouseEvent me) {
    // сохранить координаты
    mouseX = me.getX();
    mouseY = me.getY();
    movX = me.getX();
    movY = me.getY();
    msg = "**";
    repaint();
}

// обработать событие перемещения мыши
public void mouseMoved(MouseEvent me) {
    // сохранить координаты
    movX = me.getX();
    movY = me.getY();
    repaint(0, 0, 100, 20);
}

// отобразить сообщение в окне апплета
public void paint(Graphics g) {
    g.drawString(msg, mouseX, mouseY);
    g.drawString("Mouse at " + movX + ", " + movY, 0, 10);
    // Курсор мыши в точке с указанными координатами
}
}

```

Пример выполнения данного апплета показан на рис. 25.3.

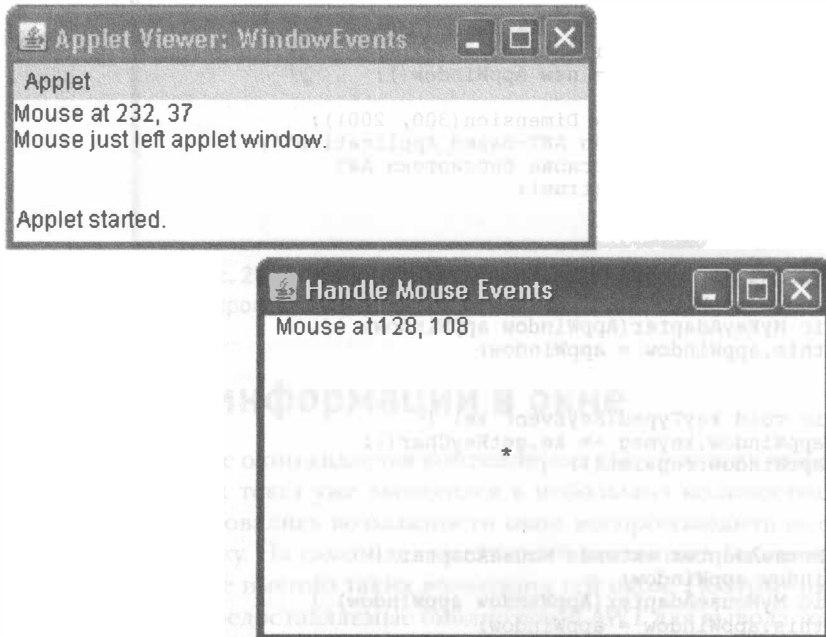


Рис. 25.3. Обработка событий от мыши в дочернем окне и окне апплета

Создание оконной прикладной программы

Создание апплетов является типичным примером применения библиотеки AWT в Java, тем не менее на основе библиотеки AWT можно также создавать автономные прикладные программы. Для этого достаточно создать экземпляр нужного окна или нескольких окон в методе `main()`. Например, в следующей прикладной программе создается обрамляющее окно, реагирующее на щелчки кнопкой мыши и нажатия клавиш.

```
// Создать прикладную программу на основе библиотеки AWT
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

// создать обрамляющее окно
public class AppWindow extends Frame {
    String keymsg = "This is a test."; // Это тест
    String mousemsg = "";
    int mouseX=30, mouseY=30;

    public AppWindow() {
        addKeyListener(new MyKeyAdapter(this));
        addMouseListener(new MyMouseAdapter(this));
        addWindowListener(new MyWindowAdapter());
    }

    public void paint(Graphics g) {
        g.drawString(keymsg, 10, 40);
        g.drawString(mousemsg, mouseX, mouseY);
    }

    // создать окно
    public static void main(String args[]) {
        AppWindow appwin = new AppWindow();

        appwin.setSize(new Dimension(300, 200));
        appwin.setTitle("An AWT-Based Application");
        // Приложение на основе библиотеки AWT
        appwin.setVisible(true);
    }
}

class MyKeyAdapter extends KeyAdapter {
    AppWindow appWindow;
    public MyKeyAdapter(AppWindow appWindow) {
        this.appWindow = appWindow;
    }

    public void keyTyped(KeyEvent ke) {
        appWindow.keymsg += ke.getKeyChar();
        appWindow.repaint();
    }
}

class MyMouseAdapter extends MouseAdapter {
    AppWindow appWindow;
    public MyMouseAdapter(AppWindow appWindow) {
        this.appWindow = appWindow;
    }

    public void mousePressed(MouseEvent me) {
```

```

        appWindow.mouseX = me.getX();
        appWindow.mouseY = me.getY();
        appWindow.mousemsg = "Mouse Down at " + appWindow.mouseX +
                               ", " + appWindow.mouseY;
        // Кнопка мыши нажата в указанной точке
        appWindow.repaint();
    }
}

class MyWindowAdapter extends WindowAdapter {
    public void windowClosing(WindowEvent we) {
        System.exit(0);
    }
}

```

Пример выполнения данной прикладной программы показан на рис. 25.4.

Однажды созданное обрамляющее окно начинает действовать самостоятельно. Обратите внимание на то, что метод `main()` завершается вызовом метода `appwin.setVisible(true)`. Но программа продолжает выполняться до тех пор, пока не закроется окно. По существу, при создании оконной прикладной программы метод `main()` служит для запуска окна верхнего уровня. После этого прикладная программа функционирует как приложение с ГПИ, а не как консольное приложение вроде тех, что были представлены ранее.

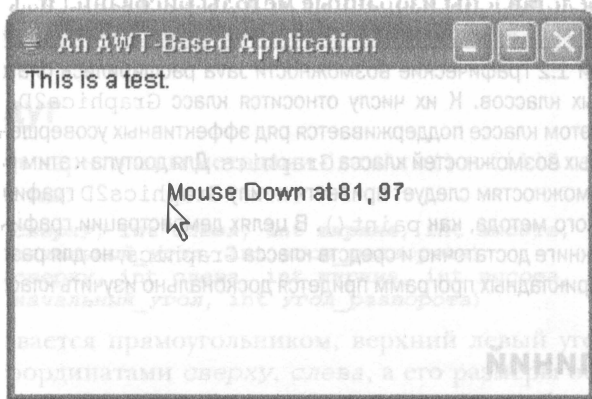


Рис. 25.4. Пример выполнения прикладной программы на основе библиотеки AWT

Отображение информации в окне

В самом общем смысле окно является контейнером для хранения информации. В предыдущих примерах текст уже выводился в небольших количествах в окне, но в них еще не использовались возможности окон воспроизводить высококачественный текст и графику. На самом деле истинный потенциал библиотеки AWT проявляется в поддержке именно таких возможностей окон. Поэтому ниже будут обсуждаться средства, предоставляемые библиотекой AWT для вывода текста, графики и шрифтового оформления. Из этого обсуждения станет очевидной эффективность и удобство этих средств.

Поддержка графики

В состав библиотеки AWT входит большое разнообразие методов, поддерживающих графику. (Эти методы поддерживаются также в окнах, создаваемых средствами библиотеки Swing.) Вся графика воспроизводится относительно окна. Это может быть главное или дочернее окно апплета или же окно автономной прикладной программы. Начало отсчета каждого окна находится в верхнем левом углу и имеет координаты 0,0, указываемые в пикселях. Весь вывод в окно выполняется в конкретном графическом контексте. *Графический контекст*, инкапсулируемый в классе Graphics, получается следующими двумя способами:

- передается в качестве аргумента методу, например `paint()` или `update()`;
- возвращается методом `getGraphics()` из класса Component.

Среди прочего, в классе Graphics определяется ряд методов для рисования различных графических объектов, в том числе линий, прямоугольников и дуг. В одних случаях графические объекты рисуются только по контуру, в других — дополнительно заполняются цветом. Графические объекты рисуются и заливаются текущим выбранным цветом, которым по умолчанию является черный. Если рисуемый графический объект выходит за пределы окна, он автоматически усекается. В этом разделе представлены избранные методы рисования из класса Graphics.

На заметку! В версии 1.2 графические возможности Java расширились благодаря внедрению нескольких новых классов. К их числу относится класс Graphics2D, расширяющий класс Graphics. В этом классе поддерживается ряд эффективных усовершенствований основных функциональных возможностей класса Graphics. Для доступа к этим расширенным функциональным возможностям следует привести к типу Graphics2D графический контекст, получаемый из такого метода, как `paint()`. В целях демонстрации графических возможностей Java в данной книге достаточно и средств класса Graphics, но для разработки полноценных графических прикладных программ придется досконально изучить класс Graphics2D.

Рисование линий

Линии рисуются методом `drawLine()`, общая форма которого приведена ниже.

```
void drawLine(int началоX, int началоY, int конецX, int конецY)
```

Метод `drawLine()` рисует линию текущим цветом от точки с координатами *началоX*, *началоY* к точке с координатами *конецX*, *конецY*.

Рисование прямоугольников

Методы `drawRect()` и `fillRect()` рисуют контурный и заполняемый прямоугольники соответственно. Ниже приведены их общие формы.

```
void drawRect(int сверху, int слева, int ширина, int высота)  
void fillRect(int сверху, int слева, int ширина, int высота)
```

Левый верхний угол прямоугольника расположен в точке с координатами *сверху*, *слева*. Размеры прямоугольника задаются в качестве параметров *ширина* и *высота*.

Для рисования прямоугольника со скругленными углами служат методы `drawRoundRect()` и `fillRoundRect()`. Ниже приведены их общие формы.

```
void drawRoundRect(int сверху, int слева, int ширина, int высота,
                  int диаметрX, int диаметрY)
void fillRoundRect(int сверху, int слева, int ширина, int высота,
                  int диаметрX, int диаметрY)
```

Прямоугольники, нарисованные этими методами, будут иметь скругленные углы. Диаметр скругления по оси X обозначается параметром *диаметрX*, а диаметр скругления дуги по оси Y — параметром *диаметрY*.

Рисование эллипсов и окружностей

Для рисования эллипса служит метод `drawOval()`, а для его заливки — метод `fillOval()`. Общие формы этих методов выглядят следующим образом:

```
void drawOval(int сверху, int слева, int ширина, int высота)
void fillOval(int сверху, int слева, int ширина, int высота)
```

Эллипс рисуется внутри ограничивающего прямоугольника, верхний левый угол которого имеет координаты *сверху*, *слева*, а размеры обозначаются параметрами *ширина* и *высота*. Чтобы нарисовать круг, следует указать ограничивающий квадрат, т.е. прямоугольник с одинаковыми значениями параметров *ширина* и *высота*.

Рисование дуг

Дуги могут быть нарисованы методами `drawArc()` и `fillArc()`, общие формы которых выглядят так:

```
void drawArc(int сверху, int слева, int ширина, int высота,
             int начальный_угол, int угол_разворота)
void fillArc(int сверху, int слева, int ширина, int высота,
            int начальный_угол, int угол_разворота)
```

Дуга ограничивается прямоугольником, верхний левый угол которого находится в точке с координатами *сверху*, *слева*, а его размеры обозначаются параметрами *ширина* и *высота*. Дуга рисуется из положения, обозначаемого параметром *начальный_угол*, на угловое расстояние, определяемое параметром *угол_разворота*. Углы указываются в градусах. Нуль градусов соответствует горизонтали в положении часовой стрелки, показывающей три часа. Дуга рисуется в направлении против часовой стрелки, если значение параметра *угол_разворота* положительно, и по часовой стрелке, если значение параметра *угол_разворота* отрицательно. Таким образом, чтобы нарисовать дугу от 12 до 6 часов, следует указать *начальный_угол* равным 90°, а *угол_разворота* — 180°.

Рисование многоугольников

Используя методы `drawPolygon()` и `fillPolygon()`, можно рисовать фигуры произвольной формы. Ниже приведены общие формы этих методов.

```
void drawPolygon(int x[], int y[], int количество_точек)
void fillPolygon(int x[], int y[], int количество_точек)
```

Конечные точки многоугольника указываются парами координат в массивах *x* и *y*. Точки с координатами в массивах *x* и *y* обозначаются параметром *количество_точек*. Имеются альтернативные формы этих методов, где многоугольник определяется объектом класса *Polygon*.

Демонстрация методов рисования

Описанные выше методы рисования демонстрируются в следующем примере апплета:

```
// Нарисовать графические элементы
import java.awt.*;
import java.applet.*;
/*
   <applet code="GraphicsDemo" width=350 height=700>
   </applet>
*/
public class GraphicsDemo extends Applet {
    public void paint(Graphics g) {

        // нарисовать линии
        g.drawLine(0, 0, 100, 90);
        g.drawLine(0, 90, 100, 10);
        g.drawLine(40, 25, 250, 80);

        // нарисовать прямоугольники
        g.drawRect(10, 150, 60, 50);
        g.fillRect(100, 150, 60, 50);
        g.drawRoundRect(190, 150, 60, 50, 15, 15);
        g.fillRoundRect(280, 150, 60, 50, 30, 40);

        // нарисовать эллипсы и окружности
        g.drawOval(10, 250, 50, 50);
        g.fillOval(90, 250, 75, 50);
        g.drawOval(190, 260, 100, 40);

        // нарисовать дуги
        g.drawArc(10, 350, 70, 70, 0, 180);
        g.fillArc(60, 350, 70, 70, 0, 75);

        // нарисовать многоугольник
        int xpoints[] = {10, 200, 10, 200, 10};
        int ypoints[] = {450, 450, 650, 650, 450};
        int num = 5;

        g.drawPolygon(xpoints, ypoints, num);
    }
}
```

Пример выполнения данного апплета показан на рис. 25.5.

Изменение размеров графики

Нередко возникает потребность установить такие размеры графического объекта, чтобы вписать его в текущие размеры того окна, в котором он рисуется. Для этого следует получить текущие размеры окна, вызвав метод *getSize()* для объекта окна. Этот метод возвратит размеры окна, инкапсулируя их в объекте класса *Dimension*. Получив текущие размеры окна, можно соответственно масштабировать графику, рисуемую в этом окне.

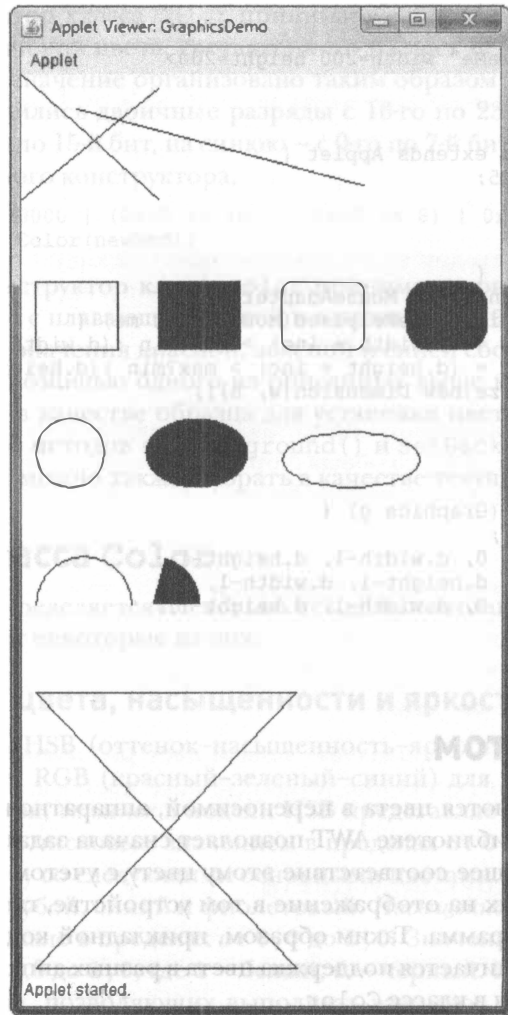


Рис. 25.5. Пример рисования графики в окне апплета

Для демонстрации этого приема рассмотрим пример апплета, где исходный квадрат размерами 200×200 пикселей будет увеличиваться на 25 пикселей после каждого щелчка кнопкой мыши до тех пор, пока не достигнет размеров 500×500 пикселей. А в результате последующего щелчка установятся исходные размеры квадрата 200×200 пикселей, и весь процесс начнется с самого начала.

Прямоугольник рисуется по внутренней границе окна, а внутри прямоугольника — знак X, чтобы таким образом заполнить окно. Этот апплет можно запустить на выполнение в средстве просмотра апплетов **appletviewer**, но нельзя в окне браузера.

```
// Изменение размеров графики, чтобы вписать ее в текущие размеры окна
import java.applet.*;
import java.awt.*;
```

```

import java.awt.event.*;
/*
<applet code="ResizeMe" width=200 height=200>
</applet>
*/

public class ResizeMe extends Applet {
    final int inc = 25;
    int max = 500;
    int min = 200;
    Dimension d;
    public ResizeMe() {
        addMouseListener(new MouseAdapter() {
            public void mouseReleased(MouseEvent me) {
                int w = (d.width + inc) > max?min : (d.width + inc);
                int h = (d.height + inc) > max?min : (d.height + inc);
                setSize(new Dimension(w, h));
            }
        });
    }

    public void paint(Graphics g) {
        d = getSize();
        g.drawLine(0, 0, d.width-1, d.height-1);
        g.drawLine(0, d.height-1, d.width-1, 0);
        g.drawRect(0, 0, d.width-1, d.height-1);
    }
}

```

Работа с цветом

В Java поддерживаются цвета в переносимой, аппаратно-независимой форме. Цветовая система в библиотеке AWT позволяет сначала задать какой угодно цвет, а затем найти наилучшее соответствие этому цвету с учетом аппаратных ограничений, накладываемых на отображение в том устройстве, где выполняется апплет или прикладная программа. Таким образом, прикладной код не должен зависеть от того, насколько отличается поддержка цвета в разных аппаратных устройствах. Цвет инкапсулируется в классе `Color`.

Как пояснялось в главе 23, в классе `Color` определяется несколько констант (вроде `Color.black`) для описания наиболее употребительных используемых цветов. Имеется также возможность создавать свои цвета, используя один из доступных конструкторов цвета. Ниже приведены три наиболее часто используемые формы конструкторов класса `Color`.

```

Color(int красный, int зеленый, int синий)
Color(int значение цвета RGB)
Color(float красный, float зеленый, float синий)

```

Первый конструктор данного класса принимает три аргумента, задающие цвет в определенном сочетании красной, зеленой и синей составляющих. Значения этих составляющих должны находиться в пределах от 0 до 255, как показано ниже.

```
new Color(255, 100, 100); // светло-красный цвет
```

Второй конструктор класса `Color` принимает единственный аргумент в виде целочисленного значения цвета, составленного из трех основных цветов (RGB). Это целочисленное значение организовано таким образом, чтобы на красную составляющую приходились двоичные разряды с 16-го по 23-й бит, на зеленую составляющую — с 8-го по 15-й бит, на синюю — с 0-го по 7-й бит. Ниже приведен пример применения такого конструктора.

```
int newRed = (0xff000000 | (0xc0 << 16) | (0x00 << 8) | 0x00);
Color darkRed = new Color(newRed);
```

И последний конструктор класса `Color` принимает три значения составляющих цвета в формате с плавающей точкой и в пределах от 0,0 до 1,0, обозначающих относительные значения красной, зеленой и синей составляющих цвета.

Составив цвет с помощью одного из описанных выше конструкторов, можно воспользоваться им в качестве образца для установки цвета переднего плана и/или фона с помощью методов `setForeground()` и `setBackground()`, описанных в главе 23. Этот цвет можно также выбрать в качестве текущего для рисования.

Методы из класса `Color`

В классе `Color` определяется несколько методов, помогающих манипулировать цветами. Рассмотрим некоторые из них.

Использование цвета, насыщенности и яркости

Цветовая модель HSB (оттенок–насыщенность–яркость) представляет собой альтернативу модели RGB (красный–зеленый–синий) для указания конкретного цвета. Образно говоря, *оттенок* в модели HSB представляет собой цветовой круг. Он может быть задан числовым значением в пределах от 0,0 до 1,0, приблизительно образующих радуугу со следующими основными цветами: красный, оранжевый, желтый, зеленый, голубой, синий и фиолетовый. *Насыщенность* определяет интенсивность цвета по шкале в пределах от 0,0 до 1,0. Значения *яркости* также находятся в пределах от 0,0 до 1,0, где 1 — белый, 0 — черный. В классе `Color` поддерживаются два метода, позволяющих выполнять взаимное преобразование цвета из моделей RGB и HSB. Ниже приведены общие формы этих методов.

```
static int HSBtoRGB(float оттенок, float насыщенность, float яркость)
static float[] RGBtoHSB(int красный, int зеленый, int синий,
    float значения[])
```

Метод `HSBtoRGB()` возвращает упакованное значение цвета RGB, совместимое с конструктором `Color(int)`. Метод `RGBtoHSB()` возвращает массив чисел с плавающей точкой, представляющих значения цвета HSB, соответствующие составляющим цвета RGB. Если массив *значения* не пуст, то он принимает заданные значения цвета HSB и возвращается. В противном случае создается новый массив, в котором возвращаются значения цвета HSB. Но в любом случае в элементе этого массива по индексу 0 содержится цвет, в элементе по индексу 1 — насыщенность, в элементе по индексу 2 — яркость.

Методы `getRed()`, `getGreen()`, `getBlue()`

Вызвав методы `getRed()`, `getGreen()` и `getBlue()`, можно получить красную, зеленую и синюю составляющие цвета по отдельности. Ниже приведены общие формы этих методов.

```
int getRed()
int getGreen()
int getBlue()
```

Каждый из этих методов возвращает соответствующую составляющую цвета RGB, извлекаемую из младших 8 битов целого числа в вызывающем объекте типа `Color`.

Метод `getRgba()`

Для получения упакованного представления цвета RGB предусмотрен метод `getRed()`. Ниже приведена его общая форма. Значение, возвращаемое этим методом, организовано описанным выше образом.

```
int getRGB()
```

Установка текущего цвета графики

По умолчанию графические объекты рисуются текущим цветом переднего плана. Этот цвет можно изменить, вызвав метод `setColor()` из класса `Graphics`:

```
void setColor(Color новый_цвет)
```

где параметр *новый_цвет* обозначает новый цвет рисования. Вызвав метод `getColor()`, можно получить текущий установленный цвет, как показано ниже.

```
Color getColor()
```

Апплет, демонстрирующий цвета

В следующем примере апплета составляется несколько цветов, с помощью которых воспроизводятся различные графические объекты:

```
// Продемонстрировать цвета
import java.awt.*;
import java.applet.*;
/*
  <applet code="ColorDemo" width=300 height=200>
  </applet>
*/

public class ColorDemo extends Applet {
    // нарисовать линии
    public void paint(Graphics g) {
        Color c1 = new Color(255, 100, 100);
        Color c2 = new Color(100, 255, 100);
        Color c3 = new Color(100, 100, 255);

        g.setColor(c1);
        g.drawLine(0, 0, 100, 100);
        g.drawLine(0, 100, 100, 0);

        g.setColor(c2);
        g.drawLine(40, 25, 250, 180);
        g.drawLine(75, 90, 400, 400);
    }
}
```

```

g.setColor(c3);
g.drawLine(20, 150, 400, 40);
g.drawLine(5, 290, 80, 19);

g.setColor(Color.red);
g.drawOval(10, 10, 50, 50);
g.fillOval(70, 90, 140, 100);

g.setColor(Color.blue);
g.drawOval(190, 10, 90, 30);
g.drawRect(10, 10, 60, 50);

g.setColor(Color.cyan);
g.fillRect(100, 10, 60, 50);
g.drawRoundRect(190, 10, 60, 50, 15, 15);
}
}

```

Установка режима рисования

Режим рисования определяет, каким образом графические объекты рисуются в окне. По умолчанию новое содержимое, выводимое в окне, замещает любое существующее в нем содержимое. Но, вызвав метод `setXORMode()`, можно также получить новые графические объекты, объединенные с помощью логической операции исключающее ИЛИ с предыдущим содержимым окна. Ниже приведена общая форма этого метода.

```
void setXORMode(Color xorColor)
```

Здесь *xorColor* обозначает цвет, объединяемый с помощью логической операции исключающее ИЛИ с содержимым окна во время рисования. Преимущество режима рисования с объединением содержимого по исключающему ИЛИ состоит в том, что он гарантирует видимость нового объекта независимо от того, каким цветом был нарисован прежний объект.

Чтобы вернуться к методу рисования с перекрытием, нужно вызвать метод `setPaintMode()` следующим образом:

```
void setPaintMode()
```

Как правило, режим рисования с замещением графических объектов применяется для обычного вывода, а режим рисования с объединением содержимого по исключающему ИЛИ — для специальных целей. В приведенном ниже примере апплета отображается перекрестье, отслеживающее курсор мыши. Вертикальная и горизонтальная черточки перекрестья объединяются по исключающему ИЛИ в режиме рисования и поэтому остаются видимыми в окне независимо от цвета фона.

```

// Продемонстрировать применение режима рисования
// с объединением содержимого по исключающему ИЛИ
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="XOR" width=400 height=200>
</applet>

```

```

*/
public class XOR extends Applet {
    int chsX=100, chsY=100;
    public XOR() {
        addMouseListener(new MouseMotionAdapter() {
            public void mouseMoved(MouseEvent me) {
                int x = me.getX();
                int y = me.getY();
                chsX = x-10;
                chsY = y-10;
                repaint();
            }
        });
    }

    public void paint(Graphics g) {
        g.drawLine(0, 0, 100, 100);
        g.drawLine(0, 100, 100, 0);
        g.setColor(Color.blue);
        g.drawLine(40, 25, 250, 180);
        g.drawLine(75, 90, 400, 400);
        g.setColor(Color.green);
        g.drawRect(10, 10, 60, 50);
        g.fillRect(100, 10, 60, 50);
        g.setColor(Color.red);
        g.drawRoundRect(190, 10, 60, 50, 15, 15);
        g.fillRoundRect(70, 90, 140, 100, 30, 40);
        g.setColor(Color.cyan);
        g.drawLine(20, 150, 400, 40);
        g.drawLine(5, 290, 80, 19);

        // объединить черточки перекрестья по исключающему ИЛИ
        g.setXORMode(Color.black);
        g.drawLine(chsX-10, chsY, chsX+10, chsY);
        g.drawLine(chsX, chsY-10, chsX, chsY+10);
        g.setPaintMode();
    }
}

```

Пример выполнения данного апплета показан на рис. 25.6.

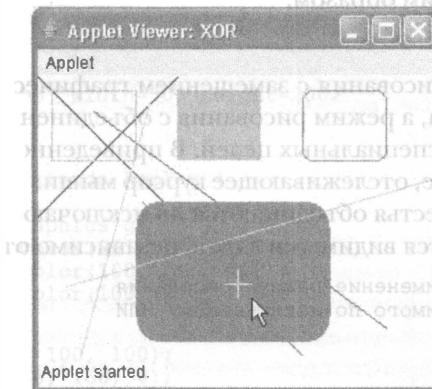


Рис. 25.6. Пример применения режима рисования с объединением содержимого по исключающему ИЛИ

Работа со шрифтами

В библиотеке AWT поддерживается немало печатных шрифтов. Ранее традиционные для печати шрифты были важной частью формируемых на компьютере документов и изображений. Библиотека AWT обеспечивает удобство обращения со шрифтами, абстрагируя операции с ними и позволяя динамически выбирать их.

У шрифтов имеется наименование семейства, логическое имя шрифта и наименование гарнитуры. *Наименование семейства* является обобщенным названием шрифта, например Courier. *Логическое имя* определяет название, связываемое с конкретным шрифтом во время выполнения, например Monospaced, а *наименование гарнитуры* — конкретный шрифт, например Courier Italic.

Шрифты инкапсулированы в классе Font. Некоторые методы, определенные в классе Font, перечислены в табл. 25.2. В этом классе определяется также ряд статических методов.

Таблица 25.2. Избранные методы из класса Font

Метод	Описание
static Font decode(String строка)	Возвращает шрифт по имени, обозначаемому параметром <i>строка</i>
boolean equals(Object объект_шрифта)	Возвращает логическое значение true , если вызывающий объект содержит такой же шрифт, как и указанный в качестве параметра <i>объект_шрифта</i>
String getFamily()	Возвращает наименование семейства шрифтов, к которому относится вызывающий шрифт
static Font getFont(String свойство)	Возвращает шрифт, связанный с указанным системным <i>свойством</i> . Если указанное <i>свойство</i> отсутствует, то возвращается пустое значение null
static Font getFont(String свойство, Font стандартный_шрифт)	Возвращает шрифт, связанный с указанным системным <i>свойством</i> . Если указанное <i>свойство</i> отсутствует, то возвращается заданный <i>стандартный_шрифт</i>
String getFontName()	Возвращает название гарнитуры вызывающего шрифта
String getName()	Возвращает логическое имя вызывающего шрифта
int getSize()	Возвращает размер вызывающего шрифта в пунктах
int getStyle()	Возвращает значения начертаний вызывающего шрифта
int hashCode()	Возвращает хеш-код, связанный с вызывающим объектом
boolean isBold()	Возвращает логическое значение true , если шрифт содержит значение полужирного начертания BOLD , а иначе — логическое значение false
boolean isItalic()	Возвращает логическое значение true , если шрифт содержит значение наклонного начертания ITALIC , а иначе — логическое значение false
boolean isPlain()	Возвращает логическое значение true , если шрифт содержит значение простого начертания PLAIN , а иначе — логическое значение false
String toString()	Возвращает строковый эквивалент вызывающего шрифта

В классе `Font` определены переменные, перечисленные в табл. 25.3.

Таблица 25.3. Переменные, определенные в классе `Font`

Переменная	Значение
<code>String name</code>	Имя шрифта
<code>float pointSize</code>	Размер шрифта в пунктах
<code>int size</code>	Размер шрифта в точках
<code>int style</code>	Начертание шрифта

Определение доступных шрифтов

Для работы со шрифтами зачастую нужно знать, какие именно шрифты установлены и доступны на конкретном компьютере. Для получения этих сведений служит метод `getAvailableFontFamilyNames()`, определенный в классе `GraphicsEnvironment`, общая форма которого приведена ниже.

`String[] getAvailableFontFamilyNames()`

Этот метод возвращает массив символьных строк, содержащих наименования доступных семейств шрифтов. Кроме того, в классе `GraphicsEnvironment` определен метод `getAllFonts()`, общая форма которого показана ниже. Этот метод возвращает массив объектов типа `Font`, описывающих все доступные шрифты.

`Font[] getAllFonts()`

Эти методы являются членами класса `GraphicsEnvironment`, поэтому для их вызова потребуется ссылка на объект данного класса. Эту ссылку можно получить, вызвав статический метод `getLocalGraphicsEnvironment()`, определенный в классе `GraphicsEnvironment`:

`static GraphicsEnvironment getLocalGraphicsEnvironment()`

В следующем примере апплета показано, как получить наименования всех доступных семейств шрифтов:

```
// Отобразить доступные шрифты
/*
  <applet code="ShowFonts" width=550 height=60>
  </applet>
*/

import java.applet.*;
import java.awt.*;

public class ShowFonts extends Applet {
    public void paint(Graphics g) {
        String msg = "";
        String FontList[];

        GraphicsEnvironment ge =
            GraphicsEnvironment.getLocalGraphicsEnvironment();
        FontList = ge.getAvailableFontFamilyNames();
        for(int i = 0; i < FontList.length; i++)
```

```

        msg += FontList[i] + " ";
        g.drawString(msg, 4, 16);
    }
}

```

Пример выполнения данного апплета показан на рис. 25.7. Но имейте в виду, что список шрифтов может у вас отличаться от показанного на рис. 25.7.

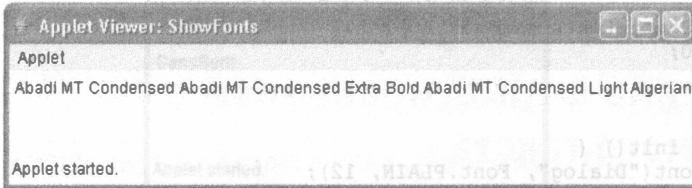


Рис. 25.7. Пример получения списка доступных шрифтов

Создание и выбор шрифта

Чтобы выбрать новый шрифт, следует сначала создать объект класса `Font`, описывающий шрифт. Один из конструкторов класса `Font` имеет следующую общую форму:

`Font(String имя_шрифта, int начертание, int размер_в_пунктах)`

где параметр *имя_шрифта* обозначает наименование нужного шрифта. Это наименование может быть задано с помощью логического имени или названия гарнитуры. Во всех средах Java поддерживаются следующие шрифты: `Dialog`, `DialogInput`, `Sans Serif`, `Serif` и `Monospaced`. Шрифт `Dialog` применяется в диалоговых окнах системы и выбирается по умолчанию в отсутствие каких-нибудь других установленных шрифтов. Имеется также возможность воспользоваться любым другим шрифтом, который поддерживается в конкретной исполняющей среде. Тем не менее такие шрифты могут быть доступны не всегда.

Стиль (или начертание) шрифта обозначается параметром *начертание*. Значение начертания шрифта может состоять из одной или нескольких следующих констант: `Font.PLAIN` (простое начертание), `Font.BOLD` (полужирное начертание) и `Font.ITALIC` (наклонное начертание). Начертания шрифтов можно сочетать, объединяя их с помощью операции логическое ИЛИ. Например, сочетание констант `Font.BOLD` и `Font.ITALIC` задает полужирное наклонное начертание шрифта. И наконец, размер шрифта в пунктах обозначается параметром *размер_в_пунктах*.

Чтобы воспользоваться только что созданным шрифтом, следует выбрать его с помощью метода `setFont()`, определенного в классе `Component`. Ниже приведена общая форма этого метода, где *объект_шрифта* обозначает объект, содержащий требуемый шрифт.

`void setFont(Font объект_шрифта)`

В приведенном ниже примере апплета выводятся образцы каждого из стандартных шрифтов. Всякий раз, когда в окне апплета производится щелчок кнопкой мыши, выбирается новый шрифт и отображается его наименование.

```

// Показать шрифты
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
  <applet code="SampleFonts" width=200 height=100>
  </applet>
*/

public class SampleFonts extends Applet {
    int next = 0;
    Font f;
    String msg;

    public void init() {
        f = new Font("Dialog", Font.PLAIN, 12);
        msg = "Dialog";
        setFont(f);
        addMouseListener(new MyMouseAdapter(this));
    }

    public void paint(Graphics g) {
        g.drawString(msg, 4, 20);
    }
}

class MyMouseAdapter extends MouseAdapter {
    SampleFonts sampleFonts;

    public MyMouseAdapter(SampleFonts sampleFonts) {
        this.sampleFonts = sampleFonts;
    }

    public void mousePressed(MouseEvent me) {
        // сменить шрифт после каждого щелчка кнопкой мыш
        sampleFonts.next++;
        switch(sampleFonts.next) {
            case 0:
                sampleFonts.f = new Font("Dialog", Font.PLAIN, 12);
                sampleFonts.msg = "Dialog";
                break;
            case 1:
                sampleFonts.f = new Font("DialogInput", Font.PLAIN, 12);
                sampleFonts.msg = "DialogInput";
                break;
            case 2:
                sampleFonts.f = new Font("SansSerif", Font.PLAIN, 12);
                sampleFonts.msg = "SansSerif";
                break;
            case 3:
                sampleFonts.f = new Font("Serif", Font.PLAIN, 12);
                sampleFonts.msg = "Serif";
                break;
            case 4:
                sampleFonts.f = new Font("Monospaced", Font.PLAIN, 12);
                sampleFonts.msg = "Monospaced";
                break;
        }

        if(sampleFonts.next == 4) sampleFonts.next = -1;
    }
}

```

```

        sampleFonts.setFont(sampleFonts.f);
        sampleFonts.repaint();
    }
}

```

Пример выполнения данного апплета показан на рис. 25.8.

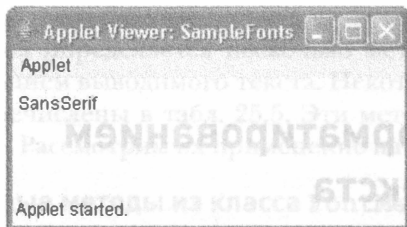


Рис. 25.8. Пример отображения образцов доступных шрифтов

Получение сведений о шрифте

Допустим, требуется получить сведения о текущем выбранном шрифте. Для этого следует сначала получить текущий шрифт, вызвав метод `getFont()`. Этот метод определен в классе `Graphics`, а его общая форма выглядит так:

Font `getFont()`

Получив текущий выбранный шрифт, можно извлечь сведения о нем, вызвав различные методы, определенные в классе `Font`. В следующем примере апплета отображаются наименование, семейство, размер и начертание выбранного в данный момент шрифта:

```

// Вывести сведения о шрифте
import java.applet.*;
import java.awt.*;

/*
 <applet code="FontInfo" width=350 height=60>
 </applet>
 */

public class FontInfo extends Applet {
    public void paint(Graphics g) {
        Font f = g.getFont();
        String fontName = f.getName();
        String fontFamily = f.getFamily();
        int fontSize = f.getSize();
        int fontStyle = f.getStyle();
        String msg = "Family: " + fontName;
        // String msg = "Семейство: " + fontName;

        msg += ", Font: " + fontFamily; // Шрифт

        msg += ", Size: " + fontSize + ", Style: "; // Размер шрифта

        if((fontStyle & Font.BOLD) == Font.BOLD)
            msg += "Bold "; // Полужирное начертание
    }
}

```



```

        if((fontStyle & Font.ITALIC) == Font.ITALIC)
            msg += "Italic "; // Наклонное начертание

        if((fontStyle & Font.PLAIN) == Font.PLAIN)
            msg += "Plain "; // Простое начертание
        g.drawString(msg, 4, 16);
    }
}

```

Управление форматированием выводимого текста

Как пояснялось выше, в Java поддерживается немало шрифтов. В большинстве из них символы не имеют одинакового размера, и поэтому большинство шрифтов пропорциональны. Высота каждого символа, длина *подстрочных элементов* (т.е. нижней части некоторых букв, например у), промежуток между горизонтальными линиями, а также размер шрифта в пунктах меняются в разных шрифтах. Все эти (и другие) атрибуты шрифтов являются переменными и не представляют особого интереса для программирующего на Java, поскольку в этом языке не требуется ручное управление почти всем выводимым текстом.

Учитывая, что размер каждого шрифта может отличаться и что шрифты могут изменяться в процессе выполнения прикладной программы, должен существовать какой-то способ определения размеров и прочих разнообразных атрибутов текущего выбранного шрифта. Например, чтобы вывести одну строку текста после другой, необходимо каким-то образом узнать высоту шрифта и количество пикселей между строками. Для этой цели в библиотеке AWT предусмотрен класс `FontMetrics`, инкапсулирующий разнообразные сведения о шрифте. Прежде всего следует определить общую терминологию, употребляемую для описания шрифтов (табл. 25.4).

Таблица 25.4. Общая терминология, употребляемая для описания шрифтов

Высота	Размер строки текста сверху вниз
Базовая линия	Линия, по которой выравниваются нижние края символов (за исключением подстрочных элементов)
Подъем	Расстояние от базовой линии до верхнего края символов
Спуск	Расстояние от базовой линии до нижнего края символов
Интерлиньяж	Расстояние между нижним и верхним краями текстовой строки текста

Как вы, должно быть, уже заметили, метод `drawString()` применялся во многих представленных ранее примерах. Этот метод выводит символьную строку текущим цветом и шрифтом, начиная с указанного местоположения. Но это местоположение находится на левом краю базовой линии символов, а не в левом верхнем углу, как это принято в других методах рисования. Типичная ошибка состоит в попытке нарисовать символьную строку в точке с теми же самыми координатами, где обычно рисуется рамка. Так, если требуется нарисовать прямоугольник,

начиная с точки, имеющей координаты 0, 0, то он появится полностью. Если же попытаться вывести символьную строку "Typesetting" (Набор текста), начиная с точки, имеющей координаты 0, 0, то появятся только подстрочные элементы букв **y**, **p** и **g**. Как станет ясно в дальнейшем, используя типографские параметры шрифта, можно определить правильное местоположение каждой отображаемой символьной строки.

В классе `FontMetrics` определяется несколько методов, которые помогают управлять форматированием выводимого текста. Некоторые из наиболее употребительных методов перечислены в табл. 25.5. Эти методы помогают правильно отобразить текст в окне. Рассмотрим их применение на ряде примеров.

Таблица 25.5. Избранные методы из класса `FontMetrics`

Метод	Описание
<code>int bytesWidth(byte b[], int начало, int количество_байтов)</code>	Возвращает ширину заданного <i>количества_байтов</i> из массива <i>b</i> , начиная с позиции <i>начало</i>
<code>int charWidth(char c[], int начало, int количество_символов)</code>	Возвращает ширину заданного <i>количества_символов</i> из массива <i>c</i> , начиная с позиции <i>начало</i>
<code>int charWidth(char c)</code>	Возвращает ширину заданного символа <i>c</i>
<code>int charWidth(int c)</code>	Возвращает ширину заданного символа <i>c</i>
<code>int getAscent()</code>	Возвращает подъем шрифта
<code>int getDescent()</code>	Возвращает спуск шрифта
<code>Font getFont()</code>	Возвращает текущий шрифт
<code>int getHeight()</code>	Возвращает высоту текстовой строки. Это значение может быть использовано для вывода многострочного текста в окне
<code>int getLeading()</code>	Возвращает пробел между строками текста
<code>int getMaxAdvance()</code>	Возвращает ширину самого широкого символа. Если эта величина недоступна, возвращается значение -1
<code>int getMaxAscent()</code>	Возвращает максимальный подъем
<code>int getMaxDescent()</code>	Возвращает максимальный спуск
<code>int[] getWidths()</code>	Возвращает ширину первых 256 символов
<code>int stringWidth(String строка)</code>	Возвращает ширину заданной <i>строки</i>
<code>String toString()</code>	Возвращает строковый эквивалент вызывающего объекта

Отображение многострочного текста

Чаще всего класс `FontMetrics` используется для определения расстояния между строками текста. Кроме того, он определяет длину строки, которую нужно отобразить. Далее будет показано, каким образом решаются эти задачи.

Для того чтобы отобразить многострочный текст, прикладная программа должна отслеживать текущую позицию выводимого текста. Всякий раз, когда встречается символ новой строки, координата Y должна быть перенесена в начало следующей строки. Когда отображается строка, координата X должна быть установлена в точку, где эта строка оканчивается. Это позволит начать вывод следующей строки таким образом, чтобы она начиналась сразу после окончания предыдущей строки.

Чтобы определить расстояние между строками, можно использовать значение, возвращаемое методом `getLeading()`. Чтобы определить общую высоту шрифта, следует добавить значение, возвращаемое методом `getAscent()`, к значению, возвращаемому методом `getDescent()`. Затем эти значения можно использовать для расположения каждой строки выводимого текста. Но во многих случаях эти значения вряд ли понадобятся по отдельности. Зачастую нужно знать лишь общую высоту строки, состоящую из суммы величин интерлиньяжа (междустрочного интервала), подъема и спуска шрифта. Эту величину можно получить, вызвав метод `getHeight()`. Координату Y следует увеличить на эту величину всякий раз, когда требуется перейти на следующую строку выводимого текста.

Чтобы начать вывод текста с того места, где был завершен вывод предыдущего текста в той же самой строке, следует знать длину в пикселях каждой отображаемой строки. Чтобы получить эту величину, следует вызвать метод `stringWidth()`. Эту величину можно использовать для вычисления координаты X всякий раз, когда выводится строка.

В приведенном ниже примере показано, как вывести многострочный текст в окне апплета, где отображается также несколько предложений на одной и той же строке. Обратите внимание на переменные `curX` и `curY`. В них отслеживается текущая позиция выводимого текста.

```
// Продемонстрировать вывод многострочного текста
import java.applet.*;
import java.awt.*;
/*
<applet code="MultiLine" width=300 height=100>
</applet>
*/

public class MultiLine extends Applet {
    int curX=0, curY=0; // текущая позиция
    public void init() {
        Font f = new Font("SansSerif", Font.PLAIN, 12);
        setFont(f);
    }

    public void paint(Graphics g) {
        FontMetrics fm = g.getFontMetrics();
        nextLine("This is on line one.", g);
        // Этот текст выводится в первой строке
        nextLine("This is on line two.", g);
        // Этот текст выводится во второй строке
        sameLine("This is on same line.", g);
        // Этот текст выводится в той же самой строке
        sameLine(" This, too.", g);
    }
}
```

```

    // И этот текст выводится в той же самой строке
    nextLine("This is on line three.", g);
    // Этот текст выводится в третьей строке
    curX = curY = 0; // установить координаты в исходное
    // состояние перед каждой перерисовкой
}

// перейти на следующую строку
void nextLine(String s, Graphics g) {
    FontMetrics fm = g.getFontMetrics();
    curY += fm.getHeight(); // перейти на следующую строку
    curX = 0;
    g.drawString(s, curX, curY);
    curX = fm.stringWidth(s); // перейти в конец строки
}

// отобразить текст в той же самой строке
void sameLine(String s, Graphics g) {
    FontMetrics fm = g.getFontMetrics();
    g.drawString(s, curX, curY);
    curX += fm.stringWidth(s); // перейти в конец строки
}
}

```

Пример выполнения данного апплета показан на рис. 25.9.

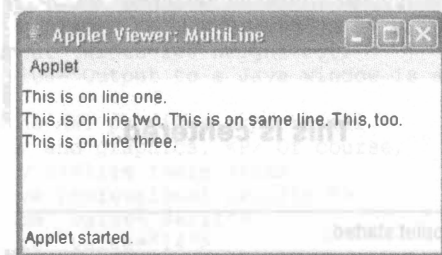


Рис. 25.9. Пример вывода многострочного текста в окне апплета

Центровка текста

В приведенном ниже примере апплета текст центруется в окне слева направо и сверху вниз. Сначала в этом апплете получаются величины подъема, спуска и ширины строки, а затем вычисляется позиция, на которой текст должен быть отображен отцентрованным.

```

// Центровка текста
import java.applet.*;
import java.awt.*;

/*
  <applet code="CenterText" width=200 height=100>
  </applet>
*/

public class CenterText extends Applet {
    final Font f = new Font("SansSerif", Font.BOLD, 18);

```

```

public void paint(Graphics g) {
    Dimension d = this.getSize();

    g.setColor(Color.white);
    g.fillRect(0, 0, d.width, d.height);
    g.setColor(Color.black);
    g.setFont(f);
    drawCenteredString("This is centered.", d.width, d.height, g);
    // Этот текст отцентрирован
    g.drawRect(0, 0, d.width-1, d.height-1);
}

public void drawCenteredString(String s, int w, int h, Graphics g) {
    FontMetrics fm = g.getFontMetrics();
    int x = (w - fm.stringWidth(s)) / 2;
    int y = (fm.getAscent() + (h - (fm.getAscent() +
        fm.getDescent()))/2);
    g.drawString(s, x, y);
}
}

```

Пример выполнения данного апплета показан на рис. 25.10.

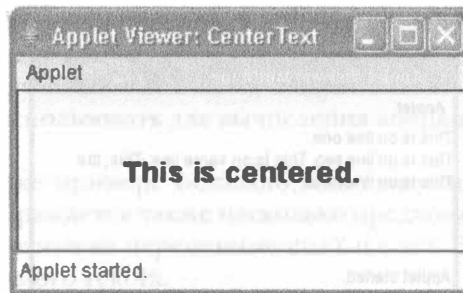


Рис. 25.10. Пример центровки текста в окне апплета

Выравнивание многострочного текста

В редакторе текст обычно отображается выровненным таким образом, чтобы один или оба его края образовывали ровную вертикальную линию. Так, в большинстве редакторов текст можно выравнивать по левому и/или по правому краю, а также по центру. В приведенном ниже примере апплета показано, как добиться выравнивания многострочного текста.

В этом апплете выравниваемые строки текста разбиваются на отдельные слова. Длина каждого слова отслеживается в текущем шрифте, и когда слово не умещается в текущей строке, то производится автоматический переход на следующую строку. Каждая завершенная строка отображается в окне, выровненной выбранным в данный момент стилем. Всякий раз, когда производится щелчок кнопкой мыши в окне апплета, стиль выравнивания изменяется. Пример выполнения данного апплета показан на рис. 25.11.

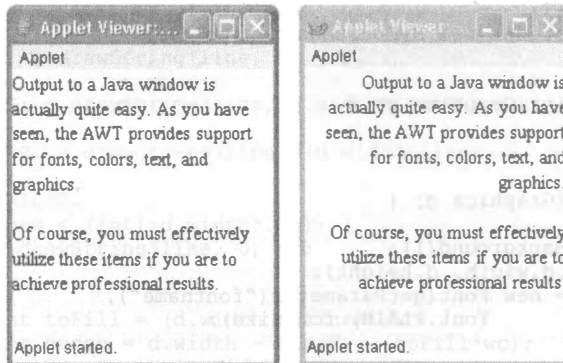


Рис. 25.11. Пример выравнивания многострочного текста в окне апплета

```
// Продемонстрировать выравнивание многострочного текста
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

/* <title>Text Layout</title>
   <applet code="TextLayout" width=200 height=200>
     <param name="text" value="Output to a Java window is actually
       quite easy.
       As you have seen, the AWT provides support for
       fonts, colors, text, and graphics. <P> Of course,
       you must effectively utilize these items
       if you are to achieve professional results.">
     <param name="fontname" value="Serif">
     <param name="fontSize" value="14">
   </applet>
*/

public class TextLayout extends Applet {
    final int LEFT = 0;
    final int RIGHT = 1;
    final int CENTER = 2;
    final int LEFTRIGHT = 3;
    int align;
    Dimension d;
    Font f;
    FontMetrics fm;
    int fontSize;
    int fh, bl;
    int space;
    String text;

    public void init() {
        setBackground(Color.white);
        text = getParameter("text");
        try {
            fontSize = Integer.parseInt(getParameter("fontSize"));
        } catch (NumberFormatException e) {
            fontSize=14;
        }
    }
}
```

```

        align = LEFT;
        addMouseListener(new MyMouseAdapter(this));
    }

    public void paint(Graphics g) {
        update(g);
    }

    public void update(Graphics g) {
        d = getSize();
        g.setColor(getBackground());
        g.fillRect(0,0,d.width, d.height);
        if(f==null) f = new Font(getParameter("fontname"),
                                Font.PLAIN, fontSize);
        g.setFont(f);

        if(fm == null) {
            fm = g.getFontMetrics();
            bl = fm.getAscent();
            fh = bl + fm.getDescent();
            space = fm.stringWidth(" ");
        }

        g.setColor(Color.black);
        StringTokenizer st = new StringTokenizer(text);
        int x = 0;
        int nextx;
        int y = 0;
        String word, sp;
        int wordCount = 0;
        String line = "";
        while (st.hasMoreTokens()) {
            word = st.nextToken();
            if(word.equals("<P>")) {
                drawString(g, line, wordCount,
                           fm.stringWidth(line), y+bl);
                line = "";
                wordCount = 0;
                x = 0;
                y = y + (fh * 2);
            }
            else {
                int w = fm.stringWidth(word);
                if(( nextx = (x+space+w)) > d.width ) {
                    drawString(g, line, wordCount,
                               fm.stringWidth(line), y+bl);
                    line = "";
                    wordCount = 0;
                    x = 0;
                    y = y + fh;
                }
                if(x!=0) {sp = " ";} else {sp = "";}
                line = line + sp + word;
                x = x + space + w;
                wordCount++;
            }
        }
        drawString(g, line, wordCount, fm.stringWidth(line), y+bl);
    }

    public void drawString(Graphics g, String line,

```

```

        int wc, int lineW, int y) {
switch(aligned) {
    case LEFT: g.drawString(line, 0, y);
                break;
    case RIGHT: g.drawString(line, d.width-lineW, y);
                break;
    case CENTER: g.drawString(line, (d.width-lineW)/2, y);
                break;
    case LEFTRIGHT:
        if(lineW < (int)(d.width*.75)) {
            g.drawString(line, 0, y);
        }
        else {
            int toFill = (d.width - lineW)/wc;
            int nudge = d.width - lineW - (toFill*wc);
            int s = fm.stringWidth(" ");
            StringTokenizer st = new StringTokenizer(line);
            int x = 0;
            while(st.hasMoreTokens()) {
                String word = st.nextToken();
                g.drawString(word, x, y);
                if(nudge>0) {
                    x = x + fm.stringWidth(word) + space + toFill +
                        toFill + 1;
                    nudge--;
                } else {
                    x = x + fm.stringWidth(word) + space + toFill;
                }
            }
        }
        break;
    }
}

}

class MyMouseAdapter extends MouseAdapter {
    TextLayout tl;
    public MyMouseAdapter(TextLayout tl) {
        this.tl = tl;
    }

    public void mouseClicked(MouseEvent me) {
        tl.align = (tl.align + 1) % 4;
        tl.repaint();
    }
}

```

Рассмотрим подробнее, каким образом действует этот апплет. Сначала в нем создается несколько констант для определения стиля выравнивания, а затем объявляется ряд переменных. Метод `init()` получает отображаемый текст. Далее размер шрифта инициализируется в блоке операторов `try/catch`, где затем устанавливается размер шрифта 14 пунктов, если параметр `fontSize` отсутствует в дескрипторе HTML-разметки. Параметру `text` присваивается длинная строка текста с дескриптором HTML `<P>` в качестве разделителя абзаца.

Метод `update()` реализует механизм выравнивания текста в данном примере. В этом методе устанавливается шрифт, а из объекта типографских параметров шрифта получается базовая линия и высота. Далее создается объект класса `StringTokenizer`, с помощью которого извлекается следующая лексема (строка,

отделенная пробелами) из строки, заданной в параметре `text`. Если следующей лексемой оказывается дескриптор `<P>`, то выполняется вертикальная прокрутка. В противном случае метод `update()` проверяет, уместится ли длина лексемы, набранной текущим шрифтом, по ширине столбца. Если строка уже заполнена текстом или же лексем для вывода больше не осталось, то строка выводится специальной версией метода `drawString()`.

Первые три случая вызова метода `drawString()` просты. В каждом из них строка, передаваемая в переменной `line`, выравнивается по левому или правому краю или же по центру столбца в зависимости от текущего стиля выравнивания. Так, если выбран стиль `LEFTRIGHT`, выравнивание производится по обоим краям строки. Это означает, что оставшееся свободное пространство нужно вычислить (как разность ширины строки и столбца) и равномерно распределить между словами. В последнем методе из класса адаптера выбирается следующий стиль выравнивания всякий раз, когда производится щелчок кнопкой мыши в окне апплета.

Применение элементов управления, диспетчеров компоновки и меню из библиотеки AWT

В этой главе продолжается обзор библиотеки AWT. Сначала будут рассмотрены стандартные элементы управления и диспетчеры компоновки. Затем речь пойдет о меню и строке меню. Далее обсуждаются два компонента верхнего уровня: обычное диалоговое окно и диалоговое окно выбора файлов. И в конце главы обработка событий будет рассмотрена под другим углом.

Элементами управления называют компоненты, которые дают пользователю возможность по-разному взаимодействовать с прикладной программой. К числу наиболее распространенных элементов управления относится экранная кнопка. *Диспетчер компоновки* автоматически размещает компоненты в контейнере. Поэтому внешний вид окна зависит как от состава элементов управления, так и от диспетчера компоновки, с помощью которого они размещаются в окне.

Кроме элементов управления, обрамляющее окно может также включать *строку меню* стандартного стиля. Каждый пункт полосы меню раскрывает меню, в котором пользователь может выбрать необходимую ему команду. Строка меню всегда располагается в верхней части окна.

Несмотря на отличия во внешнем виде, строки меню обрабатываются почти так же, как и другие элементы управления. И хотя размещать компоненты в окне можно вручную, сделать это, как правило, не так-то просто. Диспетчер компоновки выполняет эту задачу автоматически. В примерах, приведенных в начале этой главы, где рассматриваются различные элементы управления, применяется диспетчер компоновки, выбираемый по умолчанию. Он отображает компоненты в контейнере, размещая их слева направо и сверху вниз. После того как будут рассмотрены элементы управления, речь пойдет о диспетчерах компоновки, а также будет разъяснено, как лучше всего располагать элементы управления в окне.

Прежде чем перейти непосредственно к изложению материала этой главы, следует особо подчеркнуть, что в настоящее время ГПИ прикладных программ редко строятся только средствами библиотеки AWT, поскольку для этой цели в Java внедрены более эффективные библиотеки Swing и JavaFX. Тем не менее материал, представленный в этой главе, не теряет по-прежнему свою актуальность по ряду причин. Во-первых, большую часть сведений и приемов, связанных с элементами управления и обработкой событий, можно распространить и на другие библиотеки Java, предназначенные для построения ГПИ. (Как упоминалось в предыдущей главе, библиотека Swing построена на основе библиотеки AWT.) Во-вторых, обсуждаемые здесь диспетчеры компоновки могут применяться и в библиотеке Swing.

В-третьих, компоненты библиотеки AWT могут оказаться более подходящими для разработки мелких прикладных программ с ГПИ. И наконец, не исключено, что придется сопровождать или обновлять унаследованный код, в котором применяются средства из библиотеки AWT, и эта причина, вероятно, важнее всего для всех программирующих на Java.

Основные положения об элементах управления

В библиотеке AWT поддерживаются следующие типы элементов управления.

- Метки
- Экранные кнопки
- Флажки
- Списки выбора
- Списки
- Полосы прокрутки
- Элементы редактирования текста

Все эти элементы управления относятся к подклассам, производным от класса `Component`.

Ввод и удаление элементов управления

Чтобы включить элемент управления в состав окна, его нужно сначала ввести в него. Для этого необходимо создать экземпляр требуемого элемента управления, а затем ввести его в окно с помощью метода `add()`, определенного в классе `Container`. У метода `add()` имеется несколько общих форм. В примерах, приведенных в начале данной главы, используется следующая общая форма этого метода:

```
Component add(Component ссылка)
```

где параметр *ссылка* обозначает конкретную ссылку на экземпляр элемента управления, который требуется ввести. Метод `add()` возвращает ссылку на этот объект. Как только элемент управления будет введен, он появится в родительском окне при его отображении.

Иногда требуется удалить элементы управления из окна. Для этой цели служит метод `remove()`, который также определен в классе `Container`. Ниже приведена одна из его общих форм.

```
void remove(Component ссылка)
```

Здесь параметр *ссылка* обозначает конкретную ссылку на экземпляр элемента управления, который требуется удалить. Вызвав метод `removeAll()`, можно удалить все элементы управления.

Реагирование на элементы управления

За исключением меток, которые являются пассивными элементами пользовательского интерфейса, каждый элемент управления извещает о событии в тот момент, когда к нему обращается пользователь. Например, когда пользователь щелкает на экранной кнопке, наступает событие, обозначающее эту кнопку. Как правило, для обработки событий от элементов управления в прикладной программе сначала реализуется соответствующий интерфейс, а затем регистрируется получатель событий от каждого элемента управления. Как пояснялось в главе 24, после установки приемника событий извещения о событиях будут передаваться ему автоматически. В последующих разделах описывается соответствующий интерфейс для каждого из перечисленных выше элементов управления.

Исключение типа `HeadlessException`

Большинство рассматриваемых в этой главе элементов управления из библиотеки AWT имеют конструкторы, способные генерировать исключение типа `HeadlessException` при попытке создать экземпляр компонента ГПИ в неинтерактивной среде (т.е. в такой среде, где, например, нет монитора, мыши или клавиатуры). Исключение типа `HeadlessException` было внедрено в версии Java 1.4. С помощью этого исключения можно написать код, который можно приспособить к неинтерактивным средам. (Разумеется, это возможно далеко не всегда.) Исключение типа `HeadlessException` не обрабатывается в примерах программ, представленных в этой главе, поскольку для демонстрации элементов управления из библиотеки AWT требуется интерактивная среда.

Метки

Самым простым элементом управления является метка. *Метка* представлена объектом класса `Label` и содержит символьную строку, которая отображается как метка. Метки являются пассивными элементами управления, не поддерживающими взаимодействие с пользователем. В классе `Label` определяются следующие конструкторы:

```
Label() throws HeadlessException  
Label(String строка) throws HeadlessException  
Label(String строка, int способ) throws HeadlessException
```

В первой форме конструктора создается пустая метка, а во второй форме — метка, содержащая заданную *строку*, которая выравнивается по левому краю. В третьей форме конструктора создается метка, содержащая заданную *строку*, выравнивание которой определяется параметром *способ*. Этот параметр должен принимать одну из трех констант: `Label.LEFT`, `Label.RIGHT` или `Label.CENTER`.

Изменить состояние текста в метке можно с помощью метода `setText()`, а получить текущую метку — с помощью метода `getText()`. Ниже приведены общие формы этих методов.

```
void setText(String строка)  
String getText()
```

В методе `setText()` параметр *строка* обозначает новую метку. А метод `getText()` возвращает текущую метку.

Выровнять строку в метке можно с помощью метода `setAlignment()`. Чтобы получить текущий стиль выравнивания, следует вызвать метод `getAlignment()`. Ниже приведены общие формы этих методов.

```
void setAlignment(int способ)
int getAlignment()
```

Параметр *способ* должен принимать одну из трех перечисленных выше констант. В следующем примере создаются три метки, которые затем вводятся в окне апплета:

```
// Продемонстрировать применение меток
import java.awt.*;
import java.applet.*;
/*
  <applet code="LabelDemo" width=300 height=200>
  </applet>
*/

public class LabelDemo extends Applet {
    public void init() {
        Label one = new Label("One");
        Label two = new Label("Two");
        Label three = new Label("Three");

        // ввести метки в окне апплета
        add(one);
        add(two);
        add(three);
    }
}
```

На рис. 26.1 приведен результат вывода меток в окне апплета `LabelDemo`. Обратите внимание на то, что размещение меток в окне выполнено с помощью диспетчера компоновки, выбираемого по умолчанию. Далее в этой главе будет показано, как точнее управлять размещением меток в окне.

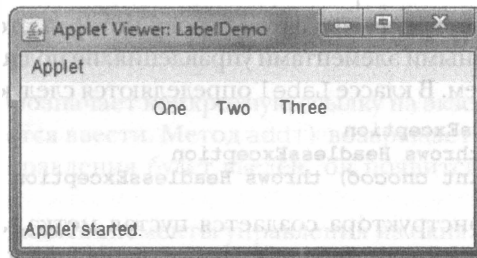


Рис. 26.1. Вывод меток в окне апплета `LabelDemo`

Экранные кнопки

Вероятно, наиболее широко употребляемым элементом управления является *экранная кнопка* — компонент, который содержит метку и извещает о событии, когда

пользователь щелкает на нем кнопкой мыши. Экранные кнопки представлены объектами класса `Button`. В классе `Button` определяются следующие конструкторы:

```
Button() throws HeadlessException  
Button(String строка) throws HeadlessException
```

В первой форме конструктора создается пустая кнопка, а во второй форме — кнопка с заданной меткой *строка*. Как только кнопка будет создана, с помощью метода `setLabel()` можно задать ее метку. А получить метку можно с помощью метода `getLabel()`. Ниже приведены общие формы этих методов, где параметр строка обозначает новую метку для кнопки.

```
void setLabel(String строка)  
String getLabel()
```

Обработка событий от кнопок

Когда пользователь щелкает на экранной кнопке, наступает событие действия. Извещение о нем посылается любому приемнику событий, который был предварительно зарегистрирован на получение извещений о событиях действия от данного компонента. Каждый приемник событий реализует интерфейс `ActionListener`. В этом интерфейсе определяется метод `actionPerformed()`, который вызывается при наступлении события действия. В качестве параметра этому методу передается объект класса `ActionEvent`. Он содержит ссылку на кнопку, сгенерировавшую событие, а также ссылку на *строку с командой действия*, связанную с этой кнопкой. По умолчанию строкой с командой действия является метка кнопки. Как правило, для обозначения кнопки служит ссылка на кнопку или строка с командой действия. (Оба способа обозначения экранных кнопок демонстрируются в приведенных далее примерах.)

В приведенном ниже примере апплета демонстрируется создание трех кнопок с метками "Yes" (Да), "No" (Нет) и "Undecided" (Неопределенно). Когда пользователь щелкает на одной из этих кнопок, выводится сообщение, извещающее о выборе кнопки. В данном примере команда действия кнопки (которая по умолчанию является ее меткой) служит для определения нажатой кнопки. Чтобы получить метку, вызывается метод `getActionCommand()` для объекта типа `ActionEvent`, который передается методу `actionPerformed()`. Пример выполнения апплета `ButtonDemo` приведен на рис. 26.2.

```
// Продемонстрировать применение кнопок  
import java.awt.*;  
import java.awt.event.*;  
import java.applet.*;  
/*  
    <applet code="ButtonDemo" width=250 height=150>  
    </applet>  
*/
```

```
public class ButtonDemo extends Applet implements ActionListener {  
    String msg = "";  
    Button yes, no, maybe;  
  
    public void init() {  
        yes = new Button("Yes");  
        no = new Button("No");
```

```

        maybe = new Button("Undecided");

        add(yes);
        add(no);
        add(maybe);

        yes.addActionListener(this);
        no.addActionListener(this);
        maybe.addActionListener(this);
    }

    public void actionPerformed(ActionEvent ae) {
        String str = ae.getActionCommand();
        if(str.equals("Yes")) {
            msg = "You pressed Yes.";
            // Нажата кнопка Yes
        }
        else if(str.equals("No")) {
            msg = "You pressed No.";
            // Нажата кнопка No
        }
        else {
            msg = "You pressed Undecided.";
            // Нажата кнопка Undecided
        }
        repaint();
    }

    public void paint(Graphics g) {
        g.drawString(msg, 6, 100);
    }
}

```

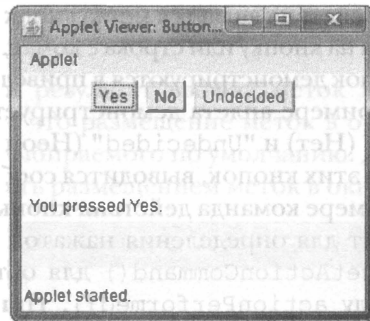


Рис. 26.2. Выбор одной из трех кнопок
в окне апплета ButtonDemo

Как упоминалось выше, помимо строк с командами действия кнопок, выяснить, какая именно кнопка была нажата, можно, сравнив объект, полученный из метода `getSource()`, с объектами кнопок, введенных в окне. Для этого придется вести список вводимых объектов. Такой способ демонстрируется в следующем примере апплета:

```

// Распознать объекты типа Button
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
    <applet code="ButtonList" width=250 height=150>
    </applet>
*/

```

```
public class ButtonList extends Applet implements ActionListener {
    String msg = "";
    Button bList[] = new Button[3];

    public void init() {
        Button yes = new Button("Yes");
        Button no = new Button("No");
        Button maybe = new Button("Undecided")

        // сохранить ссылки на кнопки при их вводе
        bList[0] = (Button) add(yes);
        bList[1] = (Button) add(no);
        bList[2] = (Button) add(maybe);

        // зарегистрировать приемники на получение уведомлений
        // о событиях действия
        for(int i = 0; i < 3; i++) {
            bList[i].addActionListener(this);
        }
    }

    public void actionPerformed(ActionEvent ae) {
        for(int i = 0; i < 3; i++) {
            if(ae.getSource() == bList[i]) {
                msg = "You pressed " + bList[i].getLabel();
                // Нажата указанная кнопка
            }
        }
        repaint();
    }

    public void paint(Graphics g) {
        g.drawString(msg, 6, 100);
    }
}
```

В данном примере при вводе кнопок в окне апплета ссылка на каждую кнопку записывается в массив. (Напомним, что метод `add()` возвращает ссылку на кнопку при ее вводе.) В дальнейшем этот массив используется в методе `actionPerformed()`, чтобы выяснить, какая именно кнопка была нажата.

В простых прикладных программах распознать кнопки по их меткам, как правило, нетрудно. Но если метку кнопки предполагается изменять во время выполнения программы или использовать кнопки с одинаковыми метками, то выяснить, какая именно кнопка была нажата, можно очень просто, если воспользоваться ссылкой на ее объект. А если вызвать метод `setActionCommand()`, то в строке с командой действия, связанной с кнопкой, можно задать содержимое, отличающееся от метки кнопки. Этот метод изменяет строку с командой действия, не оказывая никакого влияния на строку, используемую в качестве метки кнопки. Таким образом, задав строку с командой действия, можно отделить команду действия от метки кнопки.

В некоторых случаях события действия, генерируемые кнопками или другими элементами управления, можно обрабатывать с помощью анонимного внутреннего класса (см. главу 24) или лямбда-выражения (см. главу 15). В качестве примера ниже показано, как установить обработчики событий действия в предыдущих примерах апплетов, используя лямбда-выражения.


```
// Применить лямбда-выражения для обработки событий действия
yes.addActionListener((ae) -> {
    msg = "You pressed " + ae.getActionCommand(); // Нажата кнопка ...
    repaint();
});

no.addActionListener((ae) -> {
    msg = "You pressed " + ae.getActionCommand();
    repaint();
});

maybe.addActionListener((ae) -> {
    msg = "You pressed " + ae.getActionCommand();
    repaint();
});
```

Этот код вполне работоспособен потому, что в интерфейсе `ActionListener` определяется единственный абстрактный метод, а следовательно, он является функциональным интерфейсом и может применяться в лямбда-выражениях. В общем, лямбда-выражение можно применять для обработки события от элемента управления из библиотеки AWT, если в его приемнике определяется функциональный интерфейс. Например, интерфейс `ItemListener` также является функциональным. Безусловно, традиционный способ обработки событий, применение для этой цели анонимного внутреннего класса или лямбда-выражения зависит от характера прикладной программы. В остальных примерах, представленных далее в этой главе, применяется традиционный способ обработки событий, чтобы исходный код этих примеров можно было скомпилировать практически в любой версии Java. Но ради интереса можно попытаться переделать обычные обработчики событий на лямбда-выражения или анонимные внутренние классы там, где это уместно.

Флажки

Флажок представляет собой элемент управления, предназначенный для включения или отключения какого-нибудь режима. Он состоит из небольшой прямоугольной ячейки, которая может содержать отметку в виде галочки, если флажок установлен, или быть пустой. Щелчком на флажке можно изменить его текущее состояние, т.е. установить или сбросить. Флажки могут использоваться как по отдельности, так и в группе. Они представлены объектами класса `Checkbox`.

В классе `Checkbox` поддерживаются следующие конструкторы:

```
Checkbox() throws HeadlessException
Checkbox(String строка) throws HeadlessException
Checkbox(String строка, boolean включено) throws HeadlessException
Checkbox(String строка, boolean включено, CheckboxGroup группа_флажков)
    throws HeadlessException
Checkbox(String строка, CheckboxGroup группа_флажков, boolean включено)
    throws HeadlessException
```

В первой форме конструктора создается флажок с пустой изначально меткой. Исходно флажок находится в сброшенном состоянии. Во второй форме конструктора создается флажок, метка которого определяется параметром *строка*. Исходно флажок находится в сброшенном состоянии. В третьей форме можно

установить исходное состояние флажка. Если параметр *включено* принимает логическое значение `true`, то флажок будет исходно установлен, а иначе — сброшен. В четвертой и пятой формах создается флажок, метка которого определяется параметром *строка*, а группа — параметром *группа_флажков*. Если флажок не является частью группы, то параметр *группа_флажков* должен принимать пустое значение `null`. (Группы флажков описываются в следующем разделе.) Значение параметра *включено* определяет исходное состояние флажка.

Для получения текущего состояния флажка служит метод `getState()`, а для его установки в нужное состояние — метод `setState()`. Вызвав метод `getLabel()`, можно получить текущую метку, связанную с флажком. А для того чтобы задать метку, следует вызвать метод `setLabel()`. Ниже приведены общие формы этих методов.

```
boolean getState()  
void setState(boolean включено)  
String getLabel()  
void setLabel(String строка)
```

Если параметр *включено* принимает логическое значение `true`, то флажок будет установлен. А если этот параметр принимает логическое значение `false`, то флажок будет сброшен. Заданная *строка* становится новой меткой, связанной с вызывающим флажком.

Обработка событий от флажков

Всякий раз, когда флажок устанавливается или сбрасывается, наступает событие от элемента. Извещение о нем передается любому приемнику событий, который ранее зарегистрировался на получение извещений о событиях от элементов данного компонента. Каждый приемник событий реализует интерфейс `ItemListener`. В этом интерфейсе определяется метод `itemStateChanged()`. Объект класса `ItemEvent` задается в качестве параметра данного метода. Он содержит сведения о событии (например, выбор или отмена выбора).

В приведенном ниже примере апплета демонстрируется создание и применение четырех флажков. Исходно установлен первый флажок. Состояние всех флажков отображается в окне апплета. Оно обновляется всякий раз, когда изменяется состояние любого флажка. Пример установки флажков в окне данного апплета приведен на рис. 26.3.

```
// Продемонстрировать применение флажков  
import java.awt.*;  
import java.awt.event.*;  
import java.applet.*;  
/*  
  <applet code="CheckboxDemo" width=240 height=200>  
  </applet>  
*/  
  
public class CheckboxDemo extends Applet implements ItemListener {  
    String msg = "";  
    Checkbox windows, android, solaris, mac;  
  
    public void init() {
```

```

windows = new Checkbox("Windows", null, true);
android = new Checkbox("Android");
solaris = new Checkbox("Solaris");
mac = new Checkbox("Mac OS");

add(windows);
add(android);
add(solaris);
add(mac);

windows.addItemListener(this);
android.addItemListener(this);
solaris.addItemListener(this);
mac.addItemListener(this);
}

public void itemStateChanged(ItemEvent ie) {
    repaint();
}

// отобразить текущее состояние флажков
public void paint(Graphics g) {
    msg = "Current state: "; // Текущее состояние
    g.drawString(msg, 6, 80);
    msg = " Windows: " + windows.getState();
    g.drawString(msg, 6, 100);
    msg = " Android: " + android.getState();
    g.drawString(msg, 6, 120);
    msg = " Solaris: " + solaris.getState();
    g.drawString(msg, 6, 140);
    msg = " Mac OS: " + mac.getState();
    g.drawString(msg, 6, 160);
}
}

```

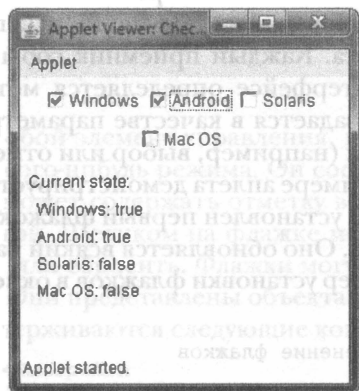


Рис. 26.3. Установка флажков
в окне апплета CheckboxDemo

Кнопки-переключатели

Допускается создание группы взаимоисключающих флажков, где одновременно может быть установлен один (и только один) флажок. Такие флажки нередко

называют *кнопками-переключателями*, потому что они похожи на переключатели каналов в автомобильном радиоприемнике, где одновременно можно выбрать только одну радиостанцию. Чтобы создать ряд кнопок-переключателей, нужно сначала определить группу, к которой они должны принадлежать, а затем указать эту группу при создании кнопок-переключателей. Кнопки-переключатели представлены объектами класса `CheckboxGroup`. В этом классе определен только конструктор по умолчанию, создающий пустую группу.

Чтобы выяснить, какая именно кнопка-переключатель установлена на данный момент, следует вызвать метод `getSelectedCheckBox()`. А установить кнопку-переключатель можно с помощью метода `setSelectedCheckbox()`. Ниже приведены общие формы этих методов.

```
Checkbox getSelectedCheckbox()  
void setSelectedCheckbox(Checkbox кнопка-переключатель)
```

Здесь параметр *кнопка-переключатель* обозначает ту кнопку-переключатель, которую требуется установить. При этом установленная ранее кнопка-переключатель сбрасывается. В следующем примере апплета демонстрируется применение группы кнопок-переключателей:

```
// Продемонстрировать применение кнопок-переключателей
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="CBGroup" width=240 height=200>
  </applet>
*/

public class CBGroup extends Applet implements ItemListener {
    String msg = "";
    Checkbox windows, android, solaris, mac;
    CheckboxGroup cbg;
    public void init() {
        cbg = new CheckboxGroup();
        windows = new Checkbox("Windows", cbg, true);
        android = new Checkbox("Android", cbg, false);
        solaris = new Checkbox("Solaris", cbg, false);
        mac = new Checkbox("Mac OS", cbg, false);

        add(windows);
        add(android);
        add(solaris);
        add(mac);

        windows.addItemListener(this);
        android.addItemListener(this);
        solaris.addItemListener(this);
        mac.addItemListener(this);
    }

    public void itemStateChanged(ItemEvent ie) {
        repaint();
    }

    // отобразить текущее состояние кнопок-переключателей
    public void paint(Graphics g) {
```

```

    msg = "Current selection: ";
    msg += cbg.getSelectedCheckbox().getLabel();
    g.drawString(msg, 6, 100);
}
}

```

Пример установки кнопок-переключателей в окне апплета CbGroup приведен на рис. 26.4. Обратите внимание на круглую форму кнопок-переключателей.



Рис. 26.4. Установка кнопок-переключателей в окне апплета CbGroup

Элементы управления выбором

Класс *Choice* служит для создания *раскрывающегося списка* элементов, из которого пользователь может делать свой выбор. Поэтому элемент управления типа *Choice* является разновидностью меню. Будучи неактивным, компонент типа *Choice* занимает ровно столько свободного пространства, сколько требуется для отображения выбранного на данный момент элемента. Когда пользователь выбирает этот элемент щелчком на нем, раскрывается весь список, в котором можно выбрать новый элемент. Каждый элемент в списке представлен символьной строкой, которая появляется в виде выровненной по левому краю метки в том порядке, в каком данный элемент вводится в объект класса *Choice*. В классе *Choice* определяется только конструктор по умолчанию, который создает пустой список.

Чтобы ввести элемент выбора в список, следует вызвать метод `add()`. Этот метод имеет следующую общую форму:

```
void add(String имя)
```

где параметр *имя* обозначает вводимый элемент. Ввод элементов в список осуществляется в том порядке, в каком делаются вызовы метода `add()`.

Чтобы выяснить, какой именно элемент выбран на данный момент из списка, достаточно вызвать один из двух методов: `getSelectedItem()` или `getSelectedIndex()`. Ниже приведены общие формы этих методов.

```
String getSelectedItem()
int getSelectedIndex()
```

Метод `getSelectedItem()` возвращает символьную строку, содержащую имя элемента, а метод `getSelectedIndex()` — индекс элемента. Первый элемент имеет нулевой индекс. По умолчанию выбирается первый элемент, введенный в список.

Чтобы выяснить, сколько элементов содержится в списке, следует вызвать метод `getItemCount()`. А для того чтобы сделать элемент выбранным в данный момент, достаточно передать методу `select()` начинающийся с нуля целочисленный индекс или символьную строку, совпадающую с именем элемента в списке. Общие формы этих методов приведены ниже.

```
int getItemCount()
void select(int индекс)
void select(String имя)
```

По указанному индексу можно получить имя, связанное с элементом, доступным по этому индексу, вызвав метод `getItem()`, имеющий приведенную ниже форму, где *индекс* обозначает конкретный индекс требуемого элемента.

```
String getItem(int индекс)
```

Обработка событий от раскрывающихся списков

Всякий раз, когда из раскрывающегося списка выбирается элемент, наступает соответствующее событие. Извещение об этом событии передается всем приемникам, которые предварительно зарегистрировались на получение извещений о событиях от элементов данного компонента. Каждый приемник событий реализует интерфейс `ItemListener`. В этом интерфейсе определяется метод `itemStateChanged()`. В качестве параметра этого метода передается объект типа `ItemEvent`.

В приведенном ниже примере апплета демонстрируется создание двух раскрывающихся списков типа `Choice`. В одном из них производится выбор операционной системы, а в другом — браузера. Пример выбора элемента из раскрывающегося списка в окне апплета `ChoiceDemo` приведен на рис. 26.5.

```
// Продемонстрировать применение раскрывающихся списков
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="ChoiceDemo" width=300 height=180>
  </applet>
*/
```

```
public class ChoiceDemo extends Applet implements ItemListener {
    Choice os, browser;
    String msg = "";

    public void init() {
        os = new Choice();
        browser = new Choice();

        // ввести элементы в список операционных систем
        os.add("Windows");
        os.add("Android");
        os.add("Solaris");
        os.add("Mac OS");
```

```

// ввести элементы в список браузеров
browser.add("Internet Explorer");
browser.add("Firefox");
browser.add("Chrome");

// ввести списки выбора в окне
add(os);
add(browser);

// зарегистрировать приемники событий от элементов
os.addItemListener(this);
browser.addItemListener(this);
}

public void itemStateChanged(ItemEvent ie) {
    repaint();
}

// отобразить результаты текущего выбора из списков
public void paint(Graphics g) {
    msg = "Current OS: "; // Текущая ОС
    msg += os.getSelectedItem();
    g.drawString(msg, 6, 120);
    msg = "Current Browser: "; // Текущий браузер
    msg += browser.getSelectedItem();
    g.drawString(msg, 6, 140);
}
}

```

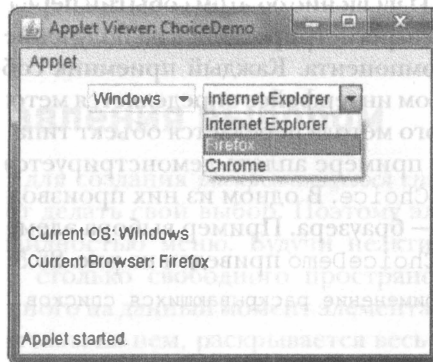


Рис. 26.5. Выбор элемента из раскрывающегося списка в окне апплета ChoiceDemo

Использование списков

В классе `List` предоставляется компактный прокручиваемый список, из которого можно выбирать многие элементы. В отличие от объекта класса `Choice`, который отображает только один элемент, выбранный из раскрывающегося списка, объект класса `List` можно создать таким образом, чтобы он показывал любое количество элементов выбора в видимом окне. Подобный список можно создать и таким образом, чтобы из него можно было выбрать несколько элементов. В классе `List` предоставляются следующие конструкторы:

```
List() throws HeadlessException  
List(int количество_строк) throws HeadlessException  
List(int количество_строк, boolean выбор_нескольких_элементов)  
    throws HeadlessException
```

В первой форме конструктора создается элемент управления типа `List`, который позволяет одновременно выбрать только один элемент. Во второй форме конструктора параметр *количество_строк* обозначает количество элементов, которые будут постоянно видимы в списке, тогда как остальные элементы можно просмотреть, прокрутив список. Третья форма позволяет выбрать одновременно два или несколько элементов из списка, если параметр *выбор_нескольких_элементов* принимает логическое значение `true`. Если же этот параметр принимает логическое значение `false`, то из списка можно будет выбрать только один элемент.

Для ввода элемента в список следует вызвать метод `add()`, у которого имеются две формы:

```
void add(String имя)  
void add(String имя, int индекс)
```

где параметр *имя* обозначает имя элемента, вводимого в список. В первой форме элементы вводятся в конце списка, а во второй элемент вводится в список по указанному *индексу*. Индексация элементов в списке начинается с нуля. Чтобы ввести элемент в конце списка, следует указать значение `-1` параметра *индекс*.

Чтобы выяснить, какой элемент выбран в данный момент из тех списков, где допускается выбирать только один элемент, следует вызвать метод `getSelectedItem()` или `getSelectedIndex()`. Ниже приведены общие формы этих методов.

```
String getItem()   
int getSelectedIndex()
```

Метод `getSelectedItem()` возвращает символьную строку, содержащую имя элемента. Если выбрано несколько элементов или пока еще не выбран ни один из них, то возвращается пустое значение `null`. Метод `getSelectedIndex()` возвращает индекс выбранного элемента. Первый элемент имеет нулевой индекс. Если же выбрано несколько элементов или пока еще не выбран ни один из них, то возвращается пустое значение `-1`.

Чтобы выяснить, какие элементы выбраны в данный момент из тех списков, где допускается одновременно выбирать несколько элементов, следует вызвать метод `getSelectedItems()` или `getSelectedIndexes()`. В частности, метод `getSelectedItems()` возвращает массив, содержащий имена выбранных на данный момент элементов, а метод `getSelectedIndexes()` — массив, содержащий индексы выбранных в данный момент элементов. Ниже приведены общие формы этих методов.

```
String[] getSelectedItems()  
int[] getSelectedIndexes()
```

Чтобы выяснить, сколько элементов содержится в списке, следует вызвать метод `getItemCount()`. А с помощью метода `select()` можно определить, какой именно элемент выбран в данный момент. Для этих целей служит целочисленный индекс, начинающийся с нуля. Ниже приведены общие формы этих методов.

```
int getItemCount()  
void select(int индекс)
```


По указанному *индексу* можно получить имя, связанное с элементом, доступным по этому индексу, если вызвать метод `getItem()`. Этот метод имеет приведенную ниже общую форму, где *индекс* обозначает конкретный индекс требуемого элемента.

```
String getItem(int индекс)
```

Обработка событий от списков

Для обработки событий от списков следует реализовать интерфейс `ActionListener`. Всякий раз, когда производится двойной щелчок на элементе списка типа `List`, создается объект типа `ActionEvent`. Его метод `getActionCommand()` можно вызвать для извлечения имени вновь выбранного элемента. Кроме того, при выборе или отмене выбора элемента одним щелчком на нем создается объект типа `ItemEvent`. Вызвав его метод `getStateChanged()`, можно выяснить, чем было вызвано данное событие: выбором элемента или отменой его выбора. А метод `getItemSelectable()` возвращает ссылку на объект, инициировавший данное событие.

В приведенном ниже примере апплета раскрывающиеся списки типа `Choice` из предыдущего примера заменены списками типа `List`: одним — для множественного выбора, а другим — для однократного выбора. Пример выбора нескольких элементов из раскрывающихся списков в окне апплета `ListDemo` приведен на рис. 26.6.

```
// Продемонстрировать применение списков
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="ListDemo" width=300 height=180>
  </applet>
*/

public class ListDemo extends Applet implements ActionListener {
    List os, browser;
    String msg = "";

    public void init() {
        os = new List(4, true);
        browser = new List(4, false);

        // ввести элементы в список операционных систем
        os.add("Windows");
        os.add("Android");
        os.add("Solaris");
        os.add("Mac OS");

        // ввести элементы в список браузеров
        browser.add("Internet Explorer");
        browser.add("Firefox");
        browser.add("Chrome");

        browser.select(1);

        // ввести списки в окно
        add(os);
        add(browser);

        // зарегистрировать приемники событий действия
        os.addActionListener(this);
        browser.addActionListener(this);
    }
}
```

```
}

public void actionPerformed(ActionEvent ae) {
    repaint();
}

// отобразить текущие результаты выбора из списков
public void paint(Graphics g) {
    int idx[];

    msg = "Current OS: "; // Текущая ОС
    idx = os.getSelectedIndexes();
    for(int i=0; i<idx.length; i++)
        msg += os.getItem(idx[i]) + " ";
    g.drawString(msg, 6, 120);
    msg = "Current Browser: "; // Текущий браузер
    msg += browser.getSelectedItems();
    g.drawString(msg, 6, 140);
}
}
```

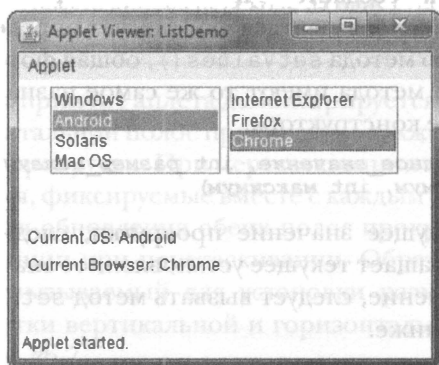


Рис. 26.6. Выбор нескольких элементов из раскрывающихся списков в окне апплета ListDemo

Управление полосами прокрутки

Полосы прокрутки служат для выбора непрерывных значений в заданных пределах от минимума и максимума. Полосы прокрутки могут быть ориентированы по вертикали и по горизонтали. В действительности полоса прокрутки состоит из нескольких отдельных частей. На обоих концах полосы находится кнопка со стрелкой, щелчок на которой вызывает перемещение текущего значения полосы прокрутки на одну единицу в направлении, указываемом стрелкой. Текущее прокручиваемое значение по отношению к заданным минимальному и максимальному пределам обозначается с помощью *ползунка* на полосе прокрутки. Пользователь может перетащить ползунок на другую позицию, после чего новое значение будет отражено на полосе прокрутки. Для прокрутки с шагом больше 1 пользователь может щелкнуть кнопкой мыши на самой полосе прокрутки по обе стороны от ползунка. Как правило, такое действие сводится к некоторой форме перелистывания страниц вверх и вниз. Полосы прокрутки инкапсулируются в классе `Scrollbar`.

В классе `Scrollbar` определяются следующие конструкторы:

```
Scrollbar()
Scrollbar(int стиль)
Scrollbar(int стиль, int исходное_значение,
          int размер_ползунка, int минимум, int максимум)
```

В первой форме конструктора создается вертикальная полоса прокрутки, а во второй и в третьей формах можно задавать ориентацию полосы прокрутки. Если параметр *стиль* принимает значение константы `Scrollbar.VERTICAL`, создается вертикальная полоса прокрутки, а если он принимает значение константы `Scrollbar.HORIZONTAL` – то горизонтальная полоса прокрутки. В третьей форме конструктора параметр *исходное_значение* обозначает конкретное значение, определяющее исходное положение полосы прокрутки; параметр *размер_ползунка* – высоту ползунка в текущих единицах; параметры *минимум* и *максимум* – соответствующие предельные значения прокрутки.

Если полоса прокрутки создается с помощью одного из первых двух конструкторов, то, прежде чем воспользоваться ею, следует определить параметры полосы прокрутки с помощью метода `setValues()`, общая форма которого приведена ниже. Параметры этого метода имеют то же самое назначение, что и в третьей форме описанного выше конструктора.

```
void setValues(int исходное_значение, int размер_ползунка,
              int минимум, int максимум)
```

Чтобы выяснить текущее значение прокрутки, следует вызвать метод `getValue()`, который возвращает текущее установленное значение. А для того чтобы установить текущее значение, следует вызвать метод `setValue()`. Общие формы этих методов показаны ниже.

```
int getValue()
void setValue(int новое_значение)
```

Здесь параметр *новое_значение* обозначает устанавливаемое новое значение прокрутки. После установки этого значения ползунок расположится на полосе прокрутки в соответствии с новым значением.

Вызывая методы `getMinimum()` и `getMaximum()`, можно получить минимальное и максимальное значения прокрутки соответственно. Ниже приведены общие формы этих методов.

```
int getMinimum()
int getMaximum()
```

По умолчанию прокрутка вверх или вниз производится с единичным шагом. При этом значение единичного шага добавляется к значению прокрутки или соответственно вычитается из него. Изменить шаг прокрутки можно, вызвав метод `setUnitIncrement()`. А постраничная прокрутка по умолчанию выполняется с шагом 10. Величину этого шага можно изменить, вызвав метод `setBlockIncrement()`. Ниже приведены общие формы этих методов.

```
void setUnitIncrement(int новый_шаг)
void setBlockIncrement(int новый_шаг)
```

Обработка событий от полос прокрутки

Для обработки событий от полос прокрутки следует реализовать интерфейс `AdjustmentListener`. Всякий раз, когда пользователь начинает прокрутку, формируется объект класса `AdjustmentEvent`. Его метод `getAdjustmentType()` служит для определения типа события настройки. В табл. 26.1 перечислены типы событий настройки.

Таблица 26.1. События настройки

Событие	Описание
<code>BLOCK_DECREMENT</code>	Листание страницы вниз
<code>BLOCK_INCREMENT</code>	Листание страницы вверх
<code>TRACK</code>	Абсолютное отслеживание
<code>UNIT_DECREMENT</code>	Нажата кнопка перехода на одну строку вниз
<code>UNIT_INCREMENT</code>	Нажата кнопка перехода на одну строку вверх

В приведенном ниже примере апплета демонстрируется создание и применение вертикальной и горизонтальной полос прокрутки, а также отображаются текущие установки обеих полос прокрутки. При перетаскивании курсора мыши в окне координаты его положения, фиксируемые вместе с каждым событием перемещения, будут использоваться для обновления обеих полос прокрутки. Звездочка отображается на текущей позиции при перетаскивании. Обратите внимание на метод `setPreferredSize()`, вызываемый для установки размеров обеих полос прокрутки. Пример прокрутки вертикальной и горизонтальной полос в окне апплета `SBDemo` приведен на рис. 26.7.

```
// Продемонстрировать применение полос прокрутки
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="SBDemo" width=300 height=200>
  </applet>
*/

public class SBDemo extends Applet
implements AdjustmentListener, MouseMotionListener {
    String msg = "";
    Scrollbar vertSB, horzSB;

    public void init() {
        int width = Integer.parseInt(getParameter("width"));
        int height = Integer.parseInt(getParameter("height"));
        vertSB = new Scrollbar(Scrollbar.VERTICAL, 0, 1, 0, height);
        vertSB.setPreferredSize(new Dimension(20, 100));

        horzSB = new Scrollbar(Scrollbar.HORIZONTAL, 0, 1, 0, width);
        horzSB.setPreferredSize(new Dimension(100, 20));

        add(vertSB);
        add(horzSB);
    }
}
```

```

// зарегистрировать приемники событий настройки
vertSB.addAdjustmentListener(this);
horzSB.addAdjustmentListener(this);

addMouseMotionListener(this);
}

public void adjustmentValueChanged(AdjustmentEvent ae) {
    repaint();
}

// обновить полосу прокрутки в ответ на перетаскивание
// курсора мыши
public void mouseDragged(MouseEvent me) {
    int x = me.getX();
    int y = me.getY();
    vertSB.setValue(y);
    horzSB.setValue(x);
    repaint();
}

// Это требуется для интерфейса MouseMotionListener
public void mouseMoved(MouseEvent me) {
}

// отобразить текущие значения прокрутки
public void paint(Graphics g) {
    msg = "Vertical: " + vertSB.getValue();
    // Прокрутка по вертикали
    msg += ", Horizontal: " + horzSB.getValue();
    // Прокрутка по горизонтали
    g.drawString(msg, 6, 160);

    // отобразить текущее положение перетаскиваемого
    // курсора мыши
    g.drawString("**", horzSB.getValue(), vertSB.getValue());
}
}

```

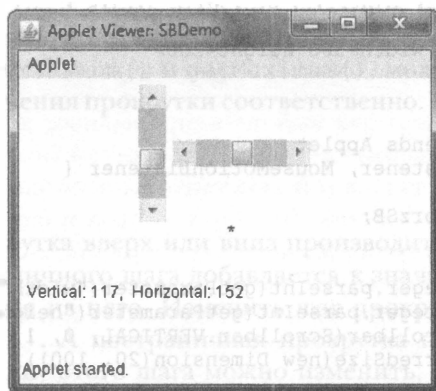


Рис. 26.7. Прокрутка вертикальной и горизонтальной
полос в окне апплета SBDemo

Текстовые поля

Класс `TextField` реализует однострочную область для ввода текста, которая называется *текстовым полем*. В текстовых полях пользователь может вводить текст построчно и редактировать его. Класс `TextField` является производным от класса `TextComponent` и имеет следующие конструкторы:

```
TextField() throws HeadlessException
TextField(int количество_символов) throws HeadlessException
TextField(String строка) throws HeadlessException
TextField(String строка, int количество_символов)
    throws HeadlessException
```

В первой форме конструктора создается текстовое поле, а во второй форме — текстовое поле шириной, равной заданному *количеству_символов*. В третьей форме конструктора создаваемое текстовое поле инициализируется указанной *строкой*, а в четвертой форме — текстовое поле не только инициализируется указанной *строкой*, но и получает ширину, равную заданному *количеству_символов*.

В классе `TextField` и его суперклассе `TextComponent` предоставляется ряд методов для обращения с текстовыми полями. Чтобы получить строку из текстового поля, следует вызвать метод `getText()`, а для того чтобы задать в нем текст — метод `setText()`. Ниже приведены общие формы этих методов, где *строка* обозначает новую строку текста.

```
String getText()
void setText(String строка)
```

Пользователь может выделить часть текста в текстовом поле, но это можно сделать и программно с помощью метода `select()`. Вызвав в программе метод `getSelectedText()`, можно получить текст, выделенный в данный момент. Ниже приведены общие формы этих методов.

```
String getSelectedText()
void select(int начальный_индекс, int конечный_индекс)
```

Метод `getSelectedText()` возвращает выделенный текст. А метод `select()` выделяет символы от позиции *начальный_индекс* и до позиции *конечный_индекс-1*.

Вызвав метод `setEditable()`, пользователю можно разрешить изменять содержимое текстового поля, а вызвав метод `isEditable()` — редактировать текст. Ниже приведены общие формы этих методов.

```
boolean isEditable()
void setEditable(boolean правка)
```

Метод `isEditable()` возвращает логическое значение `true`, если текст можно изменять, а если текст изменять нельзя — то логическое значение `false`. Если параметр *правка* метода `setEditable()` принимает логическое значение `true`, текст можно изменять. А если этот параметр принимает логическое значение `false`, то текст изменять нельзя.

Иногда нужно сделать невидим текст, вводимый пользователем (например, при вводе пароля). С помощью метода `setEchoChar()` можно запретить эхоот-

бражение вводимых символов. Этот метод задает одиночный символ, который объект класса `TextField` будет отображать вместо вводимых символов. Вызвав метод `getEchoChar()`, можно выяснить, находится ли текстовое поле в режиме эхоотображения вводимых символов. А для того чтобы получить эхо-символ отображения, следует вызвать метод `getEchoChar()`. Ниже приведены общие формы этих методов.

```
void setEchoChar(char символ)
boolean echoCharIsSet()
char getEchoChar()
```

Здесь параметр *символ* обозначает эхо-символ. Если указать нулевое значение параметра *символ*, то восстанавливается нормальное отображение вводимых символов на экране.

Обработка событий в текстовых полях

Текстовые поля выполняют собственные функции редактирования, и поэтому прикладная программа, как правило, вообще не будет реагировать на события ввода отдельных символов в текстовом поле. Но в то же время может возникнуть потребность реагировать на нажатие пользователем клавиши <Enter>. При нажатии этой клавиши наступает событие действия.

В следующем примере апплета создается классическая экранная форма для ввода имени пользователя и пароля:

```
// Продемонстрировать применение текстового поля
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="TextFieldDemo" width=380 height=150>
  </applet>
*/

public class TextFieldDemo extends Applet
implements ActionListener {

    TextField name, pass;

    public void init() {
        Label namep = new Label("Name: ", Label.RIGHT);
        // Имя пользователя
        Label passp = new Label("Password: ", Label.RIGHT);
        // Пароль пользователя
        name = new TextField(12);
        pass = new TextField(8);
        pass.setEchoChar('?');
        add(namep);
        add(name);
        add(passp);
        add(pass);

        // зарегистрировать приемники событий действия
        name.addActionListener(this);
        pass.addActionListener(this);
    }
}
```

```

// Пользователь нажал клавишу <Enter>
public void actionPerformed(ActionEvent ae) {
    repaint();
}

public void paint(Graphics g) {
    g.drawString("Name: " + name.getText(), 6, 60);
    // Имя пользователя
    g.drawString("Selected text in name: " +
        name.getSelectedText(), 6, 80);
    // Выделенный текст в поле имени пользователя
    g.drawString("Password: " + pass.getText(), 6, 100);
    // Пароль пользователя
}
}

```

Пример ввода имени пользователя и пароля в окне апплета `TextFieldDemo` приведен на рис. 26.8.

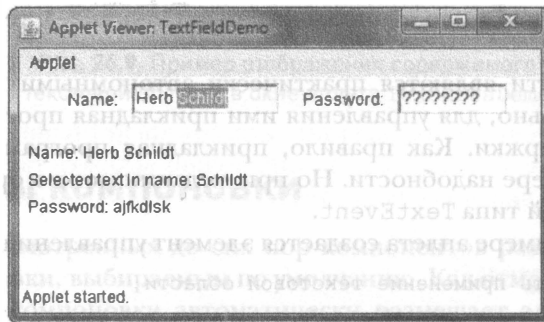


Рис. 26.8. Пример ввода имени пользователя и пароля в окне апплета `TextFieldDemo`

Текстовые области

Иногда одной строки для ввода текста оказывается недостаточно. На этот случай в библиотеке AWT предоставляется класс `TextArea`, реализующий простой редактор многострочного текста. Ниже приведены конструкторы этого класса.

```

TextArea() throws HeadlessException
TextArea(int количество_строк, int количество_строк)
    throws HeadlessException
TextArea(String строка) throws HeadlessException
TextArea(String строка, int количество_строк, int количество_символов)
    throws HeadlessException
TextArea(String строка, int количество_строк, int количество_символов,
    int полосы_прокрутки) throws HeadlessException

```

Здесь параметр *количество_строк* обозначает высоту текстовой области в строках, а параметр *количество_символов* — ее ширину в символах. Первоначальный текст можно задать в качестве параметра *строка*. В пятой форме конструктора можно также задать *полосы_прокрутки*, которые могут понадобиться для обращения с текстовой областью. Параметр *полосы_прокрутки* должен принимать одно из значений следующих констант:

<code>SCROLLBARS_BOTH</code>	<code>SCROLLBARS_NONE</code>
<code>SCROLLBARS_HORIZONTAL_ONLY</code>	<code>SCROLLBARS_VERTICAL_ONLY</code>

Класс `TextArea` является производным от класса `TextComponent`. Следовательно, в нем поддерживаются методы `getText()`, `setText()`, `getSelectedText()`, `select()`, `isEditable()` и `setEditable()`, описанные в предыдущем разделе.

Класс `TextArea` предоставляет следующие дополнительные методы:

```
void append(String строка)
void insert(String строка, int индекс)
void replaceRange(String строка, int начальный_индекс,
                  int конечный_индекс)
```

Метод `append()` вводит заданную строку в конце текущего текста. Метод `insert()` вставляет заданную строку по указанному индексу. Чтобы заменить текст, следует вызвать метод `replaceRange()`. Он заменяет символы от позиции `начальный_индекс` и до позиции `конечный_индекс-1` текстом из заданной строки.

Текстовые области являются практически автономными элементами управления, а следовательно, для управления ими прикладная программа не несет дополнительные издержки. Как правило, прикладная программа лишь получает текущий текст по мере надобности. Но при желании можно организовать прием и обработку событий типа `TextEvent`.

В следующем примере апплета создается элемент управления типа `TextArea`:

```
// Продемонстрировать применение текстовой области
import java.awt.*;
import java.applet.*;

/*
<applet code="TextAreaDemo" width=300 height=250>
</applet>
*/

public class TextAreaDemo extends Applet {

    public void init() {
        String val =
            "Java 8 is the latest version of the most\n" +
            "widely-used computer language for Internet programming.\n" +
            "Building on a rich heritage, Java has advanced both\n" +
            "the art and science of computer language design.\n\n" +
            "One of the reasons for Java's ongoing success is its\n" +
            "constant, steady rate of evolution. Java has never stood\n" +
            "still. Instead, Java has consistently adapted to the\n" +
            "rapidly changing landscape of the networked world.\n" +
            "Moreover, Java has often led the way, charting the\n" +
            "course for others to follow.";

        TextArea text = new TextArea(val, 10, 30);
        dd(text);
    }
}
```

Пример отображения содержимого текстовой области в окне апплета `TextAreaDemo` приведен на рис. 26.9.

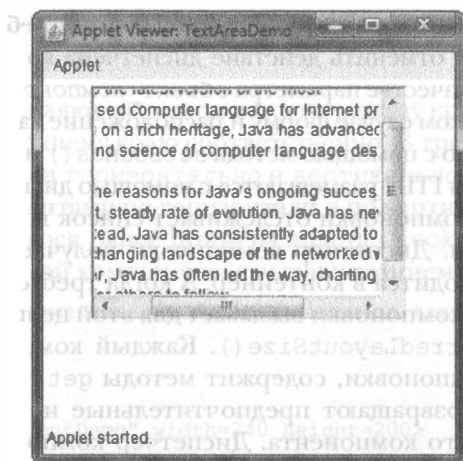


Рис. 26.9. Пример отображения содержимого текстовой области в окне апплета TextAreaDemo

Диспетчеры компоновки

Каждый из рассмотренных до сих пор компонентов размещался в окне диспетчером компоновки, выбираемым по умолчанию. Как отмечалось в начале этой главы, диспетчер компоновки автоматически размещает элементы управления в окне по определенному алгоритму. Если у вас имеется опыт разработки прикладных программ для других сред с ГПИ (например, для Windows), то вы, возможно, привыкли размещать элементы управления ГПИ вручную. Элементы управления, создаваемые средствами Java, можно также размещать вручную, но делать это не стоит по следующим причинам. Во-первых, размещать вручную большое количество компонентов — довольно утомительное занятие. И во-вторых, в тот момент, когда требуется разместить какой-нибудь элемент управления, его ширина и высота могут быть неизвестны, поскольку еще не готовы компоненты платформенно-ориентированных инструментальных средств. Это напоминает ситуацию, где трудно определить причину и следствие, поскольку неясно, когда можно использовать размеры одного компонента для его расположения относительно другого компонента.

У каждого объекта класса `Container` имеется свой диспетчер компоновки, который является экземпляром любого класса, реализующего интерфейс `LayoutManager`. Диспетчер компоновки устанавливается с помощью метода `setLayout()`. Если же метод `setLayout()` не вызывается, то используется диспетчер компоновки, выбираемый по умолчанию. Всякий раз, когда изменяются размеры контейнера, в том числе и в первый раз, диспетчер компоновки применяется для размещения каждого компонента в контейнере.

Метод `setLayout()` имеет следующую общую форму:

```
void setLayout(LayoutManager объект_компоновки)
```

где параметр *объект_компоновки* обозначает ссылку на требуемый диспетчер компоновки. Если требуется отменить действие диспетчера компоновки и разместить компоненты вручную, в качестве параметра *объект_компоновки* следует передать пустое значение `null`. В таком случае форму и расположение каждого компонента придется определить вручную с помощью метода `setBounds()` из класса `Component`. Но, как правило, компоненты ГПИ размещаются с помощью диспетчера компоновки.

Каждый диспетчер компоновки отслеживает список компонентов, хранящихся под своими именами. Диспетчер компоновки получает уведомление всякий раз, когда компонент вводится в контейнер. А когда требуется изменить размеры контейнера, диспетчер компоновки вызывает для этой цели свои методы `minimumLayoutSize()` и `preferredLayoutSize()`. Каждый компонент, которым манипулирует диспетчер компоновки, содержит методы `getPreferredSize()` и `getMinimumSize()`. Они возвращают предпочтительные и минимальные размеры для отображения каждого компонента. Диспетчер компоновки будет учитывать эти размеры, если это вообще возможно, не нарушая правила компоновки. Эти методы можно переопределить в создаваемых подклассах элементов управления, а иначе предоставляются значения по умолчанию.

В Java имеется ряд предопределенных классов диспетчеров компоновки, часть из которых описывается далее. Среди них можно выбрать такой диспетчер компоновки, который лучше всего подходит для конкретной прикладной программы.

Класс `FlowLayout`

Класс `FlowLayout` реализует диспетчер поточной компоновки, выбираемый по умолчанию. Именно этот диспетчер компоновки применялся в предыдущих примерах. Диспетчер компоновки типа `FlowLayout` реализует простой стиль компоновки, который напоминает порядок следования слов в редакторе текста. Направление компоновки определяется свойством ориентации компонента в контейнере, которое по умолчанию задает направление слева направо и сверху вниз. Следовательно, по умолчанию компоненты размещаются построчно, начиная с левого верхнего угла. Но в любом случае компонент переносится на следующую строку, если он не умещается в текущей строке. Между компонентами остаются небольшие промежутки сверху, снизу, справа и слева. Ниже приведены конструкторы класса `FlowLayout`.

`FlowLayout()`

`FlowLayout(int способ)`

`FlowLayout(int способ, int горизонтально, int вертикально)`

В первой форме конструктора выполняется компоновка по умолчанию, т.е. компоненты размещаются по центру, а между ними остается промежуток пять пикселей. Во второй форме конструктора можно определить *способ* расположения каждой строки. Ниже представлены допустимые значения параметра *способ*.

- `FlowLayout.LEFT`
- `FlowLayout.CENTER`
- `FlowLayout.RIGHT`

- `FlowLayout.LEADING`
- `FlowLayout.TRAILING`

Эти значения определяют выравнивание по левому краю, по центру, по правому краю, переднему и заднему краю соответственно. В третьей форме конструктора в качестве параметров *горизонтально* и *вертикально* можно определить промежутки между компонентами по горизонтали и по вертикали. Ниже приведен вариант рассматривавшегося ранее примера апплета `CheckboxDemo`, переделанный для выполнения поточной компоновки с выравниванием по левому краю.

```
// Применить поточную компоновку с выравниванием по левому краю
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="FlowLayoutDemo" width=240 height=200>
  </applet>
*/

public class FlowLayoutDemo extends Applet
    implements ItemListener {

    String msg = "";
    Checkbox windows, android, solaris, mac;

    public void init() {
        // установить поточную компоновку с выравниванием по левому краю
        setLayout(new FlowLayout(FlowLayout.LEFT));

        windows = new Checkbox("Windows", null, true);
        android = new Checkbox("Android");
        solaris = new Checkbox("Solaris");
        mac = new Checkbox("Mac OS");

        add(windows);
        add(android);
        add(solaris);
        add(mac);

        // зарегистрировать получателя событий от элементов
        windows.addItemListener(this);
        android.addItemListener(this);
        solaris.addItemListener(this);
        mac.addItemListener(this);
    }

    // перерисовать, когда состояние флажка изменится
    public void itemStateChanged(ItemEvent ie) {
        repaint();
    }

    // отобразить текущее состояние флажков
    public void paint(Graphics g) {

        msg = "Current state: ";
        g.drawString(msg, 6, 80);
        msg = "  Windows: " + windows.getState();
        g.drawString(msg, 6, 100);
        msg = "  Android: " + android.getState();
```

```

g.drawString(msg, 6, 120);
msg = " Solaris: " + solaris.getState();
g.drawString(msg, 6, 140);
msg = " Mac: " + mac.getState();
g.drawString(msg, 6, 160);
}
}

```

Пример установки флажков в окне апплета `FlowLayoutDemo` приведен на рис.26.10. Сравните его с результатом выполнения апплета `CheckboxDemo` (см. рис. 26.3).



Рис 26.10. Пример установки флажков
в окне апплета `FlowLayoutDemo`

Класс `BorderLayout`

Этот класс реализует общий стиль граничной компоновки для окон переднего плана. Он имеет четыре узких компонента фиксированной ширины по краям и одну крупную область в центре. Четыре стороны именуются по сторонам света: север, юг, запад и восток, а область посередине называется центром. Ниже приведены конструкторы, определяемые в классе `BorderLayout`.

```

BorderLayout()
BorderLayout(int горизонтально, int вертикально)

```

В первой форме конструктора выполняется граничная компоновка по умолчанию. А во второй форме в качестве параметров *горизонтально* и *вертикально* устанавливается горизонтальный и вертикальный промежутки между компонентами.

В классе `BorderLayout` определяются следующие константы для указания областей граничной компоновки:

<code>BorderLayout.CENTER</code>	<code>BorderLayout.SOUTH</code>
<code>BorderLayout.EAST</code>	<code>BorderLayout.WEST</code>
<code>BorderLayout.NORTH</code>	

Эти константы обычно используются при вводе компонентов в компоновку с помощью следующей формы метода `add()` из класса `Container`:

```
void add(Component ссылка_на_компонент, Object область)
```

где параметр *ссылка_на_компонент* обозначает ссылку на вводимый компонент, а *область* — место для ввода компонента в компоновку.

Ниже приведен пример граничной компоновки, где в каждой области размещается отдельный компонент. Пример граничной компоновки в окне апплета BorderLayoutDemo приведен на рис. 26.11.

```
// Продемонстрировать применение граничной компоновки
import java.awt.*;
import java.applet.*;
import java.util.*;
/*
  <applet code="BorderLayoutDemo" width=400 height=200>
  </applet>
*/

public class BorderLayoutDemo extends Applet {
    public void init() {
        setLayout(new BorderLayout());

        add(new Button("This is across the top."), BorderLayout.NORTH);
        // Кнопка, размещаемая сверху по всей ширине окна
        add(new Label("The footer message might go here."),
            BorderLayout.SOUTH);
        // Здесь размещается нижний колонтитул для вывода сообщений
        add(new Button("Right"), BorderLayout.EAST); // Кнопка справа
        add(new Button("Left"), BorderLayout.WEST); // Кнопка слева

        String msg = "The reasonable man adapts " +
            "himself to the world;\n" +
            "the unreasonable one persists in " +
            "trying to adapt the world to himself.\n" +
            "Therefore all progress depends " +
            "on the unreasonable man.\n\n" +
            "        - George Bernard Shaw\n\n";
        add(new TextArea(msg), BorderLayout.CENTER);

    }
}
```

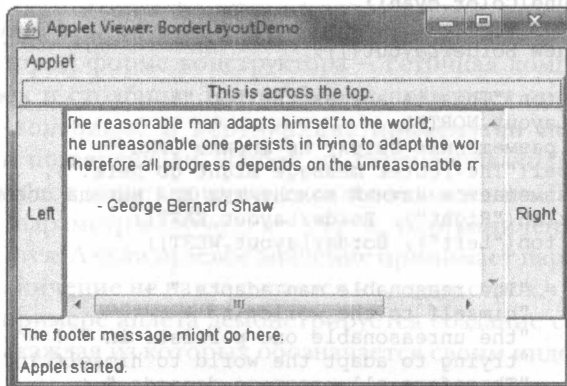


Рис. 26.11. Пример граничной компоновки в окне апплета BorderLayoutDemo

Вставки

Иногда требуется оставить небольшой промежуток между контейнером с компонентами и тем окном, где он находится. Для этого следует переопределить метод `getInsets()` из класса `Container`. Этот метод возвращает объект класса `Insets`, который содержит верхнюю, нижнюю, левую и правую *вставки*, используемые диспетчером компоновки при размещении компонентов контейнера в окне. Ниже приведен конструктор класса `Insets`.

`Insets(int сверху, int слева, int снизу, int справа)`

Значения, передаваемые в качестве параметров *сверху*, *слева*, *снизу* и *справа*, определяют соответствующие промежутки между контейнером и окном, в котором он размещается. Метод `getInsets()` имеет следующую общую форму:

`Insets getInsets()`

Переопределяя этот метод, следует вернуть новый объект типа `Insets`, содержащий требуемый промежуток для вставки. Ниже приведен измененный вариант предыдущего примера, в котором демонстрировалось применение диспетчера компоновки типа `BorderLayout`. В данном варианте компоненты расставляются с отступом десять пикселей от каждой границы. Для фона был выбран голубой цвет, чтобы вставки были лучше видны. Пример граничной компоновки со вставками в окне апплета `InsetsDemo` приведен на рис. 26.12.

```
// Продемонстрировать применение граничной компоновки со вставками
import java.awt.*;
import java.applet.*;
import java.util.*;
/*
   <applet code="InsetsDemo" width=400 height=200>
   </applet>
*/

public class InsetsDemo extends Applet {
    public void init() {
        // задать цвет фона, чтобы легко различать вставки
        setBackground(Color.cyan);

        setLayout(new BorderLayout());

        add(new Button("This is across the top."),
            BorderLayout.NORTH);
        // Кнопка, размещаемая сверху по ширине окна
        add(new Label("The footer message might go here."),
            BorderLayout.SOUTH);
        // Здесь размещается нижний колонтитул для вывода сообщений
        add(new Button("Right"), BorderLayout.EAST);
        add(new Button("Left"), BorderLayout.WEST);

        String msg = "The reasonable man adapts " +
            "himself to the world;\n" +
            "the unreasonable one persists in " +
            "trying to adapt the world to himself.\n" +
            "Therefore all progress depends " +
            "on the unreasonable man.\n\n" +
            "        - George Bernard Shaw\n\n";
    }
}
```

```

        add(new TextArea(msg), BorderLayout.CENTER);
    }
    // добавить вставки
    public Insets getInsets() {
        return new Insets(10, 10, 10, 10);
    }
}

```

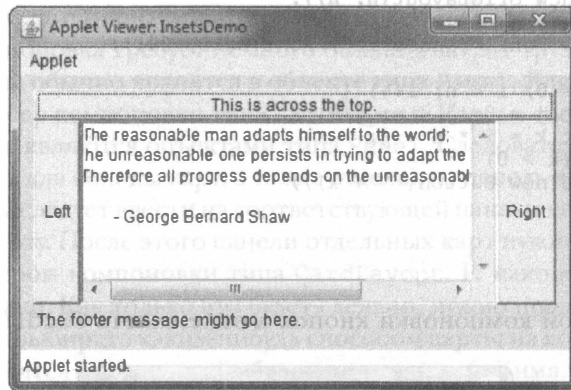


Рис. 26.12. Пример граничной компоновки со вставками в окне апплета InsetsDemo

Класс GridLayout

При использовании класса GridLayout компоненты размещаются табличным способом в двухмерной сетке. Реализуя класс GridLayout, следует определить количество строк и столбцов. Ниже приведены конструкторы, предоставляемые в классе GridLayout.

```

GridLayout()
GridLayout(int количество_строк, int количество_столбцов)
GridLayout(int количество_строк, int количество_столбцов,
           int горизонтально, int вертикально)

```

В первой форме конструктора выполняется сеточная компоновка с одним столбцом, а во второй форме конструктора — сеточная компоновка с заданным количеством строк и столбцов. Третья форма позволяет определить в качестве параметров горизонтально и вертикально промежутки между компонентами по горизонтали и по вертикали. Любой из параметров количество_строк или количество_столбцов может принимать нулевое значение. Так, если нулевое значение принимает параметр количество_строк, то ограничение на ширину столбцов не накладывается. А если нулевое значение принимает параметр количество_столбцов, то ограничение не накладывается на длину строк.

В следующем примере апплета демонстрируется создание сетки 4×4, заполняемой 15 кнопками, каждая из которых обозначается своим индексом:

```

// Продемонстрировать применение сеточной компоновки
import java.awt.*;
import java.applet.*;
/*

```



```

<applet code="GridLayoutDemo" width=300 height=200>
</applet>
*/

public class GridLayoutDemo extends Applet {
    static final int n = 4;
    public void init() {
        setLayout(new GridLayout(n, n));

        setFont(new Font("SansSerif", Font.BOLD, 24));

        for(int i = 0; i < n; i++) {
            for(int j = 0; j < n; j++) {
                int k = i * n + j;
                if(k > 0)
                    add(new Button("" + k));
            }
        }
    }
}

```

Пример сеточной компоновки кнопок в окне апплета GridLayoutDemo приведен на рис. 26.13.

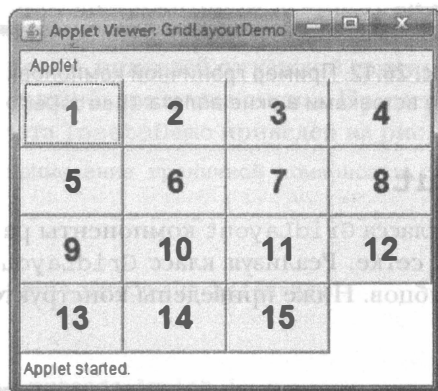


Рис. 26.13. Пример сеточной компоновки кнопок в окне апплета GridLayoutDemo

Совет! Этот пример можно взять за основу для написания программы игры в пятнашки.

Класс CardLayout

Особое место среди классов диспетчеров компоновки принадлежит классу CardLayout, поскольку он позволяет хранить разные компоновки. Каждую компоновку можно представить в виде отдельной карты из колоды. Карты можно перетасовывать как угодно, чтобы в любой момент наверху колоды находилась какая-нибудь карта. Карточная компоновка может оказаться удобной для пользовательских интерфейсов с необязательными компонентами, которые можно динамически включать и отключать в зависимости от вводимых пользователем данных. Имеется возможность подготовить разные виды компоновки и скрыть их до того момента, когда они потребуются.

В классе `CardLayout` предоставляются следующие конструкторы:

```
CardLayout()  
CardLayout(int горизонтально, int вертикально)
```

В первой форме выполняется карточная компоновка по умолчанию. А во второй форме в качестве параметров *горизонтально* и *вертикально* можно указать промежутки между компонентами по горизонтали и по вертикали.

Карточная компоновка требует немного больших затрат труда, чем другие виды компоновки. Карты обычно хранятся в объекте типа `Panel`. Для этой панели следует выбрать диспетчер компоновки типа `CardLayout`. Карты, составляющие колоду, как правило, также являются объектами типа `Panel`. Следовательно, придется сначала создать панель для колоды карт, а также отдельную панель для каждой карты из этой колоды. Затем следует ввести на соответствующей панели компоненты, формирующие каждую карту. После этого панели отдельных карт нужно ввести на главной панели с диспетчером компоновки типа `CardLayout`. И наконец, главную панель следует ввести в окно. Как только это будет сделано, нужно предоставить пользователю возможность выбирать каким-нибудь способом карты из колоды.

Когда карта вводится на панели, ей обычно присваивается имя. Для этой цели чаще всего употребляется приведенная ниже форма метода `add()`, где *имя* обозначает конкретное имя карты, панель которой определяется параметром *ссылка_на_панель*.

```
void add(Component ссылка_на_панель, Object имя)
```

Как только колода карт будет сформирована, отдельные карты в ней активизируются с помощью одного из следующих методов, определяемых в классе `CardLayout`:

```
void first(Container панель)  
void last(Container панель)  
void next(Container панель)  
void previous(Container панель)  
void show(Container панель, String имя_карты)
```

Здесь параметр *панель* обозначает ссылку на контейнер (обычно панель), где хранятся карты, а *имя_карты* — конкретное имя карты. В результате вызова метода `first()` отображается первая карта в колоде. Для отображения последней карты в колоде следует вызвать метод `last()`, для отображения следующей карты — метод `next()`, а для отображения предыдущей карты — метод `previous()`. Методы `next()` и `previous()` автоматически перебирают колоду карт снизу вверх или сверху вниз соответственно. А метод `show()` отображает карту по заданному имени_карты.

В приведенном ниже примере апплета демонстрируется двухуровневая колода карт, из которой пользователь может выбрать операционную систему. Операционные системы типа `Windows` отображаются на одной карте, а операционные системы `Mac OS` и `Solaris` — на другой.

```
// Продемонстрировать применение карточной компоновки  
import java.awt.*;  
import java.awt.event.*;  
import java.applet.*;  
/*  
<applet code="CardLayoutDemo" width=300 height=100>  
</applet>  
*/
```

```

public class CardLayoutDemo extends Applet
    implements ActionListener, MouseListener {
    Checkbox windowsXP, windows7, windows8, android, solaris, mac;
    Panel osCards;
    CardLayout cardLO;
    Button Win, Other;

    public void init() {
        Win = new Button("Windows");
        Other = new Button("Other");
        add(Win);
        add(Other);

        cardLO = new CardLayout();
        osCards = new Panel();
        osCards.setLayout(cardLO); // установить компоновку панели
                                   // для размещения карт

        windowsXP = new Checkbox("Windows XP", null, true);
        windows7 = new Checkbox("Windows 7", null, false);
        windows8 = new Checkbox("Windows 8", null, false);
        android = new Checkbox("Android");
        solaris = new Checkbox("Solaris");
        mac = new Checkbox("Mac OS");

        // ввести на панели флажки для выбора типа ОС Windows
        Panel winPan = new Panel();
        winPan.add(windowsXP);
        winPan.add(windows7);
        winPan.add(windows8);

        // ввести на панели флажки для выбора других ОС
        Panel otherPan = new Panel();
        otherPan.add(android);
        otherPan.add(solaris);
        otherPan.add(mac);

        // ввести панели отдельных карт на панели колоды карт
        osCards.add(winPan, "Windows");
        osCards.add(otherPan, "Other");

        // ввести карты на главной панели апплета
        add(osCards);

        // зарегистрировать приемники событий действия
        Win.addActionListener(this);
        Other.addActionListener(this);

        // зарегистрировать приемники событий от мыши
        addMouseListener(this);
    }

    // перебрать панели карт
    public void mousePressed(MouseEvent me) {
        cardLO.next(osCards);
    }

    // предоставить пустые реализации других методов
    // из интерфейса MouseListener
    public void mouseClicked(MouseEvent me) {
    }
    public void mouseEntered(MouseEvent me) {
    }
}

```

```
public void mouseExited(MouseEvent me) {  
}  
public void mouseReleased(MouseEvent me) {  
}  
  
public void actionPerformed(ActionEvent ae) {  
    if(ae.getSource() == Win) {  
        cardLO.show(osCards, "Windows");  
    }  
    else {  
        cardLO.show(osCards, "Other");  
    }  
}  
}
```

Пример карточной компоновки элементов управления ГПИ в окне апплета CardLayoutDemo приведен на рис. 26.14. Каждая карта активизируется нажатием ее кнопки. Щелчком кнопкой мыши можно также перебирать карты.

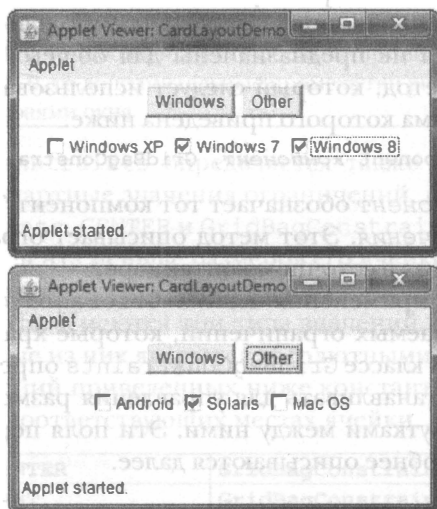


Рис. 26.14. Пример карточной компоновки элементов управления ГПИ в окне апплета CardLayoutDemo

Класс GridBagLayout

Рассмотренные выше виды компоновки вполне пригодны для применения во многих апплетах, тем не менее в некоторых из них требуется более точное управление расположением компонентов в окне. Для этой цели подходит сеточно-контейнерная компоновка, реализуемая в классе GridBagLayout. Удобство такой компоновки состоит в том, что она позволяет задавать относительное расположение компонентов, указывая его в ячейках сетки. Но самое главное, что каждый компонент может иметь свои размеры, а каждая строка — свое количество столбцов. Именно поэтому данная разновидность компоновки называется *сеточно-контейнерной* и представляет собой совокупность мелких соединенных вместе сеток.

Местонахождение и размеры каждого компонента в сеточно-контейнерной компоновке определяются рядом связанных с ним ограничений, которые содержатся в объекте класса `GridBagConstraints`. В частности, ограничения накладываются на высоту и ширину ячейки, расположение компонента, его выравнивание и точку привязки в самой ячейке.

Общая процедура сеточно-контейнерной компоновки выполняется следующим образом. Сначала создается новый объект типа `GridBagLayout` в качестве текущего диспетчера компоновки. Затем накладываются ограничения на каждый компонент, вводимый в сеточный контейнер. После этого компоненты вводятся в диспетчер компоновки. Несмотря на то что класс `GridBagLayout` является более сложным по сравнению с другими диспетчерами компоновки, пользоваться им будет нетрудно, если как следует разобраться в принципе его действия.

В классе `GridBagLayout` определяется следующий единственный конструктор: `GridBagLayout()`

Кроме того, в этом классе определяется ряд методов, многие из которых являются защищенными и не предназначены для общего употребления. Но среди них имеется один метод, который следует использовать. Это метод `setConstraints()`, общая форма которого приведена ниже.

`void setConstraints(Component компонент, GridBagConstraints ограничения)`

Здесь параметр *компонент* обозначает тот компонент, на который накладываются указанные ограничения. Этот метод описывает ограничения, накладываемые на каждый компонент в сеточном контейнере.

Залогом успешного применения класса `GridBagLayout` является тщательная установка накладываемых ограничений, которые хранятся в объекте класса `GridBagConstraints`. В классе `GridBagConstraints` определяется несколько полей, которые можно устанавливать для управления размерами компонентов, их размещением и промежутками между ними. Эти поля перечислены в табл. 26.2. Некоторые из них подробнее описываются далее.

Таблица 26.2. Поля ограничений, определяемые в классе `GridBagConstraints`

Поле	Назначение
<code>int anchor</code>	Задаёт местоположение компонента в ячейке. По умолчанию принимает значение константы <code>GridBagConstraints.CENTER</code>
<code>int fill</code>	Задаёт способ изменения размеров компонента, если они меньше размеров ячейки. Допустимыми являются значения констант <code>GridBagConstraints.NONE</code> (по умолчанию), <code>GridBagConstraints.HORIZONTAL</code> , <code>GridBagConstraints.VERTICAL</code> и <code>GridBagConstraints.BOTH</code>
<code>int gridheight</code>	Задаёт высоту компонента в ячейке. По умолчанию принимает значение 1
<code>int gridwidth</code>	Задаёт ширину компонента в ячейке. По умолчанию принимает значение 1
<code>int gridx</code>	Задаёт координату X ячейки, в которую будет введен компонент. По умолчанию принимает значение константы <code>GridBagConstraints.RELATIVE</code>

Окончание табл. 26.2

Поле	Назначение
int gridy	Задаёт координату Y ячейки, в которую будет введен компонент. По умолчанию принимает значение константы GridBagConstraints.RELATIVE
Insets insets	Задаёт вставки. По умолчанию все вставки являются нулевыми
int ipadx	Задаёт дополнительный промежуток, окружающий компонент в ячейке по горизонтали. По умолчанию принимает нулевое значение
int ipady	Задаёт дополнительный промежуток, окружающий компонент в ячейке по вертикали. По умолчанию принимает нулевое значение
double weightx	Задаёт весовое значение, которое определяет промежутки между ячейками и краями их контейнера по горизонтали. По умолчанию принимает значение 0, 0. Чем больше вес, тем больше промежуток. Если все значения равны 0, 0, то дополнительные промежутки распределяются равномерно между краями окна
double weighty	Задаёт весовое значение, которое определяет промежутки между ячейками и краями их контейнера по вертикали. По умолчанию принимает значение 0, 0. Чем больше вес, тем больше промежуток. Если все значения равны 0, 0, то дополнительные промежутки распределяются равномерно между краями окна

В классе `GridBagConstraints` определяется также ряд статических полей, которые содержат стандартные значения ограничений, например, значения констант `GridBagConstraints.CENTER` и `GridBagConstraints.VERTICAL`.

Если размеры компонента меньше размеров его ячейки, можно воспользоваться полем `anchor`, чтобы определить место в ячейке, где будет располагаться левый верхний угол компонента. Имеются три типа значений, которые можно присвоить полю `anchor`. Первые из них являются абсолютными значениями. Как можно судить по именам значений приведенных ниже констант, они определяют расположение компонента в соответствующих местах ячейки.

<code>GridBagConstraints.CENTER</code>	<code>GridBagConstraints.SOUTH</code>
<code>GridBagConstraints.EAST</code>	<code>GridBagConstraints.SOUTHEAST</code>
<code>GridBagConstraints.NORTH</code>	<code>GridBagConstraints.SOUTHWEST</code>
<code>GridBagConstraints.NORTHEAST</code>	<code>GridBagConstraints.WEST</code>
<code>GridBagConstraints.NORTHWEST</code>	

Второй тип значений, которые можно присвоить полю `anchor`, является относительным, т.е. они указываются относительно ориентации контейнера, которая в восточных языках может быть другой. Относительные значения констант перечислены ниже. Их имена описывают расположение компонентов относительно ячейки.

<code>GridBagConstraints.FIRST_LINE_END</code>	<code>GridBagConstraints.LINE_END</code>
<code>GridBagConstraints.FIRST_LINE_START</code>	<code>GridBagConstraints.LINE_START</code>
<code>GridBagConstraints.LAST_LINE_END</code>	<code>GridBagConstraints.PAGE_END</code>
<code>GridBagConstraints.LAST_LINE_START</code>	<code>GridBagConstraints.PAGE_START</code>

Третий тип значений, которые могут быть присвоены полю `anchor`, позволяет размещать компоненты вертикально по отношению к базовой линии строки. Значения констант этого типа перечислены ниже. При горизонтальном располо-

жении центровка может выполняться относительно переднего (LEADING) или заднему (TRAILING) края.

<code>GridBagConstraints.BASELINE</code>	<code>GridBagConstraints.BASELINE_LEADING</code>
<code>GridBagConstraints.BASELINE_TRAILING</code>	<code>GridBagConstraints.ABOVE_BASELINE</code>
<code>GridBagConstraints.ABOVE_BASELINE_LEADING</code>	<code>GridBagConstraints.ABOVE_BASELINE_TRAILING</code>
<code>GridBagConstraints.BELOW_BASELINE</code>	<code>GridBagConstraints.BELOW_BASELINE_LEADING</code>
<code>GridBagConstraints.BELOW_BASELINE_TRAILING</code>	

Поля `weightx` и `weighty` очень важны, хотя они и кажутся на первый взгляд непонятными. Их значения определяют, сколько дополнительного пространства будет выделено в контейнере для каждой строки и каждого столбца. По умолчанию оба поля имеют нулевые значения. Если все значения в столбце и строке оказываются нулевыми, то дополнительный промежуток распределяется равномерно между краями окна. Увеличивая вес, можно увеличить распределение свободного пространства для строки или столбца пропорционально остальным строкам или столбцам. Самый лучший способ уяснить назначения этих полей — поэкспериментировать с ними на конкретном примере.

В поле `gridwidth` можно задать ширину ячейки в единицах ячейки. По умолчанию эта переменная принимает значение 1. Чтобы компонент использовал свободное пространство в строке, следует установить значение `GridBagConstraints.REMAINDER`, а для того чтобы компонент мог использовать предпоследнюю ячейку в строке — значение `GridBagConstraints.RELATIVE`. Аналогично действует ограничение, накладываемое в поле `gridheight`, но только в вертикальном направлении.

Имеется также возможность определить значение заполнения, чтобы с его помощью увеличить минимальные размеры ячейки. Для заполнения по горизонтали следует установить соответствующее значение в поле `ipadx`, а для заполнения по вертикали — в поле `ipady`.

Ниже приведен пример апплета, в котором класс `GridBagLayout` служит для демонстрации только что рассмотренного материала. Пример сеточно-контейнерной компоновки элементов управления ГПИ в окне `GridBagDemo` приведен на рис. 26.15.

```
// Использовать класс GridBagLayout
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
    <applet code="GridBagDemo" width=250 height=200>
    </applet>
*/

public class GridBagDemo extends Applet
    implements ItemListener {

    String msg = "";
    Checkbox windows, android, solaris, mac;

    public void init() {
        GridBagLayout gbag = new GridBagLayout();
```

```
GridBagConstraints gbc = new GridBagConstraints();
setLayout(gbc);

// определить флажки
windows = new Checkbox("Windows ", null, true);
android = new Checkbox("Android");
solaris = new Checkbox("Solaris");
mac = new Checkbox("Mac OS");

// определить сеточный контейнер

// использовать нулевой вес по умолчанию для первой строки
gbc.weightx = 1.0; // использовать единичный вес для столбца
gbc.ipadx = 200; // заполнить на 200 единиц
gbc.insets = new Insets(4, 4, 0, 0); // сделать небольшую
// вставку относительно левого верхнего угла

gbc.anchor = GridBagConstraints.NORTHEAST;

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbc.setConstraints(windows, gbc);

gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.setConstraints(android, gbc);

// придать второй строке единичный вес
gbc.weighty = 1.0;

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbc.setConstraints(solaris, gbc);

gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.setConstraints(mac, gbc);

// ввести компоненты
add(windows);
add(android);
add(solaris);
add(mac);

// зарегистрировать приемники событий от элементов
windows.addItemListener(this);
android.addItemListener(this);
solaris.addItemListener(this);
mac.addItemListener(this);
}

// перерисовать, когда изменится состояние флажка
public void itemStateChanged(ItemEvent ie) {
    repaint();
}

// отобразить текущее состояние флажков
public void paint(Graphics g) {
    msg = "Current state: "; // Текущее состояние
    g.drawString(msg, 6, 80);
    msg = "  Windows: " + windows.getState();
    g.drawString(msg, 6, 100);
    msg = "  Android: " + android.getState();
    g.drawString(msg, 6, 120);
    msg = "  Solaris: " + solaris.getState();
    g.drawString(msg, 6, 140);
    msg = "  Mac: " + mac.getState();
    g.drawString(msg, 6, 160);
}
}
```




Рис. 26.15. Пример сеточно-контейнерной компоновки элементов управления ГПИ в окне GridBagDemo

В данном виде компоновки флажки выбора операционных систем размещаются в сетке 2×2. Каждая ячейка имеет заполнение 200 единиц. Каждый компонент вставляется с небольшим отступом (4 единицы) относительно левого верхнего угла ячейки. Вес столбца устанавливается единичным, благодаря чему любой дополнительный промежуток по горизонтали распределяется равномерно между столбцами. По умолчанию вес первой строки устанавливается нулевым, а вес второй строки — единичным. Это означает, что любой дополнительный промежуток по вертикали переносится во вторую строку.

Класс `GridBagLayout` реализует весьма эффективный диспетчер компоновки, и он заслуживает того, чтобы потратить некоторое время на его изучение и эксперименты с ним. После того как станет ясно назначение различных устанавливаемых компонентов класса `GridBagLayout`, им станет легче пользоваться для расположения компонентов с высокой степенью точности.

Меню и строки меню

Окно переднего плана может иметь связанную с ним строку меню, которая отображает список пунктов меню верхнего уровня. Каждый пункт меню связан с выпадающим меню. Этот принцип реализован в библиотеке AWT с помощью классов `MenuBar`, `Menu` и `MenuItem`. В общем, строка меню состоит из одного или нескольких объектов класса `Menu`. Каждый объект класса `Menu` содержит список объектов класса `MenuItem`, а каждый объект класса `MenuItem` представляет пункт меню, который может быть выбран пользователем. Класс `Menu` является производным от класса `MenuItem`, что позволяет создать иерархию вложенных подменю. В меню можно также включать отмечаемые пункты типа `CheckboxMenuItem`. Если пользователь щелкнет на таком пункте, то рядом с ним появится отметка в виде галочки.

Чтобы создать строку меню, нужно сначала получить экземпляр класса `MenuBar`. В этом классе определяется только конструктор по умолчанию. Затем следует получить экземпляры класса `Menu`, которые будут определять пункты меню, отображаемые в строке. Ниже приведены конструкторы класса `Menu`.

```
Menu() throws HeadlessException  
Menu(String имя_пункта) throws HeadlessException  
Menu(String имя_пункта, boolean удаляемый_пункт)  
    throws HeadlessException
```

Здесь параметр *имя_пункта* обозначает конкретное имя пункта меню. Если параметр *удаляемый_пункт* принимает логическое значение true, то меню можно удалить, переведя его в плавающий режим, а иначе оно останется присоединенным к строке меню. (Удаляемые меню зависят от конкретной реализации.) В первой форме конструктора создается пустое меню.

Отдельные пункты меню относятся к типу MenuItem. В классе MenuItem определяются следующие конструкторы:

```
MenuItem() throws HeadlessException  
MenuItem(String имя_пункта) throws HeadlessException  
MenuItem(String имя_пункта, MenuShortcut клавиша_доступа)  
    throws HeadlessException
```

где параметр *имя_пункта* обозначает конкретное имя, отображаемое в меню, а параметр *клавиша_доступа* — назначаемую клавишу быстрого доступа к данному пункту меню.

Пункт меню можно активизировать или деактивизировать, вызвав метод `setEnabled()`. Ниже приведена общая форма этого метода.

```
void setEnabled(boolean признак_активизации)
```

Если параметр *признак_активизации* принимает логическое значение true, то пункт меню будет активизирован. А если этот параметр принимает логическое значение false, то пункт меню будет деактивизирован.

Состояние пункта меню можно определить, вызвав метод `isEnabled()`. Ниже приведена общая форма этого метода.

```
boolean isEnabled()
```

Метод `isEnabled()` возвращает логическое значение true, если активизирован пункт меню, для которого он вызывается, а иначе — логическое значение false.

Изменить имя пункта меню можно с помощью метода `setLabel()`, а выяснить текущее имя пункта меню — с помощью метода `getLabel()`. Ниже приведены общие формы этих методов.

```
void setLabel(String новое_имя)  
String getLabel()
```

Здесь параметр *новое_имя* обозначает задаваемое новое имя вызываемого пункта меню. Метод `getLabel()` возвращает текущее имя пункта меню.

Создать отмечаемый пункт меню можно средствами класса `CheckboxMenuItem`, производного от класса `MenuItem`. У этого класса имеются следующие конструкторы:

```
CheckboxMenuItem() throws HeadlessException  
CheckboxMenuItem(String имя_пункта) throws HeadlessException  
CheckboxMenuItem(String имя_пункта, boolean включено)  
    throws HeadlessException
```

где параметр *имя_пункта* обозначает задаваемое имя, отображаемое в меню. Отмечаемые пункты меню действуют подобно переключателям. Всякий раз, когда

слева от одного из пунктов ставится отметка, его состояние изменяется. В первых двух формах конструктора отмечаемое поле не имеет метки. А в третьей форме отмечаемое поле имеет метку, если параметр *включено* принимает логическое значение true. В противном случае это поле остается пустым.

Состояние отмечаемого пункта меню можно выяснить, вызвав метод `getState()`. А присвоить пункту меню определенное состояние можно, вызвав метод `setState()`. Ниже приведены общие формы этих методов.

```
boolean getState()  
void setState(boolean включено)
```

Если пункт меню отмечен, метод `getState()` возвращает логическое значение true, а иначе — логическое значение false. Чтобы отметить пункт меню, достаточно передать методу `setState()` логическое значение true в качестве его единственного параметра. Для того чтобы снять отметку с пункта меню, этому методу следует передать логическое значение false.

После того как пункт меню будет создан, его нужно ввести в объект типа `Menu` с помощью метода `add()`, который имеет следующую общую форму:

```
MenuItem add(MenuItem пункт)
```

где параметр *пункт* обозначает вводимый пункт меню. Ввод пунктов в меню производится в том порядке, в каком осуществлялись вызовы метода `add()`. В конечном итоге возвращается заданный *пункт*.

После ввода всех пунктов в меню типа `Menu` его можно ввести, в свою очередь, в строку меню с помощью следующей версии метода `add()`, определенной в классе `MenuBar`:

```
Menu add(Menu меню)
```

где параметр *меню* обозначает вводимое меню. В конечном итоге возвращается заданное *меню*.

События в меню наступают только при выборе пункта типа `MenuItem` и `CheckboxMenuItem`. Они не наступают, например, при обращении к строке меню для отображения выпадающего меню. Всякий раз, когда выбирается пункт меню, создается объект класса `ActionEvent`. По умолчанию строка с командой действия содержит имя пункта меню. Но если вызвать метод `setActionCommand()` для пункта меню, то можно определить другую строку с командой действия. Всякий раз, когда отмечается пункт меню или снимается его отметка, создается объект класса `ItemEvent`. Таким образом, для обработки этих событий в меню следует реализовать интерфейсы `ActionListener` и `ItemListener`.

Метод `getItem()` из класса `ItemEvent` возвращает ссылку на пункт меню, сгенерировавший данное событие. Ниже приведена общая форма этого метода.

```
Object getItem()
```

Ниже приведен пример, в котором ряд вложенных меню вводится во всплывающее меню. В окне апплета отображается выбранный пункт меню, а также состояние двух отмечаемых пунктов меню. Пример раскрытия меню в окне апплета `MenuDemo` приведен на рис. 26.16.

```
// Пример применения меню
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="MenuDemo" width=250 height=250>
  </applet>
*/

// создать подкласс, производный от класса Frame
class MenuFrame extends Frame {
    String msg = "";
    CheckboxMenuItem debug, test;

    MenuFrame(String title) {
        super(title);

        // создать строку меню и ввести ее в обрамляющее окно
        MenuBar mbar = new MenuBar();
        setMenuBar(mbar);

        // создать пункты меню
        Menu file = new Menu("File");
        MenuItem item1, item2, item3, item4, item5;
        file.add(item1 = new MenuItem("New..."));
        file.add(item2 = new MenuItem("Open..."));
        file.add(item3 = new MenuItem("Close"));
        file.add(item4 = new MenuItem("-"));
        file.add(item5 = new MenuItem("Quit..."));
        mbar.add(file);

        Menu edit = new Menu("Edit");
        MenuItem item6, item7, item8, item9;
        edit.add(item6 = new MenuItem("Cut"));
        edit.add(item7 = new MenuItem("Copy"));
        edit.add(item8 = new MenuItem("Paste"));
        edit.add(item9 = new MenuItem("-"));

        Menu sub = new Menu("Special");
        MenuItem item10, item11, item12;
        sub.add(item10 = new MenuItem("First"));
        sub.add(item11 = new MenuItem("Second"));
        sub.add(item12 = new MenuItem("Third"));
        edit.add(sub);

        // создать отмечаемые пункты меню
        debug = new CheckboxMenuItem("Debug");
        edit.add(debug);
        test = new CheckboxMenuItem("Testing");
        edit.add(test);

        mbar.add(edit);

        // создать объект для обработки событий действия
        // и событий от элементов
        MyMenuHandler handler = new MyMenuHandler(this);
        // зарегистрировать этот объект для приема событий
        // действия и событий от элементов
        item1.addActionListener(handler);
        item2.addActionListener(handler);
        item3.addActionListener(handler);
        item4.addActionListener(handler);
        item5.addActionListener(handler);
        item6.addActionListener(handler);
```

```

        item7.addActionListener(handler);
        item8.addActionListener(handler);
        item9.addActionListener(handler);
        item10.addActionListener(handler);
        item11.addActionListener(handler);
        item12.addActionListener(handler);
        debug.addItemListener(handler);
        test.addItemListener(handler);

        // создать объект для обработки оконных событий
        MyWindowAdapter adapter = new MyWindowAdapter(this);

        // зарегистрировать этот объект для приема оконных событий
        addWindowListener(adapter);
    }

    public void paint(Graphics g) {
        g.drawString(msg, 10, 200);

        if(debug.getState())
            g.drawString("Debug is on.", 10, 220);
            // Отладка включена
        else
            g.drawString("Debug is off.", 10, 220);
            // Отладка отключена
        if(test.getState())
            g.drawString("Testing is on.", 10, 240);
            // Тестирование включено
        else
            g.drawString("Testing is off.", 10, 240);
            // Тестирование отключено
    }
}

class MyWindowAdapter extends WindowAdapter {
    MenuFrame menuFrame;
    public MyWindowAdapter(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }

    public void windowClosing(WindowEvent we) {
        menuFrame.setVisible(false);
    }
}

class MyMenuHandler implements ActionListener, ItemListener {
    MenuFrame menuFrame;
    public MyMenuHandler(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }
    // обработать события действия
    public void actionPerformed(ActionEvent ae) {
        String msg = "You selected "; // Выбран пункт меню
        String arg = ae.getActionCommand();
        if(arg.equals("New..."))
            msg += "New.";
        else if(arg.equals("Open..."))
            msg += "Open.";
        else if(arg.equals("Close"))
            msg += "Close.";
        else if(arg.equals("Quit..."))
            msg += "Quit.";
        else if(arg.equals("Edit"))
            msg += "Edit.";
        else if(arg.equals("Cut"))

```

```

        msg += "Cut.";
    else if(arg.equals("Copy"))
        msg += "Copy.";
    else if(arg.equals("Paste"))
        msg += "Paste.";
    else if(arg.equals("First"))
        msg += "First.";
    else if(arg.equals("Second"))
        msg += "Second.";
    else if(arg.equals("Third"))
        msg += "Third.";
    else if(arg.equals("Debug"))
        msg += "Debug.";
    else if(arg.equals("Testing"))
        msg += "Testing.";
    menuFrame.msg = msg;
    menuFrame.repaint();
}
// обработать события от элементов
public void itemStateChanged(ItemEvent ie) {
    menuFrame.repaint();
}
}

// создать обрамляющее окно
public class MenuDemo extends Applet {
    Frame f;
    public void init() {
        f = new MenuFrame("Menu Demo"); // Демонстрационное меню
        int width = Integer.parseInt(getParameter("width"));
        int height = Integer.parseInt(getParameter("height"));

        setSize(new Dimension(width, height));

        f.setSize(width, height);
        f.setVisible(true);
    }

    public void start() {
        f.setVisible(true);
    }

    public void stop() {
        f.setVisible(false);
    }
}

```

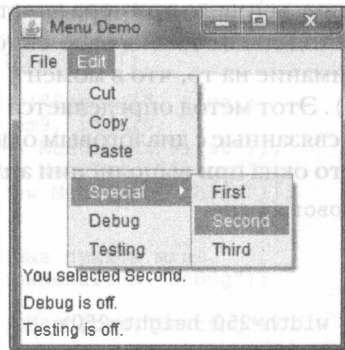


Рис. 26.16. Пример раскрытия меню в окне апплета MenuDemo

Имеется также класс `PopupMenu`, который представляет интерес для создания всплывающих или контекстных меню. Этот класс действует подобно классу `Menu`, но формирует меню, отображаемое в определенном месте. Класс `PopupMenu` предоставляет гибкую и удобную альтернативу в некоторых случаях организации меню.

Диалоговые окна

Нередко возникает потребность размещать ряд связанных вместе элементов управления в *диалоговом окне*. Диалоговые окна служат, в первую очередь, для получения данных, вводимых пользователем. И зачастую они оказываются дочерними окнами по отношению к окну верхнего уровня. У диалоговых окон отсутствует строка меню, а в остальном они функционируют подобно обрамляющим окнам. (В них можно, например, вводить элементы управления таким же образом, как и в обрамляющее окно.) Диалоговые окна могут быть модальными (т.е. режимными) или немодальными (безрежимными). Когда активным становится *модальное* диалоговое окно, то все вводимые данные направляются в него до тех пор, пока оно остается открытым. Это означает, что остальные части прикладной программы недоступны до тех пор, пока не будет закрыто диалоговое окно. Если активным становится *немодальное* диалоговое окно, то фокус ввода может быть передан другому окну прикладной программы. Таким образом, остальные части прикладной программы остаются активными и доступными. Диалоговые окна относятся к типу `Dialog`. Ниже приведены два наиболее употребительных конструктора класса `Dialog`.

```
Dialog(Frame родительское_окно, boolean режим)
Dialog(Frame родительское_окно, String заголовок, boolean режим)
```

Здесь параметр *родительское_окно* обозначает владельца диалогового окна. Если параметр *режим* принимает логическое значение `true`, то создаваемое диалоговое окно становится модальным, а иначе — немодальным. Заглавие диалогового окна можно указать в качестве параметра *заголовок*. Как правило, для создания диалоговых окон сначала получают подклассы, производные от класса `Dialog`, а затем они наделяются функциональными возможностями, требующимися для конкретного приложения.

Ниже приведен вариант предыдущего примера апплета, переделанный таким образом, чтобы отображать немодальное диалоговое окно при выборе пункта меню **New** (Создать). Обратите внимание на то, что в момент закрытия диалогового окна вызывается метод `dispose()`. Этот метод определяется в классе `Window` и освобождает все системные ресурсы, связанные с диалоговым окном. На рис. 26.17 приведен пример открытия диалогового окна при выполнении апплета `DialogDemo`.

```
// Пример применения диалогового окна
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="DialogDemo" width=250 height=250>
  </applet>
*/
// создать подкласс, производный от класса Dialog
```

```

class SampleDialog extends Dialog implements ActionListener {
    SampleDialog(Frame parent, String title) {
        super(parent, title, false);
        setLayout(new FlowLayout());
        setSize(300, 200);

        add(new Label("Press this button:"));
        Button b;
        add(b = new Button("Cancel"));
        b.addActionListener(this);
    }

    public void actionPerformed(ActionEvent ae) {
        dispose();
    }

    public void paint(Graphics g) {
        g.drawString("This is in the dialog box", 10, 70);
        // Это сообщение выводится в диалоговом окне
    }
}

// создать подкласс, производный от класса Frame
class MenuFrame extends Frame {
    String msg = "";
    CheckboxMenuItem debug, test;

    MenuFrame(String title) {
        super(title);

        // создать строку меню и ввести ее в обрамляющем окне
        MenuBar mbar = new MenuBar();
        setMenuBar(mbar);

        // создать пункты меню
        Menu file = new Menu("File");
        MenuItem item1, item2, item3, item4;
        file.add(item1 = new MenuItem("New..."));
        file.add(item2 = new MenuItem("Open..."));
        file.add(item3 = new MenuItem("Close"));
        file.add(new MenuItem("-"));
        file.add(item4 = new MenuItem("Quit..."));
        mbar.add(file);

        Menu edit = new Menu("Edit");
        MenuItem item5, item6, item7;
        edit.add(item5 = new MenuItem("Cut"));
        edit.add(item6 = new MenuItem("Copy"));
        edit.add(item7 = new MenuItem("Paste"));
        edit.add(new MenuItem("-"));

        Menu sub = new Menu("Special", true);
        MenuItem item8, item9, item10;
        sub.add(item8 = new MenuItem("First"));
        sub.add(item9 = new MenuItem("Second"));
        sub.add(item10 = new MenuItem("Third"));
        edit.add(sub);

        // создать отмечаемые пункты меню
        debug = new CheckboxMenuItem("Debug");
        edit.add(debug);
        test = new CheckboxMenuItem("Testing");
        edit.add(test);

        mbar.add(edit);
    }
}

```



```

// создать объект для обработки событий действия
// и событий от элементов
MyMenuHandler handler = new MyMenuHandler(this);
// зарегистрировать этот объект для приема событий
// действия и событий от элементов
item1.addActionListener(handler);
item2.addActionListener(handler);
item3.addActionListener(handler);
item4.addActionListener(handler);
item5.addActionListener(handler);
item6.addActionListener(handler);
item7.addActionListener(handler);
item8.addActionListener(handler);
item9.addActionListener(handler);
item10.addActionListener(handler);
debug.addItemListener(handler);
test.addItemListener(handler);

// создать объект для обработки оконных событий
MyWindowAdapter adapter = new MyWindowAdapter(this);
// зарегистрировать этот объект для приема оконных событий
addWindowListener(adapter);
}

public void paint(Graphics g) {
    g.drawString(msg, 10, 200);
    if(debug.getState())
        g.drawString("Debug is on.", 10, 220);
        // Отладка включена
    else
        g.drawString("Debug is off.", 10, 220);
        // Отладка отключена
    if(test.getState())
        g.drawString("Testing is on.", 10, 240);
        // Тестирование включено
    else
        g.drawString("Testing is off.", 10, 240);
        // Тестирование отключено
}
}

class MyWindowAdapter extends WindowAdapter {
    MenuFrame menuFrame;
    public MyWindowAdapter(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }
    public void windowClosing(WindowEvent we) {
        menuFrame.dispose();
    }
}

class MyMenuHandler implements ActionListener, ItemListener {
    MenuFrame menuFrame;
    public MyMenuHandler(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }
    // обработать события действия
    public void actionPerformed(ActionEvent ae) {
        String msg = "You selected ";
        String arg = ae.getActionCommand();
        // активизировать диалоговое окно при выборе пункта меню New
        if(arg.equals("New...")) {
            msg += "New.";
            SampleDialog d = new
                SampleDialog(menuFrame, "New Dialog Box");

```

```

        d.setVisible(true);
    }
    // попытаться определить диалоговые окна
    // для остальных пунктов меню
    else if(arg.equals("Open..."))
        msg += "Open.";
    else if(arg.equals("Close"))
        msg += "Close.";
    else if(arg.equals("Quit..."))
        msg += "Quit.";
    else if(arg.equals("Edit"))
        msg += "Edit.";
    else if(arg.equals("Cut"))
        msg += "Cut.";
    else if(arg.equals("Copy"))
        msg += "Copy."; // Копирование
    else if(arg.equals("Paste"))
        msg += "Paste."; // Вставка
    else if(arg.equals("First"))
        msg += "First."; // Первое окно
    else if(arg.equals("Second"))
        msg += "Second."; // Второе окно
    else if(arg.equals("Third"))
        msg += "Third."; // Третье окно
    else if(arg.equals("Debug"))
        msg += "Debug."; // Отладка
    else if(arg.equals("Testing"))
        msg += "Testing."; // Тестирование
    menuFrame.msg = msg;
    menuFrame.repaint();
}

public void itemStateChanged(ItemEvent ie) {
    menuFrame.repaint();
}
}

// создать обрамляющее окно
public class DialogDemo extends Applet {
    Frame f;

    public void init() {
        f = new MenuFrame("Menu Demo");
        int width = Integer.parseInt(getParameter("width"));
        int height = Integer.parseInt(getParameter("height"));

        setSize(width, height);

        f.setSize(width, height);
        f.setVisible(true);
    }

    public void start() {
        f.setVisible(true);
    }

    public void stop() {
        f.setVisible(false);
    }
}

```

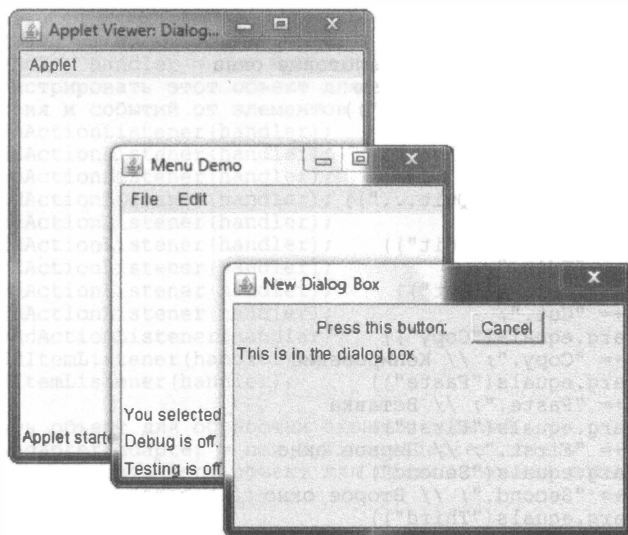


Рис. 26.17. Пример открытия диалогового окна при выполнении апплета DialogDemo.

Диалоговые окна выбора файлов

В Java предоставляется встроенный класс `FileDialog`, реализующий диалоговое окно, в котором пользователь может выбрать файл. Чтобы создать диалоговое окно выбора файлов, следует получить экземпляр класса `FileDialog`. В итоге появится диалоговое окно выбора файлов. Как правило, это стандартное диалоговое окно, предоставляемое операционной системой для выбора файлов. Ниже приведены конструкторы класса `FileDialog`.

```
FileDialog(Frame родитель, String заголовок_окна)  
FileDialog(Frame родитель, String заголовок_окна, int способ)  
FileDialog(Frame родитель)
```

Здесь параметр *родитель* обозначает владельца диалогового окна, а параметр *заголовок_окна* — название, отображаемое в строке заголовка диалогового окна. Если параметр *заголовок_окна* не указан, то заголовок диалогового окна отображается. Если параметр *способ* принимает значение константы `FileDialog.LOAD`, то файл выбирается в диалоговом окне для чтения. А если этот параметр принимает значение константы `FileDialog.SAVE`, то файл выбирается для записи. Если же параметр *способ* не указан, то по умолчанию файл выбирается в диалоговом окне для чтения.

В классе `FileDialog` предоставляются методы, позволяющие определить имя файла и путь к нему, после того как этот файл будет выбран пользователем. Ниже приведены два примера объявления этих методов. Эти методы возвращают каталог и имя файла соответственно.

```
String getDirectory()  
String getFile()
```

В следующем примере программы активизируется стандартное диалоговое окно выбора файлов:

```
/* Продемонстрировать применение диалогового окна выбора файлов.  
Это – прикладная программа, а не апплет.
```

```
*/
```

```
import java.awt.*;
```

```
import java.awt.event.*;
```

```
// создать подкласс, производный от класса Frame
```

```
class SampleFrame extends Frame {
```

```
    SampleFrame(String title) {
```

```
        super(title);
```

```
        // удалить окно после его закрытия
```

```
        addWindowListener(new WindowAdapter() {
```

```
            public void windowClosing(WindowEvent we) {
```

```
                System.exit(0);
```

```
            }
```

```
        });
```

```
    }
```

```
}
```

```
// Продемонстрировать применение класса FileDialog
```

```
class FileDialogDemo {
```

```
    public static void main(String args[]) {
```

```
        // создать обрамляющее окно, которому будет
```

```
        // принадлежать диалоговое окно
```

```
        Frame f = new SampleFrame("File Dialog Demo");
```

```
        f.setVisible(true);
```

```
        f.setSize(100, 100);
```

```
        FileDialog fd = new FileDialog(f, "File Dialog");
```

```
        fd.setVisible(true);
```

```
    }
```

```
}
```

На рис. 26.18 приведен примерный результат, выводимый данной программой. (Конфигурация диалогового окна может отличаться в зависимости от конкретной исполняющей среды.)

И последнее замечание: начиная с версии JDK 7 класс `FileDialog` можно использовать для выбора файлов из списка. Соответствующие функциональные возможности обеспечивают методы `setMultipleMode()`, `isMultipleMode()` и `getFiles()`.

0 переопределении метода `paint()`

Прежде чем завершить рассмотрение элементов управления из библиотеки AWT, следует сказать несколько слов о переопределении метода `paint()`. Хотя это и не отражено в простых примерах, демонстрирующих применение библиотеки AWT в этой книге, при переопределении метода `paint()` иногда необходимо обращаться к реализации метода `paint()` в суперклассе. Поэтому в некоторых программах придется воспользоваться следующим скелетом метода `paint()`:

```

public void paint(Graphics g) {
    // код перерисовки данного окна

    // вызвать метод paint() из суперкласса
    super.paint(g);
}

```

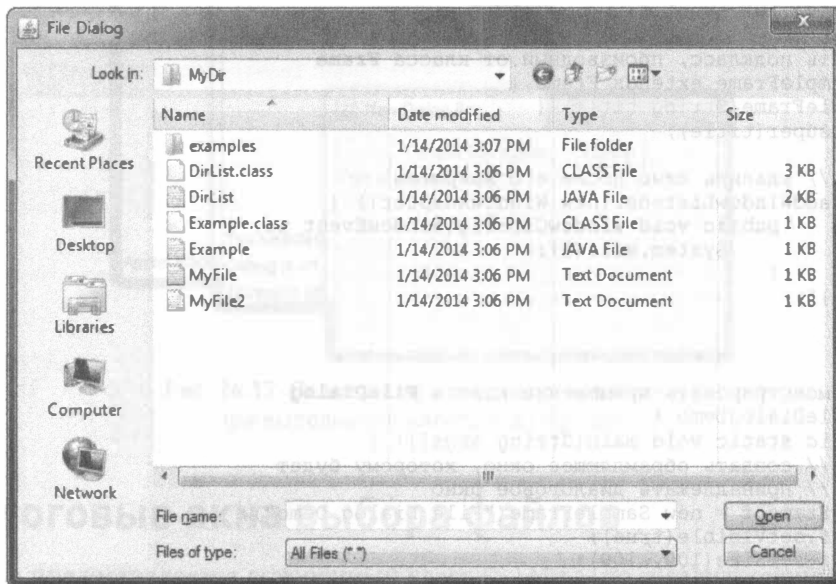


Рис. 26.18. Пример открытия диалогового окна выбора файлов

В языке Java имеются два общих типа компонентов: *тяжеловесные* и *легковесные*. У тяжеловесного компонента имеется свое базовое окно, а легковесный компонент полностью реализуется в коде Java и использует окно, предоставляемое родителем. Все элементы управления из библиотеки AWT, упомянутые в этой главе, являются тяжеловесными. Но если контейнер содержит какие-нибудь легковесные компоненты (т.е. имеет дочерние легковесные компоненты), то при перераспределении метода `paint()` для этого контейнера следует вызвать метод `super.paint()`. Вызовом метода `super.paint()` гарантируется правильность перерисовки любых легковесных дочерних компонентов, например, легковесных элементов управления. Если неясен тип дочернего компонента, то для его выяснения можно вызвать метод `isLightweight()`, определенный в классе `Component`. Этот метод возвращает логическое значение `true`, если компонент является легковесным, а иначе — логическое значение `false`.

В этой главе рассматриваются класс `Image` из библиотеки AWT и пакет `java.awt.image`. Совместно они обеспечивают поддержку *формирования изображений*, которое заключается в отображении графических изображений и манипулировании ими. *Изображением* является обычный прямоугольный графический объект. Изображения являются ключевым компонентом разработки веб-приложений. Когда в 1993 году разработчики из NCSA (National Center for Supercomputer Applications — Национальный центр по применению суперкомпьютеров) решили включить дескриптор `` в браузер Mosaic, это способствовало быстрому развитию веб. Этот дескриптор использовался для *встраивания* изображений в поток гипертекста. Язык Java расширяет этот базовый принцип, позволяя управлять изображениями программным способом. Вследствие этой важной особенности в Java обеспечивается всесторонняя поддержка формирования изображений.

Изображения представлены объектами класса `Image`, который входит в пакет `java.awt`. Манипулирование изображениями осуществляется с помощью классов из пакета `java.awt.image`, который содержит большое количество классов и интерфейсов для формирования изображений. Не имея возможности рассмотреть каждый класс и интерфейс в отдельности, сосредоточим основное внимание на тех из них, которые участвуют в основном процессе формирования изображений. Ниже перечислены классы из пакета `java.awt.image`, которые рассматриваются в этой главе.

CropImageFilter	MemoryImageSource
FilteredImageSource	PixelGrabber
ImageFilter	RGBImageFilter

В примерах из этой главы применяются следующие интерфейсы:

ImageConsumer	ImageObserver	ImageProducer
----------------------	----------------------	----------------------

Кроме того, в главе рассматривается класс `MediaTracker`, который входит в пакет `java.awt`.

Форматы файлов

Первоначально веб-изображения могли быть представлены только в формате GIF. Формат изображений GIF был разработан специалистами из компании CompuServe в 1987 году для просмотра изображений в оперативном режиме, по-

этому он был пригоден и для Интернета. Каждое изображение формата GIF может содержать не более 256 цветов. В связи с этим ограничением в 1995 году ведущие разработчики браузеров включили в них поддержку изображений формата JPEG. Формат JPEG был разработан группой специалистов по фотографии для хранения полутоновых изображений с полным спектром цветов. Если правильно сформировать подобное изображение, оно будет воспроизводиться с более высокой степенью точности и уплотняться более компактно, чем аналогичное изображение формата GIF. Имеется также формат файлов изображений PNG, который является разновидностью формата GIF. Как правило, в прикладных программах вряд ли придется обращать особое внимание на используемый формат изображений, поскольку классы изображений в Java абстрагируют все отличия в форматах благодаря ясно определенному интерфейсу.

Основы работы с изображениями: создание, загрузка и отображение

В процессе обработки изображений обычно выполняются в основном три операции: формирование изображения, его загрузка и воспроизведение. Для обращения к изображениям, хранящимся в оперативной памяти, а также к изображениям, загружаемые из внешних источников данных, в Java предоставляется класс `Image`. Таким образом, в Java предоставляются способы создания нового объекта изображения и его загрузки, а также функциональные средства, позволяющие воспроизводить изображения. Все эти средства будут рассмотрены далее по очереди.

Создание объекта класса `Image`

На первый взгляд может показаться, что для формирования изображения в оперативной памяти требуется примерно такая строка кода:

```
Image test = new Image(200, 100); // Ошибка – не работает!
```

Но на самом деле это не так. Любое изображение предназначено для того, чтобы его можно было воспроизвести на экране монитора, а класс `Image` не располагает достаточными сведениями о среде для создания подходящего формата данных, выводимых на экран. Поэтому в класс `Component` из пакета `java.awt` включен метод `createImage()`, предназначенный для создания объектов типа `Image`. (Напомним, что все компоненты из библиотеки AWT являются производными от класса `Component`, поэтому все они поддерживают метод `createImage()`.)

Метод `createImage()` имеет следующие общие формы:

```
Image createImage(ImageProducer поставщик_изображений)
Image createImage(int ширина, int высота)
```

В первой форме данного метода возвращается изображение, сформированное указанным *поставщиком_изображений*, который представляет собой объект класса, реализующего интерфейс `ImageProducer`. (О поставщиках изображений речь пойдет далее в этой главе.) Во второй форме возвращается пустое изображе-

ние, имеющее определенную ширину и высоту. Ниже приведен пример создания объекта пустого изображения.

```
Canvas c = new Canvas();  
Image test = c.createImage(200, 100);
```

В данном примере для создания объекта типа `Image` сначала получается экземпляр класса `Canvas`, а затем вызывается метод `createImage()`. Полученный объект изображения остается пока еще пустым. Далее будет показано, как заполнить его данными.

Загрузка изображения

Другой способ получить изображение состоит в том, чтобы загрузить его. Для этой цели служит метод `getImage()`, определенный в классе `Applet`. Он имеет следующие общие формы:

```
Image getImage(URL url)  
Image getImage(URL url, String имя_изображения)
```

В первой форме данного метода возвращается объект типа `Image`, инкапсулирующий изображение, находящееся в месте, определяемом параметром `url`. А во второй форме возвращается объект типа `Image`, инкапсулирующий изображение, местонахождение и имя которого определяются параметрами `url` и `имя_изображения` соответственно.

Воспроизведение изображения

Итак, получив изображение, можно воспроизвести его с помощью метода `drawImage()`, который является членом класса `Graphics`. У этого метода имеется несколько форм. Ниже представлена его общая форма, которая употребляется в данной книге.

```
boolean drawImage(Image объект_изображения, int слева, int сверху,  
ImageObserver объект_изображения)
```

В данном случае воспроизводится изображение, передаваемое в качестве параметра `объект_изображения`, а левый верхний угол этого изображения определяется параметрами `слева` и `сверху`. Параметр `объект_изображения` обозначает ссылку на класс, реализующий интерфейс `ImageObserver`. Этот интерфейс реализуют все компоненты библиотек `AWT` и `Swing`. *Наблюдатель изображения* представляет собой объект, который может наблюдать за изображением во время его загрузки. Интерфейс `ImageObserver` рассматривается в следующем разделе.

Загрузить и воспроизвести изображение методами `getImage()` и `drawImage()` совсем не трудно. Ниже приведен пример апплета, загружающего и отображающего отдельное изображение. В этом апплете загружается файл изображения `Lilies.jpg`, но для данного примера можно взять любой другой файл изображения формата `GIF`, `JPG` или `PNG`, при условии, что он будет находиться в том же каталоге, что и `HTML`-файл, содержащий апплет.

```
/*
```

```
* <applet code="SimpleImageLoad" width=400 height=345>
```



```

* <param name="img" value="Lilies.jpg">
* </applet>
*/
import java.awt.*;
import java.applet.*;

public class SimpleImageLoad extends Applet
{
    Image img;

    public void init() {
        img = getImage(getDocumentBase(), getParameter("img"));
    }

    public void paint(Graphics g) {
        g.drawImage(img, 0, 0, this);
    }
}

```

В методе `init()` переменной `img` присваивается изображение, возвращаемое методом `getImage()`. Символьная строка, возвращаемая в результате вызова метода `getParameter("img")`, используется в методе `getImage()` в качестве имени файла изображения. Это изображение загружается по URL, заданному относительно результата выполнения метода `getDocumentBase()`, т.е. по URL той HTML-страницы, на которой находился дескриптор данного апплета. Имя файла, возвращаемое в результате вызова метода `getParameter("img")`, получается из дескриптора апплета `<param name="img" value="Lilies.jpg">`. Такая разметка эквивалентна применению дескриптора HTML-разметки ``, хотя она действует немного медленнее. На рис. 27.1 представлен результат выполнения данного апплета.

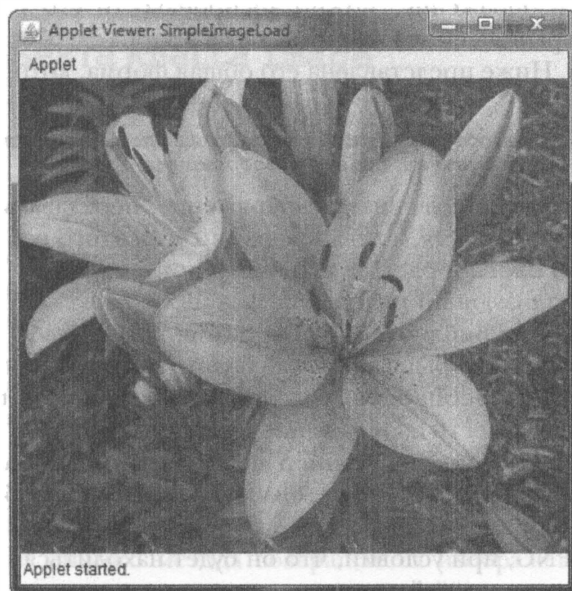


Рис. 27.1. Воспроизведение загруженного изображения в окне апплета `SimpleImageLoad`

Выполнение данного апплета начинается с загрузки изображения `img` в методе `init()`. Изображение будет появляться на экране монитора по мере его загрузки из сети, поскольку при реализации интерфейса `ImageObserver` в классе `Applet` метод `paint()` вызывается всякий раз, когда поступают новые данные изображения.

Разумеется, совсем не плохо иметь возможность наблюдать за ходом загрузки изображения, но время ожидания загрузки изображения лучше потратить на параллельное выполнение других задач. Это позволит отобразить полностью сформированное изображение на экране монитора сразу же после его загрузки. Интерфейс `ImageObserver`, о котором пойдет речь далее, можно применять для контроля за ходом загрузки изображения в то время, как на экран монитора выводится какая-нибудь другая информация.

Интерфейс `ImageObserver`

Этот интерфейс используется для получения уведомления о формировании изображения. В нем определяется единственный метод `imageUpdate()`. Используя наблюдатель изображений типа `ImageObserver`, можно выполнять ряд других действий, в том числе отображать индикатор выполнения или же переключаться на экран при получении уведомления о ходе выполнения загрузки. Такой тип уведомлений удобен для загрузки изображений по низкоскоростной сети.

У метода `imageUpdate()` имеется следующая общая форма:

```
boolean imageUpdate(Image объект_изображения, int признаки, int слева,  
                    int сверху, int ширина, int высота)
```

где параметр *объект_изображения* обозначает загружаемое изображение, а параметр *признаки* — целое число, передающее состояние обновляемого отчета. Четыре целочисленных параметра, *слева*, *сверху*, *ширина* и *высота*, определяют прямоугольник, различные значения которого зависят от значений, передаваемых в качестве параметра *признаки*. Метод `imageUpdate()` возвращает логическое значение `false`, если процесс загрузки завершен, или логическое значение `true`, если требуется обработать еще одно изображение.

Параметр *признаки* содержит один или несколько поразрядных признаков, определяемых в качестве статических переменных в интерфейсе `ImageObserver`. Эти признаки, а также передаваемые с их помощью данные перечислены в табл. 27.1.

Таблица 27.1. Поразрядные признаки, обозначаемые параметром *признаки* в методе `imageUpdate()`

Признак	Назначение
WIDTH	Параметр <i>ширина</i> является действительным и содержит значение ширины изображения
HEIGHT	Параметр <i>высота</i> является действительным и содержит значение высоты изображения
PROPERTIES	Свойства, связанные с изображением, можно получить с помощью метода <code>imgObj.getProperty()</code>

Признак	Назначение
SOMEBITS	Получены дополнительные пиксели, необходимые для рисования изображения. Параметры <i>слева</i> , <i>сверху</i> , <i>ширина</i> и <i>высота</i> определяют прямоугольник, содержащий новые пиксели
FRAMEBITS	Получен весь кадр, являющийся частью ранее нарисованного многокадрового изображения. Этот кадр можно отобразить. Параметры <i>слева</i> , <i>сверху</i> , <i>ширина</i> и <i>высота</i> не используются
ALLBITS	Изображение готово. Параметры <i>слева</i> , <i>сверху</i> , <i>ширина</i> и <i>высота</i> не используются
ERROR	Обнаружена ошибка в асинхронно отслеживаемом изображении. Изображение не готово и не может быть воспроизведено. Никаких дополнительных сведений об изображении не получено. Кроме того, будет установлен признак ABORT , чтобы указать, что процесс формирования изображения был прерван
ABORT	Формирование асинхронно отслеживаемого изображения было прервано до полного завершения этого процесса. Но если не произошло ошибки, то при попытке обратиться к какой-нибудь части данных изображения начнется повторное формирование изображения

В классе `Applet` имеется реализация метода `imageUpdate()` для интерфейса `ImageObserver`. Этот метод служит для перерисовки изображений во время их загрузки. Такое поведение можно изменить, если переопределить данный метод в своем классе. Ниже приведен пример применения метода `imageUpdate()`.

```
public boolean imageUpdate(Image img, int flags,
                           int x, int y, int w, int h) {
    if ((flags & ALLBITS) == 0) {
        System.out.println("Still processing the image.");
        // Изображение все еще обрабатывается
        return true;
    } else {
        System.out.println("Done processing the image.");
        // Обработка изображения завершена
        return false;
    }
}
```

Двойная буферизация

Изображения можно использовать не только для хранения фотографий и рисунков, как было показано выше, но и в качестве внеэкранных поверхностей рисования. С их помощью можно воспроизвести любое изображение, включая текст и графику во внеэкранном буфере, содержимое которого можно отобразить некоторое время спустя. Преимущество такого подхода заключается в том, что изображение можно увидеть лишь после того, как оно будет полностью готово к воспроизведению. Для рисования сложного изображения может потребоваться несколько миллисекунд или больше, причем для пользователя этот процесс может выглядеть как серия вспышек или мерцаний.

Подобные эффекты отвлекают внимание и приводят к тому, что пользователь воспринимает воспроизводимое изображение намного медленнее, чем это происходит на самом деле. Процесс использования внеэкранного изображения для уменьшения мерцания называется *двойной буферизацией*, поскольку экран монитора принимается в качестве буфера для пикселей воспроизводимого изображения, а внеэкранное изображение является вторым буфером, где можно подготавливать отдельные пиксели к воспроизведению.

Ранее в этой главе уже пояснялось, как создается пустой объект класса `Image`. А здесь будет показано, каким образом выполняется воспроизведение изображения на самом экране. Напомним, для этой цели требуется объект класса `Graphics`, благодаря которому можно использовать любые методы визуализации, доступные в Java. Он удобен тем, что доступ к объекту класса `Graphics`, который можно применять для воспроизведения изображения, осуществляется с помощью метода `getGraphics()`. Ниже представлен фрагмент кода, в котором формируется новое изображение, получается графический контекст и все изображение заполняется пикселями красного цвета.

```
Canvas c = new Canvas();
Image test = c.createImage(200, 100);
Graphics gc = test.getGraphics();
gc.setColor(Color.red);
gc.fillRect(0, 0, 200, 100);
```

После того как внеэкранное изображение будет создано и заполнено, его по-прежнему не будет видно. Чтобы воспроизвести это изображение, следует вызвать метод `drawImage()`. Ниже приведен пример апплета, где для воспроизведения изображения требуется немало времени. Этот пример демонстрирует, насколько двойная буферизация оказывает влияние на восприятие времени воспроизведения.

```
/*
 * <applet code=DoubleBuffer width=250 height=250>
 * </applet>
 */
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class DoubleBuffer extends Applet {
    int gap = 3;
    int mx, my;
    boolean flicker = true;
    Image buffer = null;
    int w, h;

    public void init() {
        Dimension d = getSize();
        w = d.width;
        h = d.height;
        buffer = createImage(w, h);
        addMouseListener(new MouseMotionAdapter() {
            public void mouseDragged(MouseEvent me) {
                mx = me.getX();
                my = me.getY();
                flicker = false;
                repaint();
            }
        });
    }
}
```

```

    }
    public void mouseMoved(MouseEvent me) {
        mx = me.getX();
        my = me.getY();
        flicker = true;
        repaint();
    }
    });
}

public void paint(Graphics g) {
    Graphics screengc = null;

    if (!flicker) {
        screengc = g;
        g = buffer.getGraphics();
    }

    g.setColor(Color.blue);
    g.fillRect(0, 0, w, h);

    g.setColor(Color.red);
    for (int i=0; i<w; i+=gap)
        g.drawLine(i, 0, w-i, h);
    for (int i=0; i<h; i+=gap)
        g.drawLine(0, i, w, h-i);

    g.setColor(Color.black);
    g.drawString("Press mouse button to double buffer", 10, h/2);
    // Щелкните кнопкой мыши, чтобы перейти в режим
    // двойной буферизации

    g.setColor(Color.yellow);
    g.fillOval(mx - gap, my - gap, gap*2+1, gap*2+1);

    if (!flicker) {
        screengc.drawImage(buffer, 0, 0, null);
    }
}

public void update(Graphics g) {
    paint(g);
}
}

```

Этот простой пример апплета содержит сложный метод `paint()`. В этом апплете синим цветом окрашивается фон, на котором рисуется красный муар. Далее на фоне этого муара выводится текст черного цвета и вычерчивается желтая окружность, имеющая центр с координатам `mx`, `my`. Методы `mouseMoved()` и `mouseDragged()` переопределяются для отслеживания положения курсора мыши. Эти методы одинаковы, за исключением логической переменной `flicker`. В методе `mouseMoved()` переменной `flicker` присваивается логическое значение `true`, а в методе `mouseDragger()` — логическое значение `false`. Это равнозначно результату вызова метода `repaint()`, когда переменная `flicker` содержит логическое значение `true`, если курсор перемещается без нажатия кнопки мыши, или логическое значение `false`, если курсор мыши перемещается при одновременно нажатой кнопке мыши.

Если метод `paint()` вызывается в тот момент, когда переменная `flicker` содержит логическое значение `true`, то на экране можно наблюдать выполнение каждой операции рисования. Если же был произведен щелчок кнопкой мыши, а метод `paint()` вызывается в тот момент, когда переменная `flicker` содержит логическое значение `false`, то на экране можно наблюдать совсем иную картину. Метод `paint()` заменяет ссылку `g` на класс `Graphics` графическим содержимым внеэкрannого холста `buffer`, созданного в методе `init()`, и далее все операции рисования становятся невидимыми. А в конце метода `paint()` просто вызывается метод `drawImage()` для одновременного отображения результатов выполнения всех методов рисования.

Обратите внимание на то, что теперь методу `drawImage()` можно передавать пустое значение `null` в качестве четвертого параметра. Этот параметр служит для передачи объекта типа `ImageObserver`, который получает извещение о событиях от изображения. В данном случае изображение не формируется из сетевого потока ввода-вывода, и поэтому никаких уведомлений не требуется. На моментальном снимке экрана на рис. 27.2, *слева*, показано, как будет выглядеть результат выполнения данного апплета, если не будет произведен щелчок кнопкой мыши. Как можно заметить, этот снимок был получен именно в тот момент, когда изображение было перерисовано наполовину. А на моментальном снимке экрана на рис. 27.2, *справа*, показано, что при нажатой кнопке мыши изображение оказывается полностью сформированным благодаря двойной буферизации.

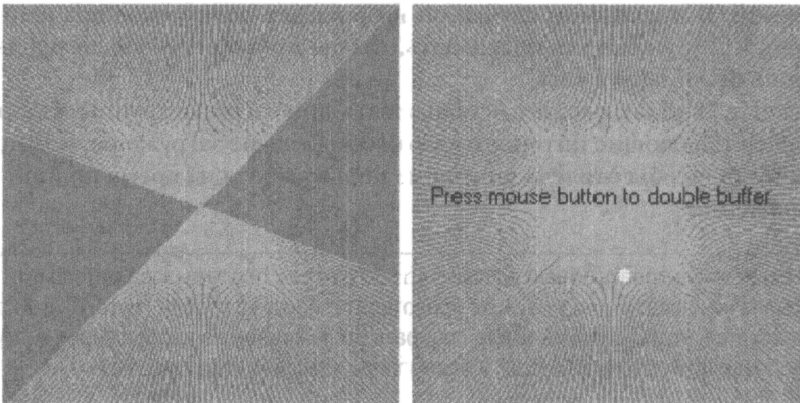


Рис. 27.2. Результат выполнения апплета `DoubleBuffer` без двойной буферизации (слева) и с двойной буферизацией (справа)

Класс `MediaTracker`

Объект класса `MediaTracker` — это объект, который параллельно проверяет состояние произвольного количества изображений. Чтобы использовать класс `MediaTracker`, нужно создать его новый экземпляр и вызвать его метод `addImage()` для наблюдения за состоянием загрузки изображения. Метод `addImage()` имеет следующие общие формы:

```
void addImage(Image объект_изображения, int идентификатор_изображения)
void addImage(Image объект_изображения, int идентификатор_изображения,
               int ширина, int высота)
```

где параметр *объект_изображения* обозначает отслеживаемое изображение. Его идентификатор передается в качестве параметра *идентификатор_изображения*. Идентификаторы не обязательно должны быть однозначными. Один и тот же идентификатор можно использовать для обозначения нескольких изображений как части группы. Кроме того, изображения с меньшими идентификаторами при загрузке имеют приоритет над большими идентификаторами. Во второй форме параметры *ширина* и *высота* определяют размеры объекта при его воспроизведении.

После того как изображение будет зарегистрировано, можно проверить, загружено ли оно, или подождать, пока оно загрузится полностью. Чтобы проверить состояние изображения, следует вызвать метод `checkID()`. В этой главе употребляется следующий вариант данного метода:

```
boolean checkID(int идентификатор_изображения)
```

где параметр *идентификатор_изображения* определяет проверяемый идентификационный номер изображения. Метод `checkID()` возвращает логическое значение `true`, если загружены все изображения, имеющие указанный идентификатор, или же если процесс загрузки был остановлен вследствие ошибки или прерван пользователем. В противном случае он возвращает логическое значение `false`. Метод `checkAll()` можно использовать, чтобы проверить, все ли наблюдаемые изображения были загружены.

Класс `MediaTracker` следует использовать при загрузке группы изображений. Если все представляющие интерес изображения еще не загружены, можно отобразить что-нибудь, чтобы отвлечь внимание пользователя на время полной загрузки всех изображений.

Внимание! Если использовать объект класса `MediaTracker` после вызова метода `addImage()`, то ссылка на класс `MediaTracker` предотвратит процесс сборки "мусора" в системе. Если же требуется, чтобы система могла собирать отслеживаемые изображения в "мусор", следует обеспечить сборку в "мусор" и экземпляра класса `MediaTracker`.

Ниже приведен пример апплета, в котором загружается семь изображений и отображается привлекательная столбиковая диаграмма, иллюстрирующая выполнение загрузки.

```
/*
 * <applet code="TrackedImageLoad" width=300 height=400>
 * <param name="img"
 * value="vincent+leonardo+matisse+picasso+renoir+seurat+vermeer">
 * </applet>
 */
import java.util.*;
import java.applet.*;
import java.awt.*;

public class TrackedImageLoad extends Applet implements Runnable {
    MediaTracker tracker;
```

```

int tracked;
int frame_rate = 5;
int current_img = 0;
Thread motor;
static final int MAXIMAGES = 10;
Image img[] = new Image[MAXIMAGES];
String name[] = new String[MAXIMAGES];
volatile boolean stopFlag;

public void init() {
    tracker = new MediaTracker(this);
    StringTokenizer st =
        new StringTokenizer(getParameter("img"), "+");

    while(st.hasMoreTokens() && tracked <= MAXIMAGES) {
        name[tracked] = st.nextToken();
        img[tracked] =
            getImage(getDocumentBase(), name[tracked] + ".jpg");
        tracker.addImage(img[tracked], tracked);
        tracked++;
    }
}

public void paint(Graphics g) {
    String loaded = "";
    int donecount = 0;

    for(int i=0; i<tracked; i++) {
        if (tracker.checkID(i, true)) {
            donecount++;
            loaded += name[i] + " ";
        }
    }

    Dimension d = getSize();
    int w = d.width;
    int h = d.height;

    if (donecount == tracked) {
        frame_rate = 1;
        Image_i = img[current_img++];
        int iw = i.getWidth(null);
        int ih = i.getHeight(null);
        g.drawImage(i, (w - iw)/2, (h - ih)/2, null);
        if (current_img >= tracked)
            current_img = 0;
    } else {
        int x = w * donecount / tracked;
        g.setColor(Color.black);
        g.fillRect(0, h/3, x, 16);
        g.setColor(Color.white);
        g.fillRect(x, h/3, w-x, 16);
        g.setColor(Color.black);
        g.drawString(loaded, 10, h/2);
    }
}

public void start() {
    motor = new Thread(this);
    stopFlag = false;
    motor.start();
}

public void stop() {

```



```

        stopFlag = true;
    }

    public void run() {
        motor.setPriority(Thread.MIN_PRIORITY);
        while (true) {
            repaint();
            try {
                Thread.sleep(1000/frame_rate);
            } catch (InterruptedException e) {
                System.out.println("Interrupted");
                return;
            }
            if(stopFlag)
                return;
        }
    }
}

```

В данном примере создается новый экземпляр класса `MediaTracker` в методе `init()`, после чего с помощью метода `addImage()` каждое из указанных изображений вводится как отслеживаемое. В методе `paint()` вызывается метод `checkID()` для каждого отслеживаемого изображения. После загрузки все изображения будут выведены на экран. В противном случае отображается простая столбиковая диаграмма, информирующая о количестве загруженных изображений, а под ней выводятся наименования полностью загруженных изображений.

Интерфейс `ImageProducer`

Интерфейс `ImageProducer` предназначен для объектов, которые должны предоставить данные для изображений. Объект класса, реализующего интерфейс `ImageProducer`, задает массив целых чисел или байтов, представляющий данные изображений, и формирует объекты типа `Image`. Как было показано ранее, одна из форм метода `createImage()` получает объект типа `ImageProducer` в качестве своего параметра. Пакет `java.awt.image` содержит два поставщика изображений в виде классов `MemoryImageSource` и `FilteredImageSource`. Ниже будет рассмотрен класс `MemoryImageSource` и показано создание нового объекта типа `Image` на основе данных, сформированных в апплете.

Класс `MemoryImageSource`

Этот класс формирует новое изображение типа `Image` на основе массива данных. В нем определяется несколько конструкторов. Ниже приведен один из конструкторов, который будет использоваться далее в этой главе.

```

MemoryImageSource(int ширина, int высота, int пиксель[],
                  int смещение, int ширина_строки_развертки)

```

Объект класса `MemoryImageSource` формируется на основе массива целых чисел, определяемого параметром `пиксель`, в используемой по умолчанию цветовой модели `RGB` с целью предоставить данные для объекта типа `Image`. В этой цветовой модели пиксель обозначается составным целочисленным значением альфа-ка-

нала, а также каналов красного, зеленого и синего цвета (0xAARRGGBB). Значение альфа-канала обозначает степень прозрачности пикселя. Полностью прозрачному пикселю соответствует нулевое значение, а полностью непрозрачному — значение 255. Значения ширины и высоты готового изображения передаются в качестве параметров *ширина* и *высота*. Исходная точка в массиве пикселей, с которой начнется чтение данных, определяется параметром *смещение*. Ширина строки развертки, которая нередко соответствует ширине изображения, обозначается параметром *ширина_строки_развертки*.

В следующем коротком примере апплета создается объект класса `MemoryImageSource` с помощью разновидности простого алгоритма (логической операции по-разрядное исключающее ИЛИ над координатами *x* и *y* каждого пикселя), взятого из книги *Beyond Photography, The Digital Darkroom* Джерарда Дж. Хольцманна (Gerard J. Holzmnn, Prentice Hall, 1988).

```
/*
 * <applet code="MemoryImageGenerator" width=256 height=256>
 * </applet>
 */
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class MemoryImageGenerator extends Applet {
    Image img;
    public void init() {
        Dimension d = getSize();
        int w = d.width;
        int h = d.height;
        int pixels[] = new int[w * h];
        int i = 0;

        for(int y=0; y<h; y++) {
            for(int x=0; x<w; x++) {
                int r = (x^y)&0xff;
                int g = (x*2^y*2)&0xff;
                int b = (x*4^y*4)&0xff;
                pixels[i++] = (255 << 24) | (r << 16) | (g << 8) | b;
            }
        }
        img = createImage(new MemoryImageSource(w, h, pixels, 0, w));
    }

    public void paint(Graphics g) {
        g.drawImage(img, 0, 0, this);
    }
}
```

Данные для нового объекта типа `MemoryImageSource` создаются в методе `init()`. Массив целых чисел предназначен для хранения значений пикселей; данные создаются во вложенных циклах `for`, где значения *r*, *g* и *b* оказываются смещенными на один пиксель в массиве `pixels`. В конце данного апплета вызывается метод `createImage()` с новым экземпляром класса `MemoryImageSource`, созданным из исходных данных пикселей в качестве его параметра. На рис. 27.3 показано изображение в момент запуска апплета. (В цвете оно выглядит гораздо привлекательнее.)

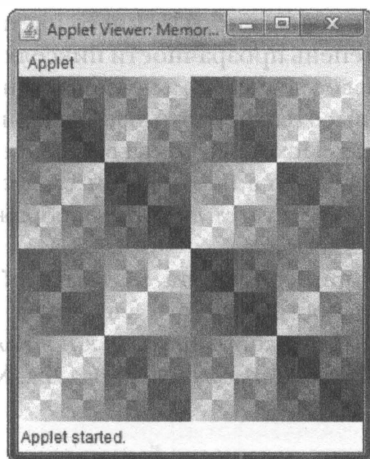


Рис. 27.3. Пример формирования изображения в окне апплета ImageGenerator

Интерфейс ImageConsumer

Интерфейс ImageConsumer предназначен для объектов, которые должны получать данные пикселей из изображений и предоставлять их в качестве другого вида данных. Таким образом, этот интерфейс является прямой противоположностью описанного ранее интерфейса ImageProducer. Объект класса, реализующего интерфейс ImageConsumer, служит для создания массивов типа int или byte, представляющих пиксели из объекта типа Image. Ниже будет рассмотрен класс PixelGrabber, предоставляющий простую реализацию интерфейса ImageConsumer.

Класс PixelGrabber

В пакете java.lang.image определен класс PixelGrabber, который является прямой противоположностью классу MemoryImageSource. Вместо того чтобы формировать изображение из массива значений пикселей, он принимает существующее изображение и захватывает в нем массив пикселей. Чтобы воспользоваться классом PixelGrabber, нужно сначала создать массив целых чисел достаточного размера для хранения в нем данных отдельных пикселей, а затем получить экземпляр класса PixelGrabber, передав его конструктору прямоугольную область, которую необходимо захватить. И наконец, для этого экземпляра следует вызывать метод grabPixels().

Ниже приведен конструктор класса PixelGrabber, употребляемый в этой главе.

```
PixelGrabber(Image объект_изображения, int слева, int сверху,
             int ширина, int высота, int пиксель[], int смещение,
             int ширина_строки_развертки)
```

Здесь параметр *объект_изображения* обозначает тот объект, пиксели которого должны быть захвачены; параметры *слева* и *сверху* – верхний левый угол

прямоугольной области; а параметры *ширина* и *высота* — размеры прямоугольной области, из которой должны быть получены пиксели изображения. Эти пиксели должны храниться в массиве, обозначаемом параметром *пиксель*, начиная с указанного *смещения*. Ширина строки развертки, которая нередко соответствует ширине изображения, обозначается параметром *ширина_строки_развертки*.

Метод `grabPixels()` определяется следующим образом:

```
boolean grabPixels() throws InterruptedException  
boolean grabPixels(long миллисекунды) throws InterruptedException
```

В обеих формах возвращается логическое значение `true` при удачном завершении этого метода, а иначе — логическое значение `false`. Во второй форме параметр *миллисекунды* определяет промежуток времени, в течение которого метод будет ожидать получения пикселей. В обеих формах генерируется исключение типа `InterruptedException`, если выполнение данного метода прерывается другим потоком исполнения.

Ниже приведен пример, в котором производится захват пикселей в изображении с последующим построением гистограммы яркости пикселей. *Гистограмма* представляет собой простой подсчет пикселей, имеющих определенный уровень яркости в пределах от 0 до 255. После того как в окне апплета будет воспроизведено изображение, на его фоне выводится гистограмма. На рис. 27.4 приведен пример вывода гистограммы на фоне изображения в окне апплета `HistoGrab`.

```
/*  
 * <applet code=HistoGrab width=400 height=345>  
 * <param name=img value=Lilies.jpg>  
 * </applet> */  
import java.applet.*;  
import java.awt.* ;  
import java.awt.image.* ;  
public class HistoGrab extends Applet {  
    Dimension d;  
    Image img;  
    int iw, ih;  
    int pixels[];  
    int w, h;  
    int hist[] = new int[256];  
    int max_hist = 0;  
  
    public void init() {  
        d = getSize();  
        w = d.width;  
        h = d.height;  
  
        try {  
            img = getImage(getDocumentBase(), getParameter("img"));  
            MediaTracker t = new MediaTracker(this);  
  
            t.addImage(img, 0);  
            t.waitForID(0);  
            iw = img.getWidth(null);  
            ih = img.getHeight(null);  
            pixels = new int[iw * ih];  
            PixelGrabber pg = new PixelGrabber(img, 0, 0, iw, ih,  
                                                pixels, 0, iw);  
  
            pg.grabPixels();  
        } catch (InterruptedException e) {  
            System.out.println("InterruptedException");  
        }  
    }  
}
```

```

    return;
}

for (int i=0; i<iw*ih; i++) {
    int p = pixels[i];
    int r = 0xff & (p >> 16);
    int g = 0xff & (p >> 8);
    int b = 0xff & (p);
    int y = (int) (.33 * r + .56 * g + .11 * b);
    hist[y]++;
}
for (int i=0; i<256; i++) {
    if (hist[i] > max_hist)
        max_hist = hist[i];
}

public void update() {}

public void paint(Graphics g) {
    g.drawImage(img, 0, 0, null);
    int x = (w - 256) / 2;
    int lasty = h - h * hist[0] / max_hist;
    for (int i=0; i<256; i++, x++) {
        int y = h - h * hist[i] / max_hist;
        g.setColor(new Color(i, i, i));
        g.fillRect(x, y, 1, h);
        g.setColor(Color.red);
        g.drawLine(x-1, lasty, x, y);
        lasty = y;
    }
}
}

```

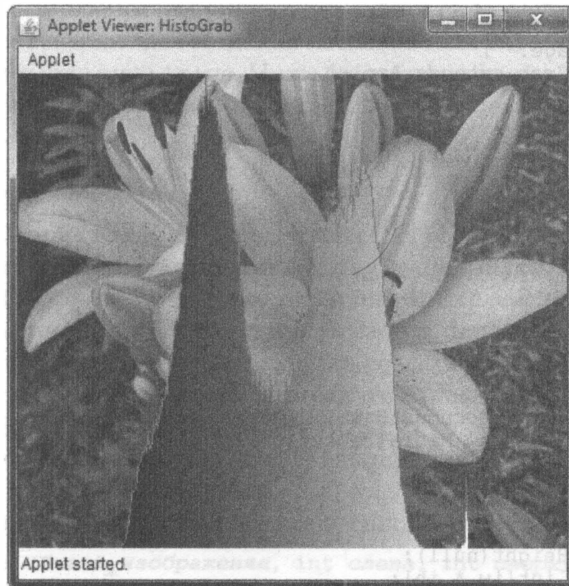


Рис. 27.4. Пример вывода гистограммы на фоне изображения в окне апплета HistoGrab

Класс ImageFilter

При наличии пары интерфейсов ImageProducer и ImageConsumer, а также реализующих их конкретных классов MemoryImageSource и PixelGrabber можно создать произвольный набор фильтров преобразования, которые будут принимать исходные пиксели изображения, видоизменять их и передавать некоторому потребителю. Этот механизм аналогичен механизму создания определенных классов на основе абстрактных классов потоков ввода-вывода InputStream, OutputStream, Reader и Writer, рассматривавшихся в главе 20. Такая модель потоков ввода-вывода изображений завершается внедрением класса ImageFilter.

Пакет java.awt.image содержит подклассы, производные от класса ImageFilter. К их числу относятся классы AreaAveragingScaleFilter, CropImageFilter, ReplicateScaleFilter и RGBImageFilter. Имеется также реализация интерфейса ImageProducer в классе FilteredImageSource, который принимает произвольный класс ImageFilter и заключает его в оболочку интерфейса ImageProducer для фильтрации формируемых им пикселей. Таким образом, экземпляр класса FilteredImageSource можно использовать в качестве экземпляра интерфейса ImageProducer при вызове метода createImage() почти так же, как и экземпляр типа BufferedInputStream в качестве потока ввода типа InputStream. Далее в этой главе будут рассмотрены два класса фильтров — CropImageFilter и RGBImageFilter.

Класс CropImageFilter

Фильтр класса CropImageFilter выполняет фильтрацию исходного изображения для извлечения прямоугольной области. Этот фильтр удобен, например, для обработки нескольких мелких изображений, сформированных из одного крупного исходного изображения. Для загрузки двадцати изображений размером 2 Кбайт потребуется больше времени, чем для загрузки одного изображения размером 40 Кбайт, составленного из многих кадров анимации. Если каждое составное изображение имеет один и тот же размер, то их можно без особого труда извлечь с помощью фильтра типа CropImageFilter, расчленив весь блок в самом начале прикладной программы. Ниже приведен пример формирования 16 изображений из одного крупного изображения. Элементы мозаики затем перетасовываются 32 раза заменой случайной пары, взятой из 16 изображений. На рис. 27.5 приведено мозаичное изображение, составленное из случайно выбранных фрагментов исходного изображения цветков в окне апплета TileImage.

```
/*
 * <applet code=TileImage width=400 height=345>
 * <param name=img value=Lilies.jpg>
 * </applet>
 */
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class TileImage extends Applet {
    Image img;
    Image cell[] = new Image[4*4];
```

```

int iw, ih;
int tw, th;

public void init() {
    try {
        img = getImage(getDocumentBase(), getParameter("img"));
        MediaTracker t = new MediaTracker(this);
        t.addImage(img, 0);
        t.waitForID(0);
        iw = img.getWidth(null);
        ih = img.getHeight(null);
        tw = iw / 4;
        th = ih / 4;
        CropImageFilter f;
        FilteredImageSource fis;
        t = new MediaTracker(this);
        for (int y=0; y<4; y++) {
            for (int x=0; x<4; x++) {
                f = new CropImageFilter(tw*x, th*y, tw, th);
                fis = new FilteredImageSource(img.getSource(), f);
                int i = y*4+x;
                cell[i] = createImage(fis);
                t.addImage(cell[i], i);
            }
        }
        t.waitForAll();
        for (int i=0; i<32; i++) {
            int si = (int)(Math.random() * 16);
            int di = (int)(Math.random() * 16);
            Image tmp = cell[si];
            cell[si] = cell[di];
            cell[di] = tmp;
        }
    } catch (InterruptedException e) {
        System.out.println("Interrupted");
    }
}

public void update(Graphics g) {
    paint(g);
}

public void paint(Graphics g) {
    for (int y=0; y<4; y++) {
        for (int x=0; x<4; x++) {
            g.drawImage(cell[y*4+x], x * tw, y * th, null);
        }
    }
}
}

```

Фильтр класса RGBImageFilter

Фильтр класса `RGBImageFilter` служит для попиксельного формирования одного изображения из другого вместе с преобразованием цветов. Этот фильтр можно применить для осветления изображения, повышения его контрастности и даже его преобразования в полутоновое изображение.

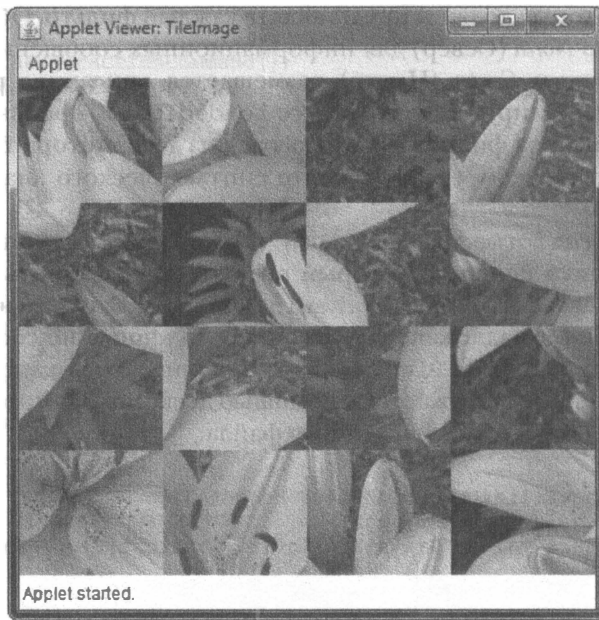


Рис. 27.5. Мозаичное изображение, составленное в окне апплета `TileImage`

Чтобы продемонстрировать применение фильтра класса `RGBImageFilter`, рассмотрим более сложный пример внедрения методики динамического подключения фильтров для обработки изображений. Для обобщения процесса фильтрации изображения служит специальный интерфейс. Это позволяет загружать подключаемые фильтры в апплет, исходя из дескрипторов `<param>`, но не имея при этом предварительной информации о каждом фильтре типа `ImageFilter`. Данный пример апплета состоит из главного класса `ImageFilterDemo`, интерфейса `PlugInFilter` и служебного класса `LoadedImage`, инкапсулирующего некоторые методы из класса `MediaTracker`, упоминаемые в этой главе. В данном примере используются также три класса фильтров, `Grayscale`, `Invert` и `Contrast`, которые просто манипулируют цветовым пространством исходного изображения с помощью фильтров класса `RGBImageFilter`, а также два дополнительных класса, `Blur` и `Sharpen`, позволяющих реализовать более сложные фильтры свертки, изменяющие значения отдельных пикселей, исходя из окружающих их пикселей в исходных данных изображения. Классы `Blur` и `Sharpen` являются производными от абстрактного вспомогательного класса `Convolver`. Рассмотрим данный пример по частям в силу его сложности.

Класс `ImageFilterDemo`

Этот класс служит каркасом для апплетов в рассматриваемом здесь примере применения фильтров изображений. В нем используется диспетчер граничной компоновки типа `BorderLayout` и панель типа `Panel` на позиции *South* (`Юг`) для разме-

щения кнопок, которые должны представлять каждый фильтр. Объект типа `Label` занимает позицию *North* (Север) для информационных сообщений о выполнении фильтра. На позиции *Center* (Центр) размещается описанное ранее изображение, которое инкапсулируется в подклассе `LoadedImage`, производном от класса `Canvas`. Кнопки выбора фильтров извлекаются из дескриптора `filters<param>`, где они разделены знаком `+`, в результате синтаксического анализа средствами класса `StringTokenizer`.

Метод `actionPerformed()` интересен тем, что метка кнопки используется в нем как имя класса фильтра, который он пытается загрузить, вызывая метод `(PlugInFilter)Class.forName(a).newInstance()`. Это надежный метод, поскольку он выполняет надлежащее действие, если кнопка не соответствует классу, реализующему интерфейс `PlugInFilter`. На рис. 27.6 показано, как выглядит апплет, когда он впервые загружается с помощью дескриптора `<applet>`, указанно-го в начале приведенного ниже исходного файла.

```
/*
 * <applet code=ImageFilterDemo width=400 height=345>
 * <param name=img value=vincent.jpg>
 * <param name=filters value="Grayscale+Invert+Contrast+Blur+ Sharpen">
 * </applet>
 */
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class ImageFilterDemo extends Applet implements ActionListener {
    Image img;
    PlugInFilter pif;
    Image fimg;
    Image curImg;
    LoadedImage lim;
    Label lab;
    Button reset;

    public void init() {
        setLayout(new BorderLayout());
        Panel p = new Panel();
        add(p, BorderLayout.SOUTH);
        reset = new Button("Reset");
        reset.addActionListener(this);
        p.add(reset);
        StringTokenizer st = new StringTokenizer(getParameter("filters"),
                                                "+");
        while(st.hasMoreTokens()) {
            Button b = new Button(st.nextToken());
            b.addActionListener(this);
            p.add(b);
        }
        lab = new Label("");
        add(lab, BorderLayout.NORTH);
        img = getImage(getDocumentBase(), getParameter("img"));
        lim = new LoadedImage(img);
        add(lim, BorderLayout.CENTER);
    }

    public void actionPerformed(ActionEvent ae) {
```

```

String a = "";
try {
    a = ae.getActionCommand();
    if (a.equals("Reset")) {
        lim.set(img);
        lab.setText("Normal");
    }
    else {
        pif = (PlugInFilter) Class.forName(a).newInstance();
        fimg = pif.filter(this, img);
        lim.set(fimg);
        lab.setText("Filtered: " + a);
    }
    repaint();
} catch (ClassNotFoundException e) {
    lab.setText(a + " not found");
    lim.set(img);
    repaint();
} catch (InstantiationException e) {
    lab.setText("couldn't new " + a);
} catch (IllegalAccessException e) {
    lab.setText("no access: " + a);
}
}
}

```

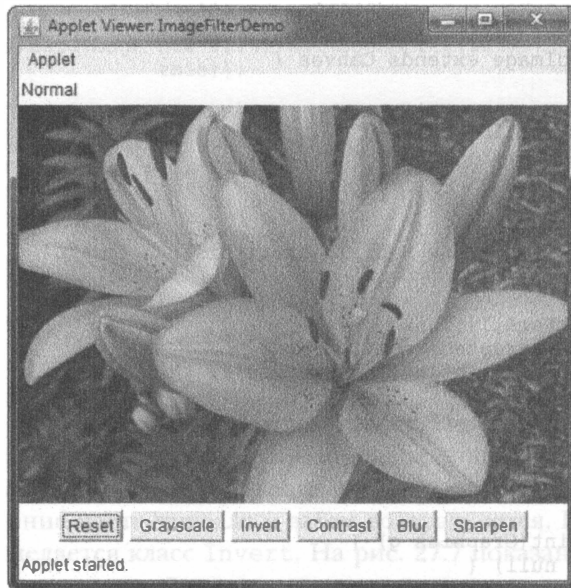


Рис. 27.6. Пример воспроизведения в обычном режиме изображения в окне апплета ImageFilterDemo

Интерфейс PlugInFilter

Для абстракции процесса фильтрации изображений служит интерфейс PlugInFilter, объявление которого приведено ниже. В нем определяется единственный

метод `filter()`, принимающий в качестве параметров апплет и исходное изображение и возвращающий новое отфильтрованное некоторым образом изображение.

```
interface PlugInFilter {
    java.awt.Image filter(java.applet.Applet a, java.awt.Image in);
}
```

Класс `LoadedImage`

Это служебный подкласс, производный от класса `Canvas`. При построении объекта конструктор этого класса принимает в качестве параметра изображение и синхронно загружает его с помощью класса `MediaTracker`. Правильное поведение класса `LoadedImage` в элементе управления типа `LayoutControl` обеспечивается благодаря тому, что в нем переопределяются методы `getPreferredSize()` и `getMinimumSize()`. В этом классе имеется также метод `set()`, позволяющий задать новое изображение типа `Image`, которое требуется воспроизвести на данном холсте типа `Canvas`. Именно таким образом отфильтрованное изображение воспроизводится по завершении его обработки подключаемым фильтром. Ниже показано, каким образом определяется класс `LoadedImage`.

```
import java.awt.*;

public class LoadedImage extends Canvas {
    Image img;

    public LoadedImage(Image i) {
        set(i);
    }

    void set(Image i) {
        MediaTracker mt = new MediaTracker(this);
        mt.addImage(i, 0);
        try {
            mt.waitForAll();
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
            return;
        }
        img = i;
        repaint();
    }

    public void paint(Graphics g) {
        if (img == null) {
            g.drawString("no image", 10, 30);
        } else {
            g.drawImage(img, 0, 0, this);
        }
    }

    public Dimension getPreferredSize() {
        return new Dimension(img.getWidth(this), img.getHeight(this));
    }

    public Dimension getMinimumSize() {
```

```

        return getPreferredSize();
    }
}

```

Класс Grayscale

Фильтр класса Grayscale является производным от класса RGBImageFilter. Это означает, что объект класса Grayscale можно передавать в качестве параметра типа ImageFilter конструктору класса FilteredImageSource. Нужно лишь переопределить метод filterRGB(), чтобы изменить исходные значения цвета. Этот метод принимает в качестве параметра значение RGB красной, зеленой и синей составляющих цвета и вычисляет яркость пикселя, используя коэффициент преобразования цвета в яркость по американскому стандарту NTSC (National Television Standards Committee — Национальный комитет по телевизионным стандартам). Затем он просто возвращает серый пиксель, имеющий такую же яркость, как и у исходного цвета. Ниже показано, каким образом определяется класс Grayscale.

```

import java.applet.*;
import java.awt.*;
import java.awt.image.*;

class Grayscale extends RGBImageFilter implements PlugInFilter {
    public Image filter(Applet a, Image in) {
        return a.createImage(new FilteredImageSource(in.getSource(),
            this));
    }

    public int filterRGB(int x, int y, int rgb) {
        int r = (rgb >> 16) & 0xff;
        int g = (rgb >> 8) & 0xff;
        int b = rgb & 0xff;
        int k = (int) (.56 * g + .33 * r + .11 * b);
        return (0xff000000 | k << 16 | k << 8 | k);
    }
}

```

Класс Invert

Фильтр класса Invert также достаточно прост. Он разделяет сначала каналы красного, зеленого и синего цвета, а затем обращает значения их цвета, вычитая их из значения 255. Обращенные значения отдельных каналов цвета составляют обратно в значение цвета пикселя и затем возвращаются. Ниже показано, каким образом определяется класс Invert. На рис. 27.7 показано изображение после обработки фильтром типа Invert.

```

import java.applet.*;
import java.awt.*;
import java.awt.image.*;

class Invert extends RGBImageFilter implements PlugInFilter {
    public Image filter(Applet a, Image in) {
        return a.createImage(new FilteredImageSource(in.getSource(),
            this));
    }

    public int filterRGB(int x, int y, int rgb) {
        int r = 0xff - (rgb >> 16) & 0xff;

```

```

        int g = 0xff - (rgb >> 8) & 0xff;
        int b = 0xff - rgb & 0xff;
        return (0xff000000 | r << 16 | g << 8 | b);
    }
}

```

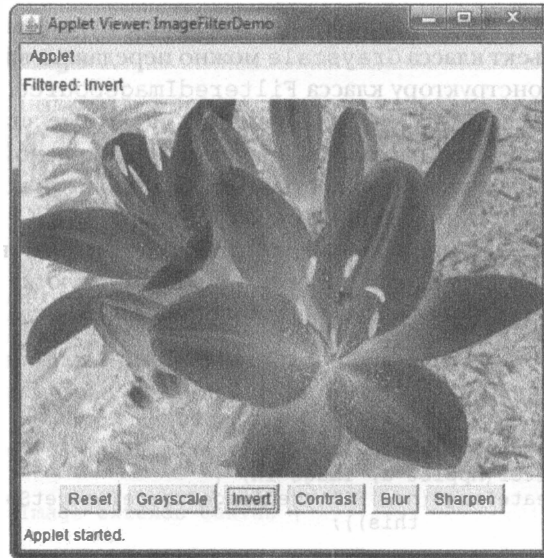


Рис. 27.7. Пример обработки изображения фильтром типа Invert в окне апплета ImageFilterDemo

Класс Contrast

Фильтр класса Contrast очень похож на фильтр типа Grayscale, за исключением того, что метод `filterRGB()` переопределяется в нем немного сложнее. В этом методе применяется алгоритм, по которому для повышения контрастности значения красной, зеленой и синей составляющих цвета умножаются по отдельности на коэффициент 1,2, если уровень их яркости выше 128, или же делятся на коэффициент 1,2, если уровень их яркости ниже 128. Измененные значения яркости ограничиваются в пределах от 0 до 255 в методе `multiclamp()`. Ниже показано, каким образом определяется класс Contrast. На рис. 27.8 показано изображение после обработки фильтром типа Contrast.

```

import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class Contrast extends RGBImageFilter implements PlugInFilter {

    public Image filter(Applet a, Image in) {
        return a.createImage(new FilteredImageSource(in.getSource(), this));
    }

    private int multiclamp(int in, double factor) {
        in = (int) (in * factor);
    }
}

```

```

        return in > 255 ? 255 : in;
    }

    double gain = 1.2;
    private int cont(int in) {
        return (in < 128) ? (int)(in/gain) : multclamp(in, gain);
    }

    public int filterRGB(int x, int y, int rgb) {
        int r = cont((rgb >> 16) & 0xff);
        int g = cont((rgb >> 8) & 0xff);
        int b = cont(rgb & 0xff);
        return (0xff000000 | r << 16 | g << 8 | b);
    }
}

```

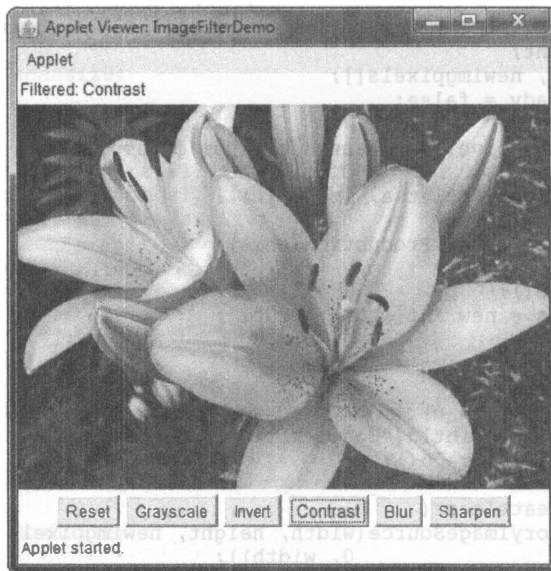


Рис. 27.8. Пример обработки изображения фильтром типа Contrast в окне апплета ImageFilterDemo

Класс Convolver

Абстрактный класс `Convolver` выполняет основные функции фильтра свертки, реализуя интерфейс `ImageConsumer` для переноса пикселей исходного изображения в массив `imgpixels`. А для отфильтрованных данных изображения он создает второй массив `newimgpixels`. Фильтры свертки выбирают вокруг каждого пикселя в изображении небольшую прямоугольную область пикселей, называемую *ядром свертки*. В данном примере выбирается прямоугольная область размерами 3×3 , чтобы принять решение, каким образом следует изменить центральный пиксель в этой области.

На заметку! Фильтр не может сразу видоизменить массив `imgpixels`, поскольку в следующем пикселе в строке развертки может быть предпринята попытка использовать исходное значение из предыдущего пикселя, которое может быть уже отфильтровано.

В обоих рассматриваемых далее конкретных подклассах фильтров представлена довольно простая реализация метода `convolver()`, в которой массив `imgpixels` используется для хранения исходных данных изображения, а массив `newimgpixels` — для хранения результатов его фильтрации. Ниже показано, каким образом определяется класс `Convolver`.

```
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

abstract class Convolver implements ImageConsumer, PlugInFilter {
    int width, height;
    int imgpixels[], newimgpixels[];
    boolean imageReady = false;

    abstract void convolve(); // здесь следует фильтр...

    public Image filter(Applet a, Image in) {
        imageReady = false;
        in.getSource().startProduction(this);

        waitForImage();
        newimgpixels = new int[width*height];

        try {
            convolve();
        } catch (Exception e) {
            System.out.println("Convolver failed: " + e);
            e.printStackTrace();
        }
        return a.createImage(
            new MemoryImageSource(width, height, newimgpixels,
                                  0, width));
    }

    synchronized void waitForImage() {
        try {
            while(!imageReady) wait();
        } catch (Exception e) {
            System.out.println("Interrupted");
        }
    }

    public void setProperties(java.util.Hashtable<?,?> dummy) { }
    public void setColorModel(ColorModel dummy) { }
    public void setHints(int dummy) { }

    public synchronized void imageComplete(int dummy) {
        imageReady = true;
        notifyAll();
    }

    public void setDimensions(int x, int y) {
        width = x;
        height = y;
    }
}
```

```

        imgpixels = new int[x*y];
    }

    public void setPixels(int x1, int y1, int w, int h,
        ColorModel model, byte pixels[], int off, int scansize) {
        int pix, x, y, x2, y2, sx, sy;

        x2 = x1+w;
        y2 = y1+h;
        sy = off;
        for(y=y1; y<y2; y++) {
            sx = sy;
            for(x=x1; x<x2; x++) {
                pix = model.getRGB(pixels[sx++]);
                if((pix & 0xff000000) == 0)
                    pix = 0x00ffffff;
                imgpixels[y*width+x] = pix;
            }
            sy += scansize;
        }
    }

    public void setPixels(int x1, int y1, int w, int h,
        ColorModel model, int pixels[], int off, int scansize) {
        int pix, x, y, x2, y2, sx, sy;

        x2 = x1+w;
        y2 = y1+h;
        sy = off;
        for(y=y1; y<y2; y++) {
            sx = sy;
            for(x=x1; x<x2; x++) {
                pix = model.getRGB(pixels[sx++]);
                if((pix & 0xff000000) == 0)
                    pix = 0x00ffffff;
                imgpixels[y*width+x] = pix;
            }
            sy += scansize;
        }
    }
}

```

На заметку! В пакете `java.awt.image` предоставляется встроенный класс `ConvolveOp` фильтра свертки. Исследуйте его функциональные возможности самостоятельно.

Класс Blur

Фильтр класса `Blur` является производным от класса `Convolver`. Он обрабатывает каждый пиксель исходного изображения из массива `imgpixels`, вычисляя среднее значение цвета в окружающей этот пиксель прямоугольной области размерами 3×3 . Вычисленное среднее значение цвета соответствующего пикселя выходного изображения сохраняется в массиве `newimgpixels`. Ниже показано, каким образом определяется класс `Blur`. На рис. 27.9 показано изображение после обработки фильтром типа `Blur`.

```

public class Blur extends Convolver {
    public void convolve() {
        for(int y=1; y<height-1; y++) {

```



```

for(int x=1; x<width-1; x++) {
    int rs = 0;
    int gs = 0;
    int bs = 0;

    for(int k=-1; k<=1; k++) {
        for(int j=-1; j<=1; j++) {
            int rgb = imgpixels[(y+k)*width+x+j];
            int r = (rgb >> 16) & 0xff;
            int g = (rgb >> 8) & 0xff;
            int b = rgb & 0xff;
            rs += r;
            gs += g;
            bs += b;
        }
    }

    rs /= 9;
    gs /= 9;
    bs /= 9;

    newimgpixels[y*width+x] = (0xff000000 |
                                rs << 16 | gs << 8 | bs);
}
}
}

```

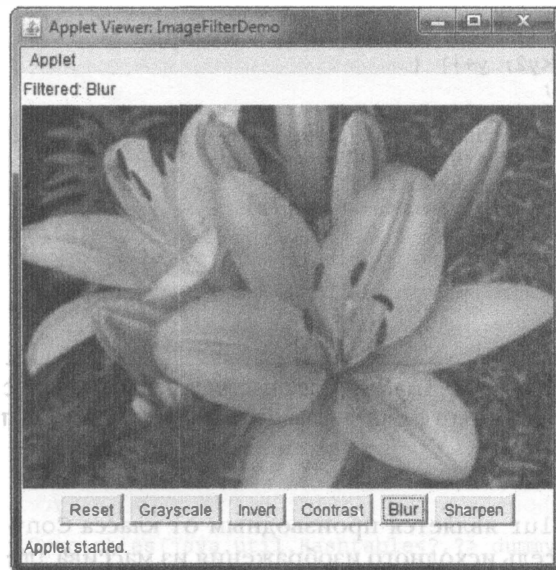


Рис. 27.9. Пример обработки изображения фильтром типа Blur в окне апплета ImageFilterDemo

Класс Sharpen

Фильтр класса Sharpen также является производным от класса Convolver и в какой-то степени противоположностью фильтру класса Blur. Он обрабатывает каждый пиксель исходного изображения из массива `imgpixels` и вычисляет

среднее значение цвета в окружающей этот пиксель прямоугольной области размерами 3×3 , не считая центральный пиксель. Значение цвета соответствующего пикселя выходного изображения получается следующим образом: сначала определяется разность значения центрального пикселя и вычисленного среднего значения окружающих пикселей, а затем эта разность складывается со значением цвета центрального пикселя и полученный результат сохраняется в массиве `newimgpixels`. По существу, если пиксель оказывается ярче на 30 единиц, чем окружающие его пиксели, то он становится еще ярче на 30 единиц. А если пиксель оказывается темнее на 10 единиц, то он становится еще темнее на 10. В итоге резкие края изображения становятся еще более резкими, а плавные участки изображения остаются без изменения. Ниже показано, каким образом определяется класс `Sharpen`. На рис. 27.10 показано изображение после обработки фильтром типа `Sharpen`.

```
public class Sharpen extends Convolver {

    private final int clamp(int c) {
        return (c > 255 ? 255 : (c < 0 ? 0 : c));
    }

    public void convolve() {
        int r0=0, g0=0, b0=0;
        for(int y=1; y<height-1; y++) {
            for(int x=1; x<width-1; x++) {
                int rs = 0;
                int gs = 0;
                int bs = 0;

                for(int k=-1; k<=1; k++) {
                    for(int j=-1; j<=1; j++) {
                        int rgb = imgpixels[(y+k)*width+x+j];
                        int r = (rgb >> 16) & 0xff;
                        int g = (rgb >> 8) & 0xff;
                        int b = rgb & 0xff;
                        if (j == 0 && k == 0) {
                            r0 = r;
                            g0 = g;
                            b0 = b;
                        } else {
                            rs += r;
                            gs += g;
                            bs += b;
                        }
                    }
                }

                rs >>= 3;
                gs >>= 3;
                bs >>= 3;
                newimgpixels[y*width+x] = (0xff000000 |
                                                clamp(r0+r0-rs) << 16 |
                                                clamp(g0+g0-gs) << 8 |
                                                clamp(b0+b0-bs));
            }
        }
    }
}
```

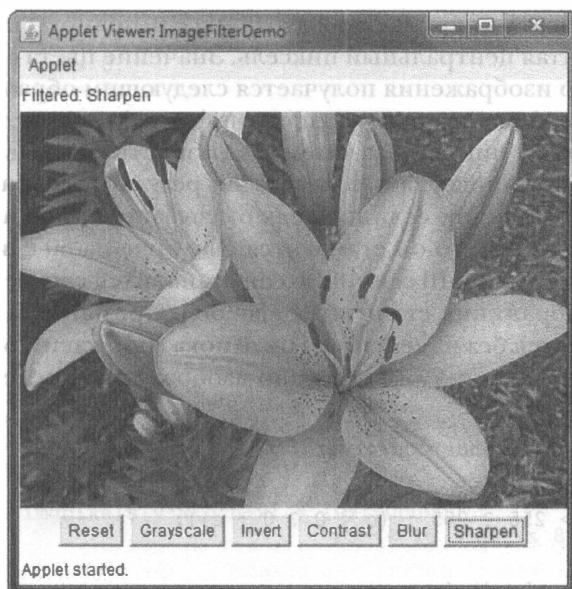


Рис. 27.10. Пример обработки изображения фильтром типа Sharpen в окне апплета ImageFilterDemo

Дополнительные классы для формирования изображений

Помимо описанных в этой главе классов для формирования изображений, в пакете `java.awt.image` содержится ряд других классов, обладающих расширенными возможностями для управления процессом формирования изображений на основе усовершенствованных методик. Для формирования изображений имеется также пакет `javax.imageio`, поддерживающий подключаемые модули для обработки изображений в разных форматах. Если вас интересуют возможности изощренного вывода графики, вам придется освоить дополнительные классы, доступные в пакетах `javax.awt.image` и `javax.imageio`.

Утилиты параллелизма

В языке Java с самого начала была предусмотрена встроенная поддержка многопоточности и синхронизации. Например, новые потоки исполнения можно создавать, реализуя интерфейс `Runnable` или расширяя класс `Thread`. Синхронизация потоков исполнения осуществляется с помощью ключевого слова `synchronized`, взаимодействие потоков исполнения обеспечивается методами `wait()` и `notify()`, определенными в классе `Object`. В свое время встроенная поддержка многопоточности стала одним из наиболее важных нововведений в языке Java и до сих пор остается одной из самых сильных его сторон.

Но какой бы концептуально безупречной ни была первоначальная поддержка многопоточности в Java, она не является идеальной для всех приложений, особенно для тех, где широко применяются многие потоки исполнения. В частности, первоначальная поддержка многопоточности лишена некоторых высокоуровневых средств, в том числе семафоров, пулов потоков исполнения и диспетчеров, которые способствуют созданию программ, интенсивно работающих в параллельном режиме.

Следует сразу же пояснить, что многопоточность применяется во многих программах на Java, которые в конечном итоге становятся параллельными. Например, многопоточность применяется во многих апплетах и сервлетах. Но в этой главе под *параллельной* подразумевается такая программа, которая в *полной мере* использует параллельно выполняющиеся потоки как ее *неотъемлемую часть*. Примером тому служит программа, в которой отдельные потоки исполнения служат для одновременного вычисления частичных результатов более крупных расчетов. Другим примером служит программа, координирующая активность нескольких потоков исполнения, каждый из которых пытается обратиться к информации, хранящейся в базе данных. В этом случае доступ в данным только для чтения может обрабатываться отдельно от доступа, требующего не только для чтения, но и записи данных.

Сначала для поддержки параллельных программ в версии JDK 5 были внедрены *утилиты параллелизма*, которые зачастую называют также *параллельным API* (*прикладным программным интерфейсом*). Первоначальный набор утилит параллелизма предоставлял немало возможностей, о которых уже давно мечтали программисты, занимающиеся разработкой параллельных прикладных программ. В частности, эти утилиты предоставляют такие синхронизаторы вроде семафоров, пулы потоков исполнения, диспетчеры, блокировки, несколько параллельных коллекций, а также рациональное применение потоков исполнения для получения результатов вычислений.

Несмотря на то что первоначальные возможности параллельного API были сами по себе впечатляющими, они, тем не менее, были значительно расширены в версии JDK 7. Самым важным дополнением явился каркас *Fork/Join Framework*, упростивший создание программ, использующих несколько процессоров (в многоядерных системах). Таким образом, этот каркас упростил разработку программ, две или больше частей которых выполняются одновременно на самом деле, а не только путем квантования времени. Нетрудно представить, что параллельное выполнение может существенно ускорить некоторые операции. Многоядерные системы уже стали нормой, и поэтому внедрение каркаса *Fork/Join Framework* оказалось не только своевременной, но и эффективной мерой. А в версии JDK 8 каркас *Fork/Join Framework* претерпел дальнейшие усовершенствования.

Кроме того, в JDK 8 были внедрены новые средства, связанные с другими частями параллельного API. Это означает, что параллельный API продолжает развиваться и расширяться, чтобы удовлетворить насущные потребности современной вычислительной среды.

Параллельный API был довольно крупным с самого начала, а дополнения в версиях JDK 7 и JDK 8 сделали его еще крупнее. Как и следовало ожидать, большинство вопросов применения утилит параллелизма довольно сложны, поэтому исчерпывающее их обсуждение выходит за рамки данной книги. Тем не менее всем программирующим на Java следует иметь хотя бы общее представление о параллельном API и некоторые практические навыки его применения. Даже в тех программах, где параллелизм не используется интенсивно, такие средства, как синхронизаторы, вызываемые потоки исполнения и исполнители, применимы в самых разных случаях. Но важнее всего, вероятно, следующее обстоятельство: в связи с широким распространением многоядерных вычислительных систем все чаще появляются решения, в которых задействован каркас *Fork/Join Framework*. По указанным выше причинам в этой главе дается краткий обзор утилит параллелизма и демонстрируется ряд примеров их применения. А завершается глава введением в каркас *Fork/Join Framework*.

Пакеты параллельного API

Утилиты параллелизма входят в состав пакета `java.util.concurrent` и двух его подпакетов — `java.util.concurrent.atomic` и `java.util.concurrent.locks`. Далее следует краткий обзор их содержимого.

Пакет `java.util.concurrent`

В пакете `java.util.concurrent` определяются основные функциональные возможности, которые поддерживают альтернативные варианты встроенных способов синхронизации и взаимодействия между потоками исполнения. В этом пакете определяются следующие основные средства параллелизма.

- Синхронизаторы.
- Исполнители.
- Параллельные коллекции.
- Каркас *Fork/Join Framework*.

Синхронизаторы предоставляют высокоуровневые способы синхронизации взаимодействия нескольких потоков. В пакете `java.util.concurrent` определен ряд классов *синхронизаторов*, перечисленных в табл. 28.1.

Таблица 28.1. Классы синхронизаторов, определенные в пакете `java.util.concurrent`

Класс	Описание
Semaphore	Реализует классический семафор
CountDownLatch	Ожидает до тех пор, пока не произойдет определенное количество событий
CyclicBarrier	Позволяет группе потоков исполнения войти в режим ожидания в предварительно заданной точке выполнения
Exchanger	Осуществляет обмен данными между двумя потоками исполнения
Phaser	Синхронизирует потоки исполнения, проходящие через несколько фаз операции

Следует иметь в виду, что каждый синхронизатор предоставляет конкретное решение задачи синхронизации. Благодаря этому можно оптимизировать работу каждого синхронизатора. Раньше подобные типы объектов синхронизации необходимо было создавать вручную. Параллельный API стандартизирует их и делает доступными для всех программирующих на Java.

Исполнители управляют исполнением потоков. На вершине иерархии исполнителей находится интерфейс `Executor`, предназначенный для запуска потока исполнения. Интерфейс `ExecutorService` расширяет интерфейс `Executor` и предоставляет методы, управляющие исполнением. Имеются следующие три реализации интерфейса `ExecutorService`: классы `ThreadPoolExecutor`, `ScheduledThreadPoolExecutor` и класс `ForkJoinPool`. В пакете `java.util.concurrent` определяется также служебный класс `Executors`, содержащий несколько статических методов, упрощающих создание различных исполнителей.

С исполнителями связаны также интерфейсы `Future` и `Callable`. Интерфейс `Future` содержит значение, возвращаемое потоком после исполнения. Таким образом, это значение определяется “на будущее”, когда поток завершит свое исполнение. Интерфейс `Callable` определяет поток исполнения, возвращающий значение.

В пакете `java.util.concurrent` определяется ряд классов параллельных коллекций, в том числе `ConcurrentHashMap`, `ConcurrentLinkedQueue` и `CopyOnWriteArrayList`. Они представляют параллельные альтернативные варианты для связанных с ними классов, определенных в каркасе коллекций `Collections Framework`.

В каркасе `Fork/Join Framework` поддерживается параллельное программирование. К числу основных в этом каркасе относятся классы `ForkJoinTask`, `ForkJoinPool`, `RecursiveTask` и `RecursiveAction`. И наконец, для усовершенствованной синхронизации потоков исполнения в пакете `java.util.concurrent` определяется перечисление `TimeUnit`.

Пакет `java.util.concurrent.atomic`

Средства, предоставляемые в этом пакете, упрощают применение переменных в параллельной среде. Эти средства эффективно обновляют значения переменных без применения блокировок. Для этой цели служат такие классы, как `AtomicInteger` и `AtomicLong`, а также методы наподобие `compareAndSet()`, `decrementAndGet()` и `getAndSet()`. Эти методы действуют в режиме одной непрерывно выполняемой операции.

Пакет `java.util.concurrent.locks`

В этом пакете предоставляется альтернатива применению синхронизированных методов. В его основу положен интерфейс `Lock`, определяющий основной механизм, применяемый для доступа к объекту и отказа в доступе. К основным методам из этого пакета относятся `lock()`, `tryLock()` и `unlock()`. Преимущество этих методов заключается в том, что они расширяют возможности управления синхронизацией. Далее в этой главе подробно рассматриваются отдельные компоненты параллельного API.

Применение объектов синхронизации

Объекты синхронизации представлены классами `Semaphore`, `CountDownLatch`, `CyclicBarrier`, `Exchanger` и `Phaser`. Совместно они позволяют без особого труда решать некоторые задачи синхронизации, справиться с которыми ранее было совсем не просто. Их можно также применять в широком ряде программ — даже в тех, где поддерживается только ограниченный параллелизм. Объекты синхронизации могут встречаться практически во всех программах на Java, поэтому остановимся на них более подробно.

Класс `Semaphore`

Первым сразу же распознаваемым среди объектов синхронизации является семафор, реализуемый в классе `Semaphore`. *Семафор* управляет доступом к общему ресурсу с помощью счетчика. Если счетчик больше нуля, доступ разрешается, а если он равен нулю, то в доступе будет отказано. В действительности этот счетчик подсчитывает *разрешения*, открывающие доступ к общему ресурсу. Следовательно, чтобы получить доступ к ресурсу, поток исполнения должен получить у семафора разрешение на доступ.

Как правило, поток исполнения, которому требуется доступ к общему ресурсу, пытается получить разрешение, чтобы воспользоваться семафором. Если значение счетчика семафора окажется больше нуля, поток исполнения получит разрешение, после чего значение счетчика семафора уменьшается на единицу. В противном случае поток будет заблокирован до тех пор, пока он не сумеет получить разрешение. Если потоку исполнения доступ к общему ресурсу больше не нужен, он освобождает разрешение, в результате чего значение счетчика семафора уве-

личивается на единицу. Если в это время другой поток исполнения ожидает разрешения, то он сразу же его получает. В Java этот механизм реализуется в классе Semaphore.

В классе Semaphore имеются два приведенных ниже конструктора:

```
Semaphore(int число)
Semaphore(int число, boolean способ)
```

Здесь параметр *число* обозначает исходное значение счетчика разрешений. Таким образом, параметр *число* определяет количество потоков исполнения, которым может быть одновременно предоставлен доступ к общему ресурсу. Если параметр *число* принимает значение 1, к ресурсу может обратиться только один поток исполнения. По умолчанию ожидающим потокам исполнения предоставляется разрешение в неопределенном порядке. Если же присвоить параметру *способ* логическое значение true, то тем самым можно гарантировать, что разрешения будут предоставляться ожидающим потокам исполнения в том порядке, в каком они запрашивали доступ.

Чтобы получить разрешение, достаточно вызвать метод `acquire()`, который имеет две формы:

```
void acquire() throws InterruptedException
void acquire(int число) throws InterruptedException
```

Первая форма запрашивает одно разрешение, а вторая — *число* разрешений. Обычно используется первая форма. Если разрешение не будет предоставлено во время вызова метода, то исполнение вызывающего потока будет приостановлено до тех пор, пока не будет получено разрешение.

Чтобы освободить разрешение, следует вызвать метод `release()`. Ниже приведены общие формы этого метода.

```
void release()
void release(int число)
```

В первой форме освобождается одно разрешение, а во второй — количество разрешений, обозначаемое параметром *число*.

Чтобы воспользоваться семафором для управления доступом к ресурсу, каждый поток исполнения, которому требуется этот ресурс, должен вызвать метод `acquire()`, прежде чем обращаться к ресурсу. Когда поток исполнения завершает пользование ресурсом, он должен вызвать метод `release()`, чтобы освободить ресурс. В приведенном ниже примере программы демонстрируется применение семафора.

```
// Простой пример применения семафора

import java.util.concurrent.*;

class SemDemo {

    public static void main(String args[]) {
        Semaphore sem = new Semaphore(1);

        new IncThread(sem, "A");
        new DecThread(sem, "B");
    }
}
```



```

}

// Общий ресурс
class Shared {
    static int count = 0;
}

// Поток исполнения, увеличивающий значение счетчика на единицу
class IncThread implements Runnable {
    String name;
    Semaphore sem;
    IncThread(Semaphore s, String n) {
        sem = s;
        name = n;
        new Thread(this).start();
    }

    public void run() {
        System.out.println("Запуск потока " + name);
        try {
            // сначала получить разрешение
            System.out.println("Поток" + name + " ожидает разрешения");
            sem.acquire();
            System.out.println("Поток" + name + " получает разрешение");
            // а теперь получить доступ к общему ресурсу

            for(int i=0; i < 5; i++) {
                Shared.count++;
                System.out.println(name + ": " + Shared.count);

                // разрешить, если возможно, переключение контекста
                Thread.sleep(10);
            }
        } catch (InterruptedException exc) {
            System.out.println(exc);
        }

        // освободить разрешение
        System.out.println("Поток" + name + " освобождает разрешение");
        sem.release();
    }
}

// Поток исполнения, уменьшающий значение счетчика на единицу
class DecThread implements Runnable {
    String name;
    Semaphore sem;

    DecThread(Semaphore s, String n) {
        sem = s;
        name = n;
        new Thread(this).start();
    }

    public void run() {
        System.out.println("Запуск потока " + name);

        try {

```

```
// сначала получить разрешение
System.out.println("Поток" + name + " ожидает разрешения");
sem.acquire();
System.out.println("Поток" + name + " получает разрешение");

// а теперь получить доступ к общему ресурсу
for(int i=0; i < 5; i++) {
    Shared.count--;
    System.out.println(name + ": " + Shared.count);

    // разрешить, если возможно, переключение контекста
    Thread.sleep(10);
}
} catch (InterruptedException exc) {
    System.out.println(exc);
}

// освободить разрешение
System.out.println("Поток" + name + " освобождает разрешение");
sem.release();
}
}
```

Ниже приведен примерный результат выполнения данной программы. (Конкретный порядок следования потоков исполнения может быть иным.)

```
Запуск потока А
Поток А ожидает разрешения
Поток А получает разрешение
А: 1
Запуск потока В
Поток В ожидает разрешения
А: 2
А: 3
А: 4
А: 5
Поток А освобождает разрешение
Поток В получает разрешение
В: 4
В: 3
В: 2
В: 1
В: 0
Поток В освобождает разрешение
```

Для управления доступом к переменной `count`, которая является статической переменной класса `Shared`, в данной программе используется семафор. Значение переменной `Shared.count` увеличивается на 5 в методе `run()` из класса `IncThread` и уменьшается на 5 в одноименном методе из класса `DecThread`. Для защиты потоков исполнения, представленных этими двумя классами, от одновременного доступа к переменной `Shared.count` такой доступ предоставляется только после того, как будет получено разрешение от управляющего семафора. По завершении доступа к данной переменной как к общему ресурсу разрешение на него освобождается. Таким образом, только один поток исполнения может одновременно получить доступ к переменной `Shared.count`, что и подтверждают результаты выполнения данной программы.

Обратите внимание на то, что в методе `run()` из классов `IncThread` и `DecThread` вызывается метод `sleep()`. Он гарантирует, что доступ к переменной `Shared.count` будет синхронизироваться семафором. В частности, вызов метода `sleep()` из метода `run()` приводит к тому, что вызывающий поток исполнения будет приостанавливаться в промежутках между последовательными попытками доступа к переменной `Shared.count`. Это, как правило, позволяет исполняться второму потоку. Но благодаря семафору второй поток исполнения должен ожидать до тех пор, пока первый поток исполнения не освободит разрешение. А это произойдет только после того, как будут завершены все попытки доступа со стороны первого потока исполнения. Таким образом, значение переменной `Shared.count` сначала увеличивается на 5 в объекте класса `IncThread`, а затем уменьшается на 5 в объекте класса `DecThread`. Увеличение и уменьшение значения этой переменной происходит *строго по порядку*.

Если бы в данном примере не использовался семафор, то попытки доступа к переменной `Shared.count`, производимые каждым потоком исполнения, осуществлялись бы одновременно, поэтому увеличение и уменьшение значения этой переменной происходило бы *не по порядку*. Чтобы убедиться в этом, попробуйте закомментировать вызовы методов `acquire()` и `release()`. Запустив данную программу на выполнение, вы обнаружите, что доступ к переменной `Shared.count` больше не является синхронизированным, и каждый поток исполнения обращается к переменной `Shared.count`, как только для него выделяется временной интервал.

Несмотря на то что применение семафора, как правило, не представляет собой сложности, как демонстрируется в предыдущем примере программы, возможны и более сложные варианты его применения. Ниже приведен один из таких примеров. Он представляет собой переработанную версию программы, реализующей функции поставщика и потребителя из главы 11. В данном варианте используются два семафора, регулирующие потоки исполнения поставщика и потребителя и гарантирующие, что после каждого вызова метода `put()` будет следовать соответствующий вызов метода `get()`.

```
// Реализация поставщика и потребителя, использующая
// семафоры для управления синхронизацией
import java.util.concurrent.Semaphore;

class Q {
    int n;

    // начать с недоступного семафора потребителя
    static Semaphore semCon = new Semaphore(0);
    static Semaphore semProd = new Semaphore(1);

    void get() {
        try {
            semCon.acquire();
        } catch (InterruptedException e) {
            System.out.println(
                "Перехвачено исключение типа InterruptedException");
        }

        System.out.println("Получено: " + n);
    }
}
```

```

        semProd.release();
    }

    void put(int n) {
        try {
            semProd.acquire();
        } catch (InterruptedException e) {
            System.out.println(
                "Перехвачено исключение типа InterruptedException");
        }
        this.n = n;
        System.out.println("Отправлено: " + n);
        semCon.release();
    }
}

class Producer implements Runnable {
    Q q;

    Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }

    public void run() {
        for(int i=0; i < 20; i++) q.put(i);
    }
}

class Consumer implements Runnable {
    Q q;
    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }

    public void run() {
        for(int i=0; i < 20; i++) q.get();
    }
}

class ProdCon {
    public static void main(String args[]) {
        Q q = new Q();
        new Consumer(q);
        new Producer(q);
    }
}

```

Ниже показана часть результатов, выводимых данной программой.

```

Отправлено: 0
Получено: 0
Отправлено: 1
Получено: 1
Отправлено: 2
Получено: 2
Отправлено: 3
Получено: 3
Отправлено: 4
Получено: 4
Отправлено: 5

```

Получено: 5

·
·
·

Как видите, в данном примере синхронизируются вызовы методов `put()` и `get()`. Это означает, что после каждого вызова метода `put()` следует вызов метода `get()`, и поэтому ни одно значение не может быть пропущено. Если бы не семафоры, вызовы метода `put()` могли бы происходить несогласованно с вызовами метода `get()`, что привело бы к пропуску некоторых значений. (Чтобы убедиться в этом, удалите код семафора из данного примера и посмотрите полученные результаты.)

Надлежащая последовательность вызовов методов `put()` и `get()` соблюдается двумя семафорами: `semProd` и `semCon`. Прежде чем метод `put()` сможет предоставить значение, он должен получить разрешение от семафора `semProd`. Установив значение, он освобождает семафор `semProd`. Прежде чем метод `get()` сможет употребить значение, он должен получить разрешение от семафора `semCon`. Употребив значение, он освобождает семафор `semCon`. Такой механизм передачи и получения значений гарантирует, что после каждого вызова метода `put()` будет следовать вызов метода `get()`.

Обратите внимание на то, что семафор `semCon` инициализируется без доступных разрешений. Этим гарантируется, что метод `put()` выполняется первым. Возможность задавать исходное состояние синхронизации является одной из самых сильных сторон семафоров.

Класс `CountDownLatch`

Иногда требуется, чтобы поток исполнения находился в режиме ожидания до тех пор, пока не наступит одно (или больше) событие. Для этих целей в параллельном API предоставляется класс `CountDownLatch`, реализующий самоблокировку с обратным отсчетом. Объект этого класса изначально создается с количеством событий, которые должны произойти до того момента, как будет снята самоблокировка. Всякий раз, когда происходит событие, значение счетчика уменьшается. Как только значение счетчика достигнет нуля, самоблокировка будет снята.

В классе `CountDownLatch` имеется приведенный ниже конструктор, где параметр *число* определяет количество событий, которые должны произойти до того, как будет снята самоблокировка.

```
CountDownLatch(int число)
```

Для ожидания по самоблокировке в потоке исполнения вызывается метод `await()`, общие формы которого приведены ниже.

```
void await() throws InterruptedException  
boolean await(long ожидание, TimeUnit единица_времени)  
throws InterruptedException
```

В первой форме ожидание длится до тех пор, пока отсчет, связанный с вызывающим объектом типа `CountDownLatch`, не достигнет нуля. А во второй форме ожидание длится только в течение определенного периода времени, определяемого параметром *ожидание*. Время ожидания указывается в единицах, обознача-

емых параметром *единица_времени*, который принимает объект перечисления `TimeUnit`, рассматриваемого далее в этой главе. Метод `await()` возвращает логическое значение `false`, если достигнут предел времени ожидания, или логическое значение `true`, если обратный отсчет достигает нуля.

Чтобы известить о событии, следует вызвать метод `countDown()`. Ниже приведена общая форма этого метода. Всякий раз, когда вызывается метод `countDown()`, отсчет, связанный с вызывающим объектом, уменьшается на единицу.

```
void countDown()
```

В приведенном ниже примере программы демонстрируется применение класса `CountDownLatch`. В этой программе устанавливается самоблокировка, которая снимается только после наступления пяти событий.

```
// Продемонстрировать применение класса CountDownLatch

import java.util.concurrent.CountDownLatch;

class CDLDemo {
    public static void main(String args[]) {
        CountDownLatch cdl = new CountDownLatch(5);

        System.out.println("Запуск потока исполнения");

        new MyThread(cdl);

        try {
            cdl.await();
        } catch (InterruptedException exc) {
            System.out.println(exc);
        }
        System.out.println("Завершение потока исполнения");
    }
}

class MyThread implements Runnable {
    CountDownLatch latch;

    MyThread(CountDownLatch c) {
        latch = c;
        new Thread(this).start();
    }

    public void run() {
        for(int i = 0; i<5; i++) {
            System.out.println(i);
            latch.countDown(); // обратный отсчет
        }
    }
}
```

Ниже приведен результат выполнения данной программы.

```
Запуск потока исполнения
0
1
2
3
4
Завершение потока исполнения
```

В теле метода `main()` устанавливается самоблокировка в виде объекта `cdl` типа `CountDownLatch` с исходным значением обратного отсчета, равным 5. Затем создается экземпляр класса `MyThread`, который начинает исполнение нового потока. Обратите внимание на то, что объект `cdl` передается в качестве параметра конструктору класса `MyThread` и сохраняется в переменной экземпляра `latch`. Далее в главном потоке исполнения вызывается метод `await()` для объекта `cdl`, в результате чего исполнение главного потока приостанавливается до тех пор, пока обратный отсчет самоблокировки не уменьшится на единицу пять раз в объекте `cdl`.

В теле метода `run()` из класса `MyThread` организуется цикл, который повторяется пять раз. На каждом шаге этого цикла вызывается метод `countDown()` для переменной экземпляра `latch`, которая ссылается на объект `cdl` в методе `main()`. По завершении пятого шага цикла самоблокировка снимается, позволяя возобновить главный поток исполнения.

Класс `CountDownLatch` является эффективным и простым в употреблении средством синхронизации, которое окажется полезным в тех случаях, когда поток исполнения должен находиться в состоянии ожидания до тех пор, пока не произойдет одно или несколько событий.

Класс `CyclicBarrier`

В программировании нередко возникают такие ситуации, когда два или больше потока должны находиться в режиме ожидания в предопределенной точке исполнения до тех пор, пока все эти потоки не достигнут данной точки. Для этой цели в параллельном API предоставляется класс `CyclicBarrier`. Он позволяет определить объект синхронизации, который приостанавливается до тех пор, пока определенное количество потоков исполнения не достигнет некоторой барьерной точки.

В классе `CyclicBarrier` определены следующие конструкторы:

```
CyclicBarrier(int количество_потоков)  
CyclicBarrier(int количество_потоков, Runnable действие)
```

где параметр *количество_потоков* определяет число потоков, которые должны достигнуть некоторого барьера до того, как их исполнение будет продолжено. Во второй форме конструктора параметр *действие* определяет поток, который будет исполняться по достижении барьера.

Общая процедура применения класса `CyclicBarrier` следующая. Прежде всего нужно создать объект класса `CyclicBarrier`, указав количество ожидающих потоков исполнения. А когда каждый поток исполнения достигнет барьера, следует вызвать метод `await()` для данного объекта. В итоге исполнение потока будет приостановлено до тех пор, пока метод `await()` не будет вызван во всех остальных потоках исполнения. Как только указанное количество потоков исполнения достигнет барьера, произойдет возврат из метода `await()`, и выполнение будет возобновлено. А если дополнительно указать какое-нибудь действие, то будет выполнен соответствующий поток.

У метода `await()` имеются следующие общие формы:

```
int await() throws InterruptedException, BrokenBarrierException  
int await(long ожидание, TimeUnit единица_времени)  
throws InterruptedException, BrokenBarrierException, TimeoutException
```

В первой форме ожидание длится до тех пор, пока каждый поток исполнения не достигнет барьерной точки. А во второй форме ожидание длится только в течение определенного периода времени, определяемого параметром *ожидание*. Время ожидания указывается в единицах, обозначаемых параметром *единица_времени*. В обеих формах возвращается значение, указывающее порядок, в котором потоки исполнения будут достигать барьерной точки. Первый поток исполнения возвращает значение, равное количеству ожидаемых потоков минус 1, а последний поток возвращает нулевое значение.

В приведенном ниже примере программы демонстрируется применение класса `CyclicBarrier`. Эта программа ожидает до тех пор, пока все три потока достигнут барьерной точки. Как только это произойдет, будет выполнен поток, определяемый действием типа `BarAction`.

```
// Продемонстрировать применение класса CyclicBarrier

import java.util.concurrent.*;

class BarDemo {
    public static void main(String args[]) {
        CyclicBarrier cb = new CyclicBarrier(3, new BarAction() );

        System.out.println("Запуск потоков");

        new MyThread(cb, "A");
        new MyThread(cb, "B");
        new MyThread(cb, "C");
    }
}

// Поток исполнения, использующий барьер типа CyclicBarrier

class MyThread implements Runnable {
    CyclicBarrier cbar;
    String name;

    MyThread(CyclicBarrier c, String n) {
        cbar = c;

        name = n;
        new Thread(this).start();
    }

    public void run() {

        System.out.println(name);

        try {
            cbar.await();
        } catch (BrokenBarrierException exc) {
            System.out.println(exc);
        } catch (InterruptedException exc) {
            System.out.println(exc);
        }
    }
}

// Объект этого класса вызывается по достижении
// барьера типа CyclicBarrier
```



```

class BarAction implements Runnable {
    public void run() {
        System.out.println("Барьер достигнут!");
    }
}

```

Ниже приведен примерный результат выполнения данной программы. (Конкретный порядок следования потоков исполнения может быть иным.)

```

Запуск потоков
A
B
C
Барьер достигнут!

```

Класс `CyclicBarrier` можно использовать повторно, поскольку он освобождает ожидающие потоки исполнения всякий раз, когда метод `await()` вызывается из заданного количества потоков исполнения. Так, если внести следующие изменения в метод `main()` из предыдущего примера программы:

```

public static void main(String args[]) {
    CyclicBarrier cb = new CyclicBarrier(3, new BarAction() );

    System.out.println("Запуск потоков");

    new MyThread(cb, "A");
    new MyThread(cb, "B");
    new MyThread(cb, "C");
    new MyThread(cb, "X");
    new MyThread(cb, "Y");
    new MyThread(cb, "Z");
}

```

то результат ее выполнения будет таким, как показано ниже. (Конкретный порядок следования потоков исполнения может быть иным.)

```

Запуск потоков
A
B
C
Барьер достигнут!
X
Y
Z
Барьер достигнут!

```

Как показывает рассмотренный выше пример, класс `CyclicBarrier` представляет изящное решение задачи, которая раньше считалась сложной.

Класс `Exchanger`

Вероятно, наиболее интересным с точки зрения синхронизации является класс `Exchanger`, предназначенный для упрощения процесса обмена данными между двумя потоками исполнения. Принцип действия класса `Exchanger` очень прост: он ожидает до тех пор, пока два отдельных потока исполнения не вызовут его метод `exchange()`. Как только это произойдет, он произведет обмен данными, предоставляемыми обоими потоками. Такой механизм обмена данными не

только изящен, но и прост в применении. Нетрудно представить, как воспользоваться классом `Exchanger`. Например, один поток исполнения подготавливает буфер для приема данных через сетевое соединение, а другой — заполняет этот буфер данными, поступающими через сетевое соединение. Оба потока исполнения действуют совместно, поэтому всякий раз, когда требуется новая буферизация, осуществляется обмен данными.

Класс `Exchanger` является обобщенным и объявляется приведенным ниже образом, где параметр `V` определяет тип обмениваемых данных.

`Exchanger<V>`

В классе `Exchanger` определяется единственный метод `exchange()`, имеющий следующие общие формы:

```
V exchange(V буфер) throws InterruptedException  
V exchange(V буфер, long ожидание, TimeUnit единица времени)  
throws InterruptedException, TimeoutException
```

где параметр *буфер* обозначает ссылку на обмениваемые данные. Возвращаются данные, полученные из другого потока исполнения. Вторая форма метода `exchange()` позволяет определить время ожидания. Главная особенность метода `exchange()` состоит в том, что он не завершится успешно до тех пор, пока не будет вызван для одного и того же объекта типа `Exchanger` из двух отдельных потоков исполнения. Подобным образом метод `exchange()` синхронизирует обмен данными.

В приведенном ниже примере программы демонстрируется применения класса `Exchanger`. В этой программе создаются два потока исполнения. В одном потоке исполнения создается пустой буфер, принимающий данные из другого потока исполнения. Таким образом, первый поток исполнения обменивает пустую символьную строку на полную.

```
// Пример применения класса Exchanger  
  
import java.util.concurrent.Exchanger;  
  
class ExgrDemo {  
    public static void main(String args[]) {  
        Exchanger<String> exgr = new Exchanger<String>();  
        new UseString(exgr);  
        new MakeString(exgr);  
    }  
}  
  
// Поток типа Thread, формирующий символьную строку  
class MakeString implements Runnable {  
    Exchanger<String> ex;  
    String str;  
    MakeString(Exchanger<String> c) {  
        ex = c;  
        str = new String();  
        new Thread(this).start();  
    }  
  
    public void run() {  
        char ch = 'A';  
        for(int i = 0; i < 3; i++) {
```

```

        // заполнить буфер
        for(int j = 0; j < 5; j++)
            str += (char) ch++;
        try {
            // обменять заполненный буфер на пустой
            str = ex.exchange(str);
        } catch (InterruptedException exc) {
            System.out.println(exc);
        }
    }
}

// Поток типа Thread, использующий символьную строку
class UseString implements Runnable {
    Exchanger<String> ex;
    String str;
    UseString(Exchanger<String> c) {
        ex = c;
        new Thread(this).start();
    }

    public void run() {
        for(int i=0; i < 3; i++) {
            try {
                // обменять пустой буфер на заполненный
                str = ex.exchange(new String());
                System.out.println("Получено: " + str);
            } catch (InterruptedException exc) {
                System.out.println(exc);
            }
        }
    }
}

```

Ниже показаны результаты выполнения данной программы.

```

Получено: ABCDE
Получено: FGHIJ
Получено: KLMNO

```

В методе `main()` данной программы сначала создается объект класса `Exchanger`. Этот объект служит для синхронизации обмена символьными строками между классами `MakeString` и `UseString`. Класс `MakeString` заполняет символьную строку данными, а класс `UseString` обменивает пустую символьную строку на полную, отображая затем ее содержимое. Обмен пустой символьной строки на полную в буфере синхронизируется методом `exchange()`, который вызывается из метода `run()` в классах `MakeString` и `UseString`.

Класс Phaser

В версии JDK 7 внедрен новый класс синхронизации под названием `Phaser`. Главное его назначение – синхронизировать потоки исполнения, которые представляют одну или несколько стадий (или фаз) выполнения действия. Например, в прикладной программе может быть несколько потоков исполнения, реализую-

ших три стадии обработки заказов. На первой стадии отдельные потоки исполнения используются для того, чтобы проверить сведения о клиенте, наличие товара на складе и его цену. По завершении этой стадии остаются два потока исполнения, где на второй стадии вычисляется стоимость доставки и сумма соответствующего налога, а на заключительной стадии подтверждается оплата и определяется ориентировочное время доставки. В прошлом для синхронизации нескольких потоков исполнения в такой прикладной программе пришлось бы немало потрудиться. А с появлением класса `Phaser` этот процесс значительно упростился.

Прежде всего следует иметь в виду, что класс `Phaser` действует подобно описанному ранее классу `CyclicBarrier`, за исключением того, что он поддерживает несколько фаз. В итоге класс `Phaser` позволяет определить объект синхронизации, ожидающий завершения определенной фазы. Затем он переходит к следующей фазе и снова ожидает ее завершения. Следует также иметь в виду, что класс `Phaser` можно использовать и для синхронизации только одной фазы. В этом отношении он действует подобно классу `CyclicBarrier`, хотя главное его назначение — синхронизация нескольких фаз. В классе `Phaser` определяются четыре конструктора. В данном разделе упоминаются следующие два конструктора этого класса:

```
Phaser()
Phaser(int количество_сторон)
```

Первый конструктор создает *синхронизатор фаз* с нулевым регистрационным счетом, а второй устанавливает значение регистрационного счета равным заданному *количеству_сторон*. Объекты, регистрируемые синхронизатором фаз, зачастую обозначаются термином *сторона*. Обычно имеется полное соответствие количества регистрируемых объектов и синхронизируемых потоков исполнения, хотя этого и не требуется. В обоих случаях текущая фаза является нулевой. Поэтому когда создается экземпляр класса `Phaser`, он первоначально находится в нулевой фазе.

Обычно класс `Phaser` используется следующим образом. Сначала создается новый экземпляр класса `Phaser`. Затем синхронизатор фаз регистрирует одну или несколько сторон, вызывая метод `register()` или указывая нужное количество сторон в конструкторе класса `Phaser`. Синхронизатор фаз ожидает до тех пор, пока все зарегистрированные стороны не завершат фазу. Сторона извещает об этом, вызывая один из многих методов, предоставляемых классом `Phaser`, например метод `arrive()` или `arriveAndAwaitAdvance()`. Как только все стороны достигнут данной фазы, она считается завершенной, и синхронизатор фаз может перейти к следующей фазе (если она имеется) или завершить свою работу. Далее этот процесс поясняется более подробно.

Для регистрации стороны после создания объекта класса `Phaser` следует вызвать метод `register()`. Ниже приведена общая форма этого метода. В итоге он возвратит номер регистрируемой фазы.

```
int register()
```

Чтобы сообщить о завершении фазы, сторона должна вызвать метод `arrive()` или какой-нибудь его вариант. Когда количество достижений конца фазы сравняется с количеством зарегистрированных сторон, фаза завершится и объект класса

`Phaser` перейдет к следующей фазе (если она имеется). Метод `arrive()` имеет следующую общую форму:

```
int arrive()
```

Этот метод сообщает, что сторона (обычно поток исполнения) завершила некоторую задачу (или ее часть). Он возвращает текущий номер фазы. Если работа синхронизатора фаз завершена, этот метод возвращает отрицательное значение. Метод `arrive()` не приостанавливает исполнение вызывающего потока. Это означает, что он не ожидает завершения фазы. Этот метод должен быть вызван только зарегистрированной стороной.

Если требуется указать завершение фазы, а затем ожидать завершения этой фазы всеми остальными зарегистрированными сторонами, следует вызвать метод `arriveAndAwaitAdvance()`. Ниже приведена общая форма этого метода.

```
int arriveAndAwaitAdvance()
```

Этот метод ожидает до тех пор, пока все стороны не достигнут данной фазы, а затем возвращает номер следующей фазы или отрицательное значение, если синхронизатор фаз завершил свою работу. Метод `arriveAndAwaitAdvance()` должен быть вызван только зарегистрированной стороной.

Поток исполнения может достигнуть данной фазы, а затем сняться с регистрации, вызвав метод `arriveAndDeregister()`. Ниже приведена общая форма этого метода.

```
int arriveAndDeregister()
```

Этот метод возвращает номер текущей фазы или отрицательное значение, если синхронизатор фаз завершил свою работу. Он не ожидает завершения фазы. Метод `arriveAndDeregister()` должен быть вызван только зарегистрированной стороной.

Чтобы получить номер текущей фазы, следует вызвать метод `getPhase()`. Его общая форма выглядит следующим образом:

```
final int getPhase()
```

Когда создается объект класса `Phaser`, первая фаза получает нулевой номер, вторая фаза — номер 1, третья фаза — номер 2 и т.д. Если вызывающий объект класса `Phaser` завершил свою работу, возвращается отрицательное значение.

В приведенном ниже примере программы демонстрируется применение класса `Phaser`. В этой программе создаются три потока, каждый из которых имеет три фазы своего исполнения. Для синхронизации каждой фазы применяется класс `Phaser`.

```
// Пример применения класса Phaser
```

```
import java.util.concurrent.*;

class PhaserDemo {
    public static void main(String args[]) {
        Phaser phsr = new Phaser(1);
        int curPhase;

        System.out.println("запуск потоков");

        new MyThread(phsr, "A");
```

```
new MyThread(phsr, "B");
new MyThread(phsr, "C");

// ожидать завершения всеми потоками исполнения первой фазы
curPhase = phsr.getPhase();
phsr.arriveAndAwaitAdvance();
System.out.println("фаза " + curPhase + " завершена");

// ожидать завершения всеми потоками исполнения второй фазы
curPhase = phsr.getPhase();
phsr.arriveAndAwaitAdvance();
System.out.println("фаза " + curPhase + " завершена");

curPhase = phsr.getPhase();
phsr.arriveAndAwaitAdvance();
System.out.println("фаза " + curPhase + " завершена");

// снять основной поток исполнения с регистрации
phsr.arriveAndDeregister();

if(phsr.isTerminated())
    System.out.println("Синхронизатор фаз завершен");
}
}

// Поток исполнения, использующий синхронизатор фаз типа Phaser
class MyThread implements Runnable {
    Phaser phsr;
    String name;

    MyThread(Phaser p, String n) {
        phsr = p;
        name = n;
        phsr.register();
        new Thread(this).start();
    }

    public void run() {

        System.out.println("Поток " + name + " начинает первую фазу");
        phsr.arriveAndAwaitAdvance(); // известить о достижении фазы

        // Небольшая пауза, чтобы не нарушить порядок вывода.
        // Только для иллюстрации, но необязательно для правильного
        // функционирования синхронизатора фаз
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
            System.out.println(e);
        }

        System.out.println("Поток " + name + " начинает вторую фазу");
        phsr.arriveAndAwaitAdvance(); // известить о достижении фазы

        // Небольшая пауза, чтобы не нарушить порядок вывода.
        // Только для иллюстрации, но необязательно для правильного
        // функционирования синхронизатора фаз
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
            System.out.println(e);
        }

        System.out.println("Поток " + name + " начинает третью фазу");
```

```

        phsr.arriveAndDeregister(); // известить о достижении фазы и
                                   // снять потоки с регистрации
    }
}

```

Ниже приведен результат, выводимый данной программой.

```

Поток А начинает первую фазу
Поток С начинает первую фазу
Поток В начинает первую фазу
Фаза 0 завершена
Поток В начинает вторую фазу
Поток С начинает вторую фазу
Поток А начинает вторую фазу
Фаза 1 завершена
Поток С начинает третью фазу
Поток В начинает третью фазу
Поток А начинает третью фазу
Фаза 2 завершена
Синхронизатор фаз завершен

```

Рассмотрим подробнее основные части данной программы. Сначала в методе `main()` создается объект `phsr` типа `Phaser` с начальным счетом сторон, равным 1 (что соответствует основному потоку исполнения). Затем создаются три объекта типа `MyThread` и запускаются три соответствующих потока исполнения. Обратите внимание на то, что объекту типа `MyThread` передается ссылка на объект `phsr` — синхронизатор фаз. Объекты типа `MyThread` используют этот синхронизатор фаз для синхронизации своих действий. Затем в методе `main()` вызывается метод `getPhase()`, чтобы получить номер текущей фазы (который первоначально является нулевым), а после этого — метод `arriveAndAwaitAdvance()`. В итоге выполнение метода `main()` приостанавливается до тех пор, пока не завершится нулевая, по существу, первая фаза. Но этого не произойдет до тех пор, пока все объекты типа `MyThread` не вызовут метод `arriveAndAwaitAdvance()`. Как только это произойдет, метод `main()` возобновит свое выполнение, сообщив о завершении нулевой фазы, и перейдет к следующей фазе. Этот процесс повторяется до завершения всех трех фаз. Затем в методе `main()` вызывается метод `arriveAndDeregister()`, чтобы снять с регистрации все три объекта типа `MyThread`. В итоге зарегистрированных сторон больше не остается, а следовательно, перейдя к следующей фазе, синхронизатор фаз завершит свою работу.

Теперь рассмотрим объект типа `MyThread`. Прежде всего обратите внимание на то, что конструктору передается ссылка на синхронизатор фаз, в котором новый поток исполнения регистрируется далее как отдельная сторона. Таким образом, каждый новый объект типа `MyThread` становится стороной, зарегистрированной синхронизатором фаз, переданным конструктору этого объекта. Обратите также внимание на то, что у каждого потока исполнения имеются три фазы. В данном примере каждая фаза состоит из метки-заполнителя, которая просто отображает имя потока исполнения и то, что он делает. Безусловно, реальный код выполнял бы в потоке более полезные действия. Между первыми двумя фазами поток исполнения вызывает метод `arriveAndAwaitAdvance()`. Таким образом, каждый поток исполнения ожидает завершения фазы всеми остальными потоками, включая и основной поток. По завершении всех потоков исполнения, включая и основной,

синхронизатор фаз переходит к следующей фазе. А по завершении третьей фазы каждый поток исполнения снимается с регистрации, вызывая метод `arriveAndDeregister()`. Как поясняется в комментариях к объектам типа `MyThread`, метод `sleep()` вызывается исключительно в иллюстративных целях, чтобы предотвратить нарушение вывода из-за многопоточности, хотя это и не обязательно для правильного функционирования синхронизатора фаз. Если удалить вызовы метода `sleep()`, то выводимый результат может выглядеть немного запутанным, но фазы все равно будут синхронизированы правильно.

Следует также иметь в виду, что в рассматриваемом здесь примере используются три однотипных потока исполнения, хотя это и не обязательное требование. Каждая сторона, пользующаяся синхронизатором фаз, может быть совершенно не похожей на остальные, выполняя свою задачу.

Все, что происходит при переходе к следующей фазе, вполне поддается контролю. Для этого следует переопределить метод `onAdvance()`. Этот метод вызывается исполняющей средой, когда синхронизатор фаз переходит от одной фазы к следующей. Ниже приведена общая форма данного метода.

```
protected boolean onAdvance(int фаза, int количество_сторон)
```

Здесь параметр *фаза* обозначает текущий номер фазы перед его приращением, а параметр *количество_сторон* — число зарегистрированных сторон. Для того чтобы завершить работу синхронизатора фаз, метод `onAdvance()` должен возвратить логическое значение `true`. А для того чтобы продолжить работу синхронизатора фаз, метод `onAdvance()` должен возвратить логическое значение `false`. В версии по умолчанию метод `onAdvance()` возвращает логическое значение `true`, чтобы завершить работу синхронизатора фаз, если зарегистрированных сторон больше нет. Как правило, переопределяемая версия данного метода должна следовать этой практике.

Метод `onAdvance()` может быть, в частности, переопределен для того, чтобы дать синхронизатору фаз возможность выполнить заданное количество фаз, а затем остановиться. Ниже приведен характерный тому пример. В данном примере создается класс `MyPhaser`, расширяющий класс `Phaser` таким образом, чтобы выполнять заданное количество фаз. С этой целью переопределяется метод `onAdvance()`. Конструктор класса `MyPhaser` принимает один аргумент, задающий количество выполняемых фаз. Обратите внимание на то, что класс `MyPhaser` автоматически регистрирует одну сторону. Это удобно для целей данного примера, но у конкретной прикладной программы могут быть другие потребности.

```
// Расширить класс Phaser и переопределить метод onAdvance()  
// таким образом, чтобы было выполнено только определенное  
// количество фаз
```

```
import java.util.concurrent.*;
```

```
// Расширить класс MyPhaser, чтобы выполнить только  
// определенное количество фаз  
class MyPhaser extends Phaser {  
    int numPhases;
```

```
    MyPhaser(int parties, int phaseCount) {  
        super(parties);
```



```

        numPhases = phaseCount - 1;
    }

    // переопределить метод onAdvance(), чтобы выполнить
    // определенное количество фаз
    protected boolean onAdvance(int p, int regParties) {
        // Следующий оператор println() требуется только для
        // целей иллюстрации. Как правило, метод onAdvance()
        // не отображает выводимые данные
        System.out.println("фаза " + p + " завершена.\n");

        // вернуть логическое значение true,
        // если все фазы завершены
        if(p == numPhases || regParties == 0) return true;

        // В противном случае вернуть логическое значение false
        return false;
    }
}

class PhaserDemo2 {
    public static void main(String args[]) {

        MyPhaser phsr = new MyPhaser(1, 4);

        System.out.println("Запуск потоков\n");

        new MyThread(phsr, "A");
        new MyThread(phsr, "B");
        new MyThread(phsr, "C");

        // ожидать завершения определенного количества фаз
        while(!phsr.isTerminated()) {
            phsr.arriveAndAwaitAdvance();
        }

        System.out.println("Синхронизатор фаз завершен");
    }
}

// Поток исполнения, использующий синхронизатор фаз типа Phaser
class MyThread implements Runnable {
    Phaser phsr;
    String name;

    MyThread(Phaser p, String n) {
        phsr = p;
        name = n;
        phsr.register();
        new Thread(this).start();
    }

    public void run() {

        while(!phsr.isTerminated()) {
            System.out.println(
                "Поток " + name + " начинает фазу " + phsr.getPhase());
            phsr.arriveAndAwaitAdvance();

            // Небольшая пауза, чтобы не нарушить порядок вывода.
            // Только для иллюстрации, но необязательно для правильного
            // функционирования синхронизатора фаз
            try {
                Thread.sleep(10);
            }
        }
    }
}

```

```

        } catch (InterruptedException e) {
            System.out.println(e);
        }
    }
}

```

Ниже приведен результат, выводимый данной программой.

```

Запуск потоков
Поток В начинает фазу 0
Поток А начинает фазу 0
Поток С начинает фазу 0
фаза 0 завершена

Поток А начинает фазу 1
Поток В начинает фазу 1
Поток С начинает фазу 1
фаза 1 завершена

Поток С начинает фазу 2
Поток В начинает фазу 2
Поток А начинает фазу 2
фаза 2 завершена

Поток С Beginning Phase 3
Поток В начинает фазу 3
Поток А начинает фазу 3
фаза 3 завершена

Синхронизатор фаз завершен

```

В методе `main()` создается один экземпляр класса `Phaser`. В качестве аргумента ему передается значение 4. Это означает, что он будет выполняться в течение четырех фаз, а затем завершится. Затем создаются три потока исполнения и начинается следующий цикл, как показано ниже.

```

// ожидать завершения определенного количества фаз
while(!phsr.isTerminated()) {
    phsr.arriveAndAwaitAdvance();
}

```

В этом цикле метод `arriveAndAwaitAdvance()` вызывается до завершения работы синхронизатора фаз, а этого не произойдет до тех пор, пока не будет выполнено определенное количество фаз. В данном случае цикл будет продолжаться до тех пор, пока не завершатся все четыре фазы. Следует также иметь в виду, что метод `arriveAndAwaitAdvance()` вызывается из потоков исполнения вплоть до завершения работы синхронизатора фаз в данном цикле. Это означает, что потоки исполняются до завершения заданного количества фаз.

А теперь рассмотрим подробнее исходный код метода `onAdvance()`. Всякий раз, когда вызывается метод `onAdvance()`, ему передается текущая фаза и количество зарегистрированных сторон. Если текущая фаза соответствует указанной фазе или количество зарегистрированных сторон равно нулю, метод `onAdvance()` возвращает логическое значение `true`, прекращая таким образом работу синхронизатора фаз. Это делается в приведенном ниже фрагменте кода. Как можно заметить, для достижения желаемого результата необходимо совсем немного кода.

```
// вернуть логическое значение true, если все фазы завершены
if(p == numPhases || regParties == 0) return true;
```

Прежде чем завершить эту тему, следует заметить, что расширять класс `Phaser` совсем не обязательно. Как и в предыдущем примере, для этого достаточно переопределить метод `onAdvance()`. В некоторых случаях может быть создан более компактный код с помощью анонимного внутреннего класса, переопределяющего метод `onAdvance()`.

У класса `Phaser` имеются дополнительные возможности, которые могут оказаться полезными в прикладных программах. В частности, вызвав метод `awaitAdvance()`, можно ожидать конкретной фазы, как показано ниже.

```
int awaitAdvance(int фаза)
```

Здесь параметр *фаза* обозначает номер фазы, в течение которой метод `awaitAdvance()` находится в состоянии ожидания до тех пор, пока не произойдет переход к следующей фазе. Этот метод завершится немедленно, если передаваемый ему параметр *фаза* не будет равен текущей фазе. Он завершится сразу и в том случае, если синхронизатор фаз завершит работу. Но если в качестве параметра *фаза* этому методу будет передана текущая фаза, то он будет ожидать перехода к следующей фазе. Этот метод должен быть вызван только зарегистрированной стороной. Имеется также прерывающая версия этого метода под названием `awaitAdvanceInterruptibly()`.

Чтобы зарегистрировать несколько сторон, следует вызвать метод `bulkRegister()`; чтобы получить количество зарегистрированных сторон — метод `getRegisteredParties()`; чтобы получить количество сторон, достигших или не достигших своей фазы, — метод `getArrivedParties()` или `getUnarrivedParties()` соответственно; а чтобы перевести синхронизатор фаз в завершённое состояние — метод `forceTermination()`.

Класс `Phaser` позволяет также создать дерево синхронизаторов фаз. Он снабжен двумя дополнительными конструкторами, которые позволяют указать родительский синхронизатор фаз и метод `getParent()`.

Применение исполнителя

В параллельном API поддерживается также средство, называемое *исполнителем* и предназначенное для создания потоков исполнения и управления ими. В этом отношении исполнитель служит альтернативой управлению потоками исполнения средствами класса `Thread`.

В основу исполнителя положен интерфейс `Executor`, в котором определяется следующий метод:

```
void execute(Runnable поток)
```

В результате вызова этого метода выполняется указанный *поток*. Следовательно, метод `execute()` запускает указанный поток на исполнение.

Интерфейс `ExecutorService` расширяет интерфейс `Executor`, дополняя его методами, помогающими управлять исполнением потоков и контролировать их.

Например, в интерфейсе `ExecutorService` определяется метод `shutdown()`, общая форма которого приведена ниже. Этот метод останавливает все потоки исполнения, находящиеся в данный момент под управлением экземпляра интерфейса `ExecutorService`.

```
void shutdown()
```

В интерфейсе `ExecutorService` определяются также методы, которые запускают потоки исполнения, возвращающие результаты, исполняют ряд потоков и определяют состояние остановки. Некоторые из этих методов будут рассмотрены далее. Имеется также интерфейс `ScheduledExecutorService`, расширяющий интерфейс `ExecutorService` для поддержки планирования потоков исполнения.

Кроме того, в параллельном API имеются три предопределенных класса исполнителей: `ThreadPoolExecutor`, `ScheduledThreadPoolExecutor` и `ForkJoinPool`. Класс `ThreadPoolExecutor` реализует интерфейсы `Executor` и `ExecutorService` и обеспечивает поддержку управляемого пула потоков исполнения. Класс `ScheduledThreadPoolExecutor` также реализует интерфейс `ScheduledExecutorService` для поддержки планирования пула потоков исполнения. А класс `ForkJoinPool` реализует интерфейсы `Executor` и `ExecutorService` и применяется в каркасе `Fork/Join Framework`, как поясняется далее в этой главе.

Пул потоков предоставляет ряд потоков исполнения для решения разнообразных задач. Вместо того создавать отдельный поток исполнения для каждой задачи, используются потоки из пула. Это позволяет сократить нагрузку, связанную с созданием множества отдельных потоков. Хотя классы `ThreadPoolExecutor` и `ScheduledThreadPoolExecutor` можно использовать напрямую, исполнитель чаще всего придется получать, вызывая один из следующих статических фабричных методов, определенных во вспомогательном классе `Executors`. Ниже приведены общие формы некоторых из этих методов.

```
static ExecutorService newCachedThreadPool()  
static ExecutorService newFixedThreadPool(int количество_потоков)  
static ScheduledExecutorService  
    newScheduledThreadPool(int количество_потоков)
```

Метод `newCachedThreadPool()` создает пул потоков исполнения, который не только вводит потоки исполнения по мере необходимости, но и по возможности повторно использует их. Метод `newFixedThreadPool()` создает пул потоков исполнения, состоящий из указанного количества потоков. А метод `newScheduledThreadPool()` создает пул потоков исполнения, в котором можно осуществлять планирование потоков исполнения. Каждый из них возвращает ссылку на интерфейс `ExecutorService`, предназначенный для управления пулом потоков исполнения.

Простой пример исполнителя

Прежде чем продолжить обсуждение данной темы, рассмотрим простой пример применения исполнителя. В приведенной ниже программе создается фиксированный пул, содержащий два потока исполнения. Затем этот пул используется для выполнения четырех задач. Таким образом, четыре задачи совместно исполь-

зуют два потока исполнения, находящихся в пуле. После того как задачи будут выполнены, пул закрывается и программа завершается.

// Простой пример применения исполнителя

```
import java.util.concurrent.*;

class SimpExec {
    public static void main(String args[]) {
        CountDownLatch cdl = new CountDownLatch(5);
        CountDownLatch cdl2 = new CountDownLatch(5);
        CountDownLatch cdl3 = new CountDownLatch(5);
        CountDownLatch cdl4 = new CountDownLatch(5);
        ExecutorService es = Executors.newFixedThreadPool(2);

        System.out.println("Запуск потоков");

        // запустить потоки исполнения
        es.execute(new MyThread(cdl, "A"));
        es.execute(new MyThread(cdl2, "B"));
        es.execute(new MyThread(cdl3, "C"));
        es.execute(new MyThread(cdl4, "D"));

        try {
            cdl.await();
            cdl2.await();
            cdl3.await();
            cdl4.await();
        } catch (InterruptedException exc) {
            System.out.println(exc);
        }

        es.shutdown();
        System.out.println("Завершение потоков");
    }
}

class MyThread implements Runnable {
    String name;
    CountDownLatch latch;

    MyThread(CountDownLatch c, String n) {
        latch = c;
        name = n;

        new Thread(this);
    }

    public void run() {
        for(int i = 0; i < 5; i++) {
            System.out.println(name + ": " + i);
            latch.countDown();
        }
    }
}
```

Ниже приведен результат выполнения данной программы.

Запуск потоков

A: 0

A: 1

```

A: 2
A: 3
A: 4
C: 0
C: 1
C: 2
C: 3
C: 4
D: 0
D: 1
D: 2
D: 3
D: 4
B: 0
B: 1
B: 2
B: 3
B: 4
Завершение потоков

```

Судя по приведенным выше результатам, выполняются все четыре задачи, несмотря на то, что в пуле содержится всего два потока исполнения. Но только две задачи могут быть выполнены одновременно, а остальные должны ожидать до тех пор, пока в пуле не освободится один из потоков исполнения, чтобы им можно было воспользоваться.

Вызов метода `shutdown()` очень важен. Если бы его не было в данной программе, она не смогла бы завершиться, поскольку исполнитель оставался бы активным. Убедиться в этом можно, закомментировав вызов метода `shutdown()` и посмотрев, что из этого получится.

Применение интерфейсов `Callable` и `Future`

К числу самых привлекательных средств в параллельном API относится интерфейс `Callable`. Он представляет поток исполнения, возвращающий значение. Объекты интерфейса `Callable` можно использовать в прикладной программе для вычисления результатов, которые затем возвращаются вызывающему потоку исполнения. И это довольно эффективный механизм, поскольку он облегчает написание кода для самых разных числовых расчетов, в которых частичные результаты вычисляются одновременно. Его можно использовать и для запуска потока исполнения, возвращающего код состояния, который свидетельствует об успешном выполнении потока.

Интерфейс `Callable` является обобщенным и объявляется следующим образом:

```
interface Callable<V>
```

где параметр `V` обозначает тип данных, возвращаемых потоком исполнения. В интерфейсе `Callable` определяется единственный метод `call()`, общая форма которого приведена ниже.

```
V call() throws Exception
```

В теле метода `call()` определяется задача, которую требуется выполнить. Когда она будет выполнена, возвращается результат. Если результат нельзя вычислить, метод `call()` генерирует исключение.

Для выполнения задачи типа `Callable` вызывается метод `submit()`, определенный в интерфейсе `ExecutorService`. У метода `submit()` имеются три общие формы, но для выполнения задачи типа `Callable` используется только одна из них:

```
<T> Future<T> submit(Callable<T> задача)
```

где параметр *задача* обозначает объект типа `Callable`, который будет выполняться в собственном потоке. Результат возвращается через объект типа `Future`.

Интерфейс `Future` является обобщенным интерфейсом и представляет значение, возвращаемое объектом типа `Callable`. А поскольку это значение будет получено через некоторое время в будущем, то имя интерфейса `Future` вполне соответствует его назначению. Интерфейс `Future` объявляется приведенным ниже образом, где параметр `V` определяет тип результата.

```
interface Future<V>
```

Чтобы получить значение, следует вызвать метод `get()` из интерфейса `Future`. Ниже приведены общие формы этого метода.

```
V get()
  throws InterruptedException, ExecutionException
V get(long ожидание, TimeUnit единица_времени)
  throws InterruptedException,
  ExecutionException, TimeoutException
```

В первой форме ожидание получения результатов длится бесконечно долго. А во второй форме ожидание длится только в течение определенного периода времени, определяемого параметром *ожидание*. Время ожидания указывается в единицах, обозначаемых параметром *единица_времени*, который принимает объект перечисления `TimeUnit`, рассматриваемого далее в этой главе.

В приведенном ниже примере программы демонстрируется применение интерфейсов `Callable` и `Future`. В этой программе формируются три задачи, выполняющие три разных вида вычислений. Первая задача возвращает суммарное значение, вторая находит длину гипотенузы прямоугольного треугольника с известными значениями длин его сторон, а третья определяет факториал заданного значения. Все три вида вычислений производятся одновременно.

```
// Пример применения интерфейса Callable
```

```
import java.util.concurrent.*;

class CallableDemo {
    public static void main(String args[]) {
        ExecutorService es = Executors.newFixedThreadPool(3);
        Future<Integer> f;
        Future<Double> f2;
        Future<Integer> f3;

        System.out.println("Запуск");

        f = es.submit(new Sum(10));
        f2 = es.submit(new Hypot(3, 4));
        f3 = es.submit(new Factorial(5));

        try {
            System.out.println(f.get());
            System.out.println(f2.get());
```

```
        System.out.println(f3.get());
    } catch (InterruptedException exc) {
        System.out.println(exc);
    }
    catch (ExecutionException exc) {
        System.out.println(exc);
    }
}

es.shutdown();
System.out.println("Завершение");
}

// Три потока исполнения вычислений
class Sum implements Callable<Integer> {
    int stop;

    Sum(int v) { stop = v; }

    public Integer call() {
        int sum = 0;
        for(int i = 1; i <= stop; i++) {
            sum += i;
        }
        return sum;
    }
}

class Hypot implements Callable<Double> {
    double sidel, side2;

    Hypot(double s1, double s2) {
        sidel = s1;
        side2 = s2;
    }

    public Double call() {
        return Math.sqrt((sidel*sidel) + (side2*side2));
    }
}

class Factorial implements Callable<Integer> {
    int stop;

    Factorial(int v) { stop = v; }

    public Integer call() {
        int fact = 1;
        for(int i = 2; i <= stop; i++) {
            fact *= i;
        }
        return fact;
    }
}
```

Ниже приведен результат выполнения данной программы.

Запуск

55

5.0

120

Завершение

Перечисление TimeUnit

В параллельном API определяются методы, принимающие параметр перечислимого типа `TimeUnit`, обозначающий период времени ожидания. Перечисление `TimeUnit` служит для обозначения *степени разрешения синхронизации*. Это перечисление определено в пакете `java.util.concurrent` и может принимать одно из следующих значений:

- DAYS
- HOURS
- MINUTES
- SECONDS
- MICROSECONDS
- MILLISECONDS
- NANOSECONDS

И хотя с помощью перечисления `TimeUnit` можно определить любое из этих значений в вызовах методов, принимающих параметр синхронизации, нет никакой гарантии того, что система сможет работать с заданным разрешением.

Ниже приведен пример, демонстрирующий применение перечисления `TimeUnit`. Класс `CallableDemo` из предыдущего примера был изменен таким образом, чтобы использовать вторую форму метода `get()`, принимающего параметр типа `TimeUnit`. В данном варианте ни один из вызываемых метода `get()` не будет ожидать дольше 10 миллисекунд.

```
try {
    System.out.println(f.get(10, TimeUnit.MILLISECONDS));
    System.out.println(f2.get(10, TimeUnit.MILLISECONDS));
    System.out.println(f3.get(10, TimeUnit.MILLISECONDS));
} catch (InterruptedException exc) {
    System.out.println(exc);
} catch (ExecutionException exc) {
    System.out.println(exc);
} catch (TimeoutException exc) {
    System.out.println(exc);
}
```

В перечислении `TimeUnit` определяются различные методы, выполняющие преобразование единиц. Ниже приведены их общие формы.

```
long convert(long время, TimeUnit единица_времени)
long toMicros(long время)
long toMillis(long время)
long toNanos(long время)
long toSeconds(long время)
long toDays(long время)
long toHours(long время)
long toMinutes(long время)
```

Метод `convert()` преобразует заданное время в единицы времени, обозначаемые параметром `единица_времени`, и возвращает результат. Методы типа `toXXX` выполняют указанное преобразование и возвращают результат.

В перечислении `TimeUnit` определяются также следующие методы синхронизации:

```
void sleep(long задержка) throws InterruptedException
void timedJoin(Thread поток, long задержка)
    throws InterruptedException
void timedWait(Object объект, long задержка)
    throws InterruptedException
```

Метод `sleep()` приостанавливает выполнение на определенный период времени, который задается в виде вызывающей константы перечислимого типа. Он преобразуется в вызов метода `Thread.sleep()`. Метод `timedJoin()` является специальной версией метода `Thread.join()`, где указанный *поток* исполнения приостанавливается на период времени, обозначаемый параметром *задержка*. Метод `timedWait()` также является специальной версией метода `Object.wait()`, где указанный *объект* ожидает в течение периода времени, обозначаемого параметром *задержка* в вызывающих единицах времени.

Параллельные коллекции

Как отмечалось ранее, в параллельном API определяется ряд коллекций, предназначенных для выполнения параллельных операций. К их числу относятся следующие коллекции:

- `ArrayBlockingQueue`
- `ConcurrentHashMap`
- `ConcurrentLinkedDeque`
- `ConcurrentLinkedQueue`
- `ConcurrentSkipListMap`
- `ConcurrentSkipListSet`
- `CopyOnWriteArrayList`
- `CopyOnWriteArraySet`
- `DelayQueue`
- `LinkedBlockingDeque`
- `LinkedBlockingQueue`
- `LinkedTransferQueue`
- `PriorityBlockingQueue`
- `SynchronousQueue`

Эти коллекции служат параллельной альтернативой соответствующим классам коллекций из каркаса `Collections Framework`. Они действуют таким же образом, как и остальные коллекции, но только поддерживают параллелизм. Программисты, имеющие опыт работы с каркасом `Collections Framework`, не испытывают никаких затруднений, применяя эти параллельные коллекции.

Блокировки

В пакете `java.util.concurrent.locks` предоставляется поддержка *блокировок*, которые являются объектами и служат альтернативой блокам кода `synchronized` для управления доступом к общему ресурсу. В общем, блокировки действуют следующим образом. Прежде чем получить доступ к общему ресурсу, запрашивается блокировка, защищающая этот ресурс. По завершении доступа к ресурсу блокировка снимается. Если один поток исполнения попытается запросить блокировку в тот момент, когда она используется каким-нибудь другим потоком исполнения, то первый поток будет ожидать до тех пор, пока блокировка не будет снята. Благодаря этому удастся избежать конфликтов при доступе к общему ресурсу.

Блокировки особенно полезны в тех случаях, когда нескольким потокам исполнения требуется получить доступ к значению из общих данных. Например, в прикладной программе складского учета может использоваться поток исполнения, в котором сначала подтверждается наличие товара на складе, а затем уменьшается количество доступных товаров после каждой продажи. Если будут выполняться два или несколько таких потоков, то без синхронизации может возникнуть ситуация, когда один поток исполнения начнет свою транзакцию в процессе выполнения транзакции другим потоком. Таким образом, в обоих потоках исполнения будет предполагаться достаточное количество товара на складе, хотя на самом деле этого товара хватит только для одной продажи. В подобных ситуациях синхронное исполнение потоков можно организовать с помощью блокировок.

Блокировка определяется в интерфейсе `Lock`. В табл. 28.2 перечислены методы, определенные в интерфейсе `Lock`. В общем, для получения блокировки следует вызвать метод `lock()`. Если блокировка недоступна, метод `lock()` войдет в состояние ожидания. Чтобы снять блокировку, следует вызвать метод `unlock()`, а для того чтобы выяснить, доступна ли блокировка, и если она доступна, получить ее, — вызвать метод `tryLock()`. Метод `tryLock()` не будет ожидать блокировку, если она недоступна. Напротив, он возвращает логическое значение `true`, если блокировка получена, а иначе — логическое значение `false`. Метод `newCondition()` возвращает объект типа `Condition`, связанный с блокировкой. Объект типа `Condition` позволяет добиться более полного контроля над блокировками с помощью методов `await()` и `signal()`, обеспечивающих такие же функциональные возможности, как и методы `Object.wait()` и `Object.notify()`.

Таблица 28.2. Методы из интерфейса `Lock`

Метод	Описание
<code>void lock()</code>	Ожидает до тех пор, пока не будет получена вызываемая блокировка
<code>void lockInterruptibly() throws InterruptedException</code>	Ожидает до тех пор, пока не будет получена вызываемая блокировка, если только не будет прервано ожидание
<code>Condition newCondition()</code>	Возвращает объект типа <code>Condition</code> , связанный с вызываемой блокировкой

Метод	Описание
<code>boolean tryLock()</code>	Пытается запросить блокировку. Этот метод не входит в состояние ожидания, если блокировка недоступна. Вместо этого он возвращает логическое значение true , если блокировка получена, или логическое значение false , если в данный момент блокировка используется другим потоком исполнения
<code>boolean tryLock(long <i>ожидание</i>, TimeUnit <i>единица_времени</i>) throws InterruptedException</code>	Пытается получить блокировку. Если блокировка недоступна, этот метод будет ожидать в течение периода времени, обозначаемого параметром <i>ожидание</i> . Время ожидания указывается в единицах, обозначаемых параметром <i>единица_времени</i> . Если блокировка получена, то возвращается логическое значение true . А если блокировка не была получена в течение заданного периода времени, то возвращается логическое значение false
<code>void unlock()</code>	Снимает блокировку

В пакете `java.util.concurrent.locks` предоставляется класс `ReentrantLock`, реализующий интерфейс `Lock`. Этот класс реализует *реентерабельную блокировку*, в которую может входить повторно поток исполнения, удерживающий в данный момент блокировку. Если поток входит в блокировку повторно, все вызовы метода `lock()` должны быть, разумеется, смещены на равное количество вызовов метода `unlock()`. В противном случае поток, пытающийся запросить блокировку, перейдет в режим ожидания до тех пор, пока она не будет снята.

В приведенном ниже примере программы демонстрируется применение блокировки. В этой программе создаются два потока исполнения, которые обращаются к переменной `Shared.count` в качестве общего ресурса. Прежде чем поток исполнения сможет обратиться к переменной `Shared.count`, он должен получить блокировку. После получения блокировки значение переменной `Shared.count` увеличивается, а далее поток исполнения входит в состояние ожидания, прежде чем снять блокировку. Вследствие этого другой поток исполнения будет пытаться получить блокировку. Но поскольку блокировка все еще удерживается первым потоком исполнения, то другой поток будет ожидать до тех пор, пока первый не выйдет из состояния ожидания и не снимет блокировку. Как показывает результат выполнения данной программы, доступ к переменной `Shared.count` синхронизируется с помощью блокировки.

// Простой пример блокировки

```
import java.util.concurrent.locks.*;

class LockDemo {

    public static void main(String args[]) {
        ReentrantLock lock = new ReentrantLock();

        new LockThread(lock, "A");
        new LockThread(lock, "B");
    }
}
```

```

}

// Общий ресурс
class Shared {
    static int count = 0;
}

// Поток исполнения, инкрементирующий значение счетчика
class LockThread implements Runnable {
    String name;
    ReentrantLock lock;
    LockThread(ReentrantLock lk, String n) {
        lock = lk;
        name = n;
        new Thread(this).start();
    }

    public void run() {

        System.out.println("Запуск потока" + name);

        try {
            // сначала заблокировать счетчик
            System.out.println(
                "Поток" + name + " ожидает блокировки счетчика");
            lock.lock();
            System.out.println("Поток" + name + " блокирует счетчик.");
            Shared.count++;
            System.out.println("Поток" + name + ": " + Shared.count);
            // а теперь переключение контекста, если это возможно
            System.out.println("Поток" + name + " ожидает");
            Thread.sleep(1000);
        } catch (InterruptedException exc) {
            System.out.println(exc);
        } finally {
            // снять блокировку
            System.out.println(
                "Поток" + name + " разблокирует счетчик");
            lock.unlock();
        }
    }
}

```

Ниже приведен примерный результат выполнения данной программы. (У вас порядок выполнения потоков может оказаться иным.)

```

Запуск потока А
Поток А ожидает блокировки счетчика
Поток А блокирует счетчик
Поток А: 1
Поток А ожидает
Запуск потока В
Поток В ожидает блокировки счетчика
Поток А разблокирует счетчик
Поток В блокирует счетчик
Поток В: 2
Поток В ожидает
Поток В разблокирует счетчик

```

В пакете `java.util.concurrent.locks` определяется также интерфейс `ReadWriteLock`, реализующий поддержку отдельных блокировок для доступа с целью

чтения и записи. Это дает возможность предоставлять несколько блокировок потокам исполнения только для чтения данных из ресурса, но не для их записи. Класс `ReentrantReadWriteLock` предоставляет соответствующую реализацию интерфейса `ReadWriteLock`.

На заметку! В версии JDK 8 введен класс `StampedLock`, реализующий специальную блокировку. Но этот класс не реализует интерфейс `Lock` или `ReadWriteLock`, а вместо этого предоставляет механизм, позволяющий использовать его средства подобно интерфейсу `Lock` или `ReadWriteLock`.

Атомарные операции

В пакете `java.util.concurrent.atomic` предоставляется альтернатива другим средствам синхронизации для чтения или записи значений переменных некоторых типов. В этом пакете доступны методы, которые получают, устанавливают или сравнивают значение переменной во время одной непрерывной, т.е. *атомарной*, операции. Это означает, что для выполнения такой операции не требуется ни блокировка, ни любой другой механизм синхронизации.

Атомарные операции выполняются с помощью классов `AtomicInteger` и `AtomicLong`, а также методов `get()`, `set()`, `compareAndSet()`, `decrementAndGet()` и `getAndSet()`, которые реализуют соответственно следующие действия: получение, установку, сравнение и установку, декремент и получение, получение и установку.

В следующем примере показано, как синхронизировать доступ к общему ресурсу с помощью класса `AtomicInteger`:

// Простой пример выполнения атомарных операций

```
import java.util.concurrent.atomic.*;

class AtomicDemo {
    public static void main(String args[]) {
        new AtomThread("A");
        new AtomThread("B");
        new AtomThread("C");
    }
}

class Shared {
    static AtomicInteger ai = new AtomicInteger(0);
}

// Поток исполнения, в котором инкрементируется значение счетчика
class AtomThread implements Runnable {
    String name;
    AtomThread(String n) {
        name = n;
        new Thread(this).start();
    }

    public void run() {
        System.out.println("Запуск потока" + name);
    }
}
```

```

        for(int i=1; i <= 3; i++)
            System.out.println("Поток" + name + " получено: " +
                               Shared.ai.getAndSet(i));
    }
}

```

В данной программе с помощью класса `Shared` сначала создается статический объект `ai` типа `AtomicInteger`, а затем три потока исполнения типа `AtomThread`. В методе `run()` объект `Shared.ai` изменяется в результате вызова метода `getAndSet()`. Этот метод возвращает предыдущее значение и устанавливает то значение, которое было передано в качестве параметра. Благодаря классу `AtomicInteger` исключается вероятность того, что два потока будут одновременно осуществлять запись данных в объект `ai`.

В общем, атомарные операции служат удобной (а, возможно, и более эффективной) альтернативой другим механизмам синхронизации при обращении к одной переменной в качестве общего ресурса. Начиная с версии JDK 8 в пакете `java.util.concurrent.atomic` предоставляются также следующие четыре класса, поддерживающие неблокируемые накопительные операции: `DoubleAccumulator`, `DoubleAdder`, `LongAccumulator` и `LongAdder`. В накопительных классах поддерживаются последовательности определяемых пользователем операций. А в суммирующих классах накапливается нарастающая сумма.

Параллельное программирование средствами Fork/Join Framework

В последние годы появилось новое важное направление в разработке программного обеспечения, называемое *параллельным программированием*. Параллельное программирование — это общее название методик, выгодно использующих вычислительные мощности многоядерных процессоров. Как известно, ныне компьютеры с многоядерными процессорами уже стали обычным явлением. К преимуществам многопроцессорных систем относится возможность значительно повысить производительность программного обеспечения. В итоге заметно возросла потребность в механизме, который позволял бы программирующим на Java просто, но эффективно пользоваться несколькими процессорами, без особого труда наращивая вычислительные мощности по мере надобности. В ответ на эту потребность в версии JDK 7 было внедрено несколько новых классов и интерфейсов для поддержки параллельного программирования. Обычно они упоминаются под общим названием *Fork/Join Framework*. Это одно из наиболее важных за последнее время дополнений библиотеки классов Java. Каркас `Fork/Join Framework` определен в пакете `java.util.concurrent`.

Каркас `Fork/Join Framework` усовершенствует многопоточное программирование двумя важными способами. Во-первых, он упрощает создание и использование нескольких потоков исполнения, и во-вторых, автоматизирует использование нескольких процессоров. Иными словами, каркас `Fork/Join Framework` позволяет автоматически наращивать вычислительные мощности в прикладных программах, увеличивая число задействованных процессоров. Благодаря этим двум

усовершенствованиям каркас Fork/Join Framework рекомендуется применять для многопоточного программирования в тех случаях, когда требуется параллельная обработка.

Прежде чем продолжить дальше, следует указать на отличие параллельного программирования от традиционного многопоточного программирования. В прошлом большинство компьютеров имело лишь один процессор, и многопоточность, прежде всего, позволяла выгодно воспользоваться временем простоя, когда программа ожидает, например, от пользователя ввода данных. При таком подходе один поток может выполняться, в то время как другой ожидает. Иными словами, в системе с одним процессором многопоточность позволяет совместно использовать этот процессор для выполнения двух или более задач. Такой тип многопоточности, как правило, поддерживается объектом класса `Thread`, как пояснялось в главе 11. И хотя эта разновидность многопоточности останется весьма полезной и впредь, она не совсем подходит для тех случаев, когда имеются два или более процессора, т.е. многоядерный компьютер.

Если имеется несколько процессоров, то требуется другой тип многопоточности, обеспечивающий настоящее параллельное выполнение. На двух или более процессорах программу можно выполнять одновременно отдельными частями. Благодаря этому значительно ускоряется выполнение некоторых видов операций, включая сортировку, преобразование или поиск в крупном массиве. Зачастую такие операции могут быть разделены на меньшие части, например, для обработки массива по частям на отдельных процессорах. Нетрудно догадаться, что такое распараллеливание операций дает немалый выигрыш в производительности, а следовательно, в будущем параллельное программирование станет неотъемлемой частью арсенала средств каждого программиста, поскольку оно открывает путь к значительному повышению производительности программного обеспечения.

Основные классы Fork/Join Framework

Каркас Fork/Join Framework входит в пакет `java.util.concurrent`. Его ядро составляют следующие классы:

<code>ForkJoinTask<V></code>	Абстрактный класс, определяющий выполняемую задачу
<code>ForkJoinPool</code>	Управляет выполнением задач типа <code>ForkJoinTask</code>
<code>RecursiveAction</code>	Является производным от класса <code>ForkJoinTask<V></code> для выполнения задач, <i>не</i> возвращающих значения
<code>RecursiveTask<V></code>	Является производным от класса <code>ForkJoinTask<V></code> для выполнения задач, возвращающих значения

Рассмотрим взаимосвязь этих классов. Класс `ForkJoinPool` управляет выполнением задачи, представленной объектом класса `ForkJoinTask`. Класс `ForkJoinTask` является абстрактным и расширяется двумя другими абстрактными классами: `RecursiveAction` и `RecursiveTask`. Как правило, эти классы расширяются в прикладном коде для формирования задачи. Прежде чем перейти к подробному рассмотрению процесса выполнения задачи, сделаем краткий обзор основных особенностей каждого из этих классов.

На заметку! Класс `CountedCompleter`, внедренный в версии JDK 8, также расширяет класс `ForkJoinTask`. Но его рассмотрение выходит за рамки данной книги.

Класс `ForkJoinTask<V>`

Класс `ForkJoinTask<V>` является абстрактным и определяет задачу, выполнением которой может управлять объект класса `ForkJoinPool`. Параметр типа `V` определяет тип результата выполнения задачи. Класс `ForkJoinTask` отличается от класса `Thread` тем, что он представляет облегченную абстракцию задачи, а не поток исполнения. Задача типа `ForkJoinTask` выполняется потоками, управляемыми из пула потоков типа `ForkJoinPool`. Такой механизм позволяет управлять выполнением большого количества задач, фактически используя небольшое число потоков исполнения. Таким образом, задачи типа `ForkJoinTask` оказываются намного эффективнее потоков исполнения типа `Thread`.

В классе `ForkJoinTask` определено немало методов. Основными из них являются методы `fork()` и `join()`, общие формы которых представлены ниже.

```
final ForkJoinTask<V> fork()  
final V join()
```

Метод `fork()` передает вызывающую задачу для асинхронного выполнения. Это означает, что поток исполнения, из которого вызывается метод `fork()`, продолжает выполняться. Как только задача будет запланирована для выполнения, метод `fork()` возвратит ссылку `this` на объект этой задачи. До версии JDK 8 это можно было сделать только в вычислительной части другой задачи типа `ForkJoinTask`, которая выполнялась из пула потоков типа `ForkJoinPool`. (Ниже будет показано, каким образом создается вычислительная часть задачи.) Но начиная с версии JDK 8 используется общий пул потоков, если метод `fork()` не вызывается при выполнении задачи из пула потоков типа `ForkJoinPool`. Метод `join()` ожидает завершения задачи, для которой он вызван. В итоге возвращается результат выполнения задачи. Таким образом, с помощью методов `fork()` и `join()` можно запустить на выполнение одну или несколько новых задач и ожидать их завершения.

Еще одним важным в классе `ForkJoinTask` является метод `invoke()`. Он объединяет операции вилочного соединения в единый вызов, поскольку запускает сначала задачу на выполнение, а затем ожидает ее завершения. Ниже приведена общая форма этого метода. Он возвращает результат выполнения вызывающей задачи.

```
final V invoke()
```

С помощью метода `invokeAll()` можно вызвать одновременно несколько задач. Ниже приведены две общие формы этого метода.

```
static void invokeAll(ForkJoinTask<?> задачаA, ForkJoinTask<?> задачаB)  
static void invokeAll(ForkJoinTask<?> ... список_задач)
```

В первой форме выполняются задачи, обозначаемые параметрами *задачаA* и *задачаB*, а во второй форме — все указанные задачи. В обоих случаях вызывающий поток исполнения ожидает завершения всех указанных задач. До версии JDK 8 метод `invokeAll()` можно было вызвать только в вычислительной части другой задачи типа `ForkJoinTask`, которая выполнялась из пула потоков типа

ForkJoinPool. А в версии JDK 8 это ограничение было ослаблено благодаря внедрению общего пула потоков.

Класс RecursiveAction

Этот класс является производным от класса ForkJoinTask и инкапсулирует задачу, которая не возвращает результат. Как правило, класс RecursiveAction расширяется в прикладном коде, чтобы сформировать задачу, возвращающую значение типа void. В классе RecursiveAction определены четыре метода, но только один из них обычно представляет какой-то интерес — это абстрактный метод compute(). Когда класс RecursiveAction расширяется с целью создать конкретный класс, то код, определяющий задачу, размещается в теле метода compute(). Метод compute() представляет *вычислительную* часть задачи. В классе RecursiveAction этот метод определяется следующим образом:

```
protected abstract void compute()
```

Обратите внимание на то, что метод compute() является защищенным. Это означает, что он может быть вызван только другими методами данного класса или производного от него класса. А поскольку этот метод еще и абстрактный, то его следует реализовать в производном классе, если только этот класс также является абстрактным.

Как правило, класс RecursiveAction служит для реализации рекурсивной стратегии выполнения задач, которые не возвращают результаты. Эта стратегия действует по принципу “разделяй и властвуй”, как поясняется в соответствующем разделе далее в этой главе.

Класс RecursiveTask<V>

Еще одним производным от класса ForkJoinTask является класс RecursiveTask<V>. Он инкапсулирует задачу, которая возвращает результат. Тип результата определяется параметром типа V. Как правило, класс RecursiveTask<V> расширяется в прикладном коде, чтобы сформировать задачу, возвращающую значение. Как и в классе RecursiveAction, в данном классе определены четыре метода, но обычно используется только абстрактный метод compute(), представляющий *вычислительную* часть задачи. Когда класс RecursiveTask<V> расширяется для создания конкретного класса, в теле метода compute() размещается код, представляющий выполняемую задачу. Этот код также должен вернуть результат выполнения задачи. В классе RecursiveAction этот метод определяется следующим образом:

```
protected abstract V compute()
```

Обратите внимание на то, что метод compute() является защищенным. Это означает, что он может быть вызван только другими методами данного класса или производного от него класса. А поскольку этот метод еще и абстрактный, то его следует реализовать в производном классе, если только этот класс также является абстрактным. Будучи реализованным, метод compute() должен возвращать результат выполнения задачи.

Как правило, класс `RecursiveTask` служит для реализации рекурсивной стратегии выполнения задач, которые не возвращают результаты. Данная стратегия действует по принципу “разделяй и властвуй”, как поясняется в соответствующем разделе далее в этой главе.

Класс `ForkJoinPool`

Выполнение задач типа `ForkJoinTask` происходит из пула потоков типа `ForkJoinPool`, который управляет также выполнением других задач. Следовательно, чтобы запустить на выполнение задачу типа `ForkJoinTask`, сначала потребуется объект типа `ForkJoinPool`. В версии JDK 8 появились два способа получения объекта типа `ForkJoinPool`. Во-первых, этот объект можно создать явным образом, используя конструктор класса `ForkJoinPool`. И во-вторых, можно воспользоваться так называемым *общим пулом*. Общий пул был внедрен в версии JDK 8 и является статическим объектом типа `ForkJoinPool`, автоматически доступным для применения. Ниже представлены методы из класса `ForkJoinPool`, начиная с построения пула вручную.

В классе `ForkJoinPool` определено несколько конструкторов. Ниже приведены два наиболее употребительных конструктора этого класса.

```
ForkJoinPool ()  
ForkJoinPool (int уровень_параллелизма)
```

Первый конструктор создает пул по умолчанию, обеспечивающий уровень параллелизма, равный количеству процессоров, доступных в системе. А второй конструктор позволяет задать конкретный *уровень_параллелизма*. Значение параметра *уровень_параллелизма* должно быть больше нуля, но не больше предела реализации. Уровень параллелизма определяет количество потоков, которые могут исполняться одновременно. В итоге уровень параллелизма фактически определяет количество задач, которые могут выполняться одновременно. (Разумеется, количество одновременно выполняемых задач не может превышать количество доступных процессоров.) Однако уровень параллелизма *не* ограничивает количество задач, которыми может управлять пул потоков. На самом деле пул потоков типа `ForkJoinPool` может управлять намного большим количеством задач, чем его уровень параллелизма. Кроме того, уровень параллелизма — это лишь цель, а не средство, дающее какую-то гарантию.

Как только будет создан экземпляр класса `ForkJoinPool`, задачу можно запустить на выполнение самыми разными способами. Задача, запускаемая на выполнение первой, обычно считается основной. Эта задача нередко запускает подчиненные задачи, которыми также управляет пул потоков. Самый распространенный способ запустить основную задачу — вызвать метод `invoke()` из класса `ForkJoinPool`. Ниже приведена общая форма этого метода.

```
<T> T invoke (ForkJoinTask<T> задача)
```

Этот метод запускает указанную задачу и возвращает результат ее выполнения. Это означает, что вызывающий код ожидает завершения метода `invoke()`.

Чтобы запустить задачу на выполнение и не ждать ее завершения, можно воспользоваться методом `execute()`. Ниже приведена одна из его форм.

```
void execute (ForkJoinTask<?> задача)
```

В данном случае указанная задача запускается на выполнение, но вызывающий код не ждет ее завершения. Вместо этого вызывающий код продолжает выполняться асинхронно.

Начиная с версии JDK 8 строить объект типа `ForkJoinPool` явным образом совсем не обязательно, поскольку для этой цели имеется общий пул. Как правило, если созданный явным образом пул не используется, то вместо него автоматически выбирается общий пул. Вызвав метод `commonPool()`, определенный в классе `ForkJoinPool`, можно получить ссылку на общий пул, хотя это и необязательно. Ниже приведена общая форма этого метода.

```
static ForkJoinPool commonPool()
```

Этот метод возвращает ссылку на общий пул, обеспечивающий исходный уровень параллелизма. Этот уровень может быть задан с помощью системного свойства. (Подробнее об этом см. в документации на параллельный API.) Как правило, выбираемый по умолчанию общий пул вполне подходит для многих приложений. Разумеется, вам ничто не мешает построить свой пул.

Запустить задачу на выполнение из общего пула можно двумя способами. Во-первых, вызвав метод `commonPool()`, можно получить ссылку на пул, а затем вызвать по этой ссылке метод `invoke()` или `execute()`, как описано выше. И во-вторых, в любой части задачи, кроме вычислительной, можно вызвать метод `fork()` или `invoke()` из класса `ForkJoinTask`. В последнем случае общий пул выбирается автоматически. Иными словами, метод `fork()` или `invoke()` запустит задачу на выполнение из общего пула, если задача еще не выполняется в пуле типа `ForkJoinPool`.

Пул типа `ForkJoinPool` управляет выполнением своих потоков по принципу *перехвата работы*. Каждый рабочий поток исполнения поддерживает очередь задач. Если очередь задач одного рабочего потока исполнения окажется пустой, он возьмет задачу из другого рабочего потока исполнения. Такой принцип способствует повышению общей производительности и помогает равномерно распределять нагрузку. (Вследствие того что время ЦП требуется другим процессам в системе, даже два рабочих потока исполнения с одинаковыми задачами в своих очередях могут и не завершиться одновременно.)

Следует также иметь в виду, что в пуле типа `ForkJoinPool` используются *потокосы демоны*. Потокосый демон автоматически завершается вместе со всеми пользовательскими потоками. Таким образом, нет никакой необходимости явно завершать работу пула типа `ForkJoinPool`. Тем не менее это можно сделать, вызвав метод `shutdown()`. Впрочем, вызов метода `shutdown()` не оказывает никакого влияния на общий пул.

Стратегия “разделяй и властвуй”

Как правило, пользователи каркаса `Fork/Join Framework` пользуются стратегией “разделяй и властвуй”, положенной в основу рекурсии. Именно поэтому оба класса, производных от класса `ForkJoinTask`, называются `RecursiveAction` и `RecursiveTask`. Ожидается, что при формировании своей задачи вилочного соединения программирующие на Java будут расширять один из этих классов.

Стратегия “разделяй и властвуй”, положенная в основу рекурсии, подразумевает разделение задачи на подзадачи до тех пор, пока их объем не станет достаточно мелким для последовательной обработки. Например, задача преобразования каждого из N элементов массива целых чисел может быть разделена на две подзадачи, каждая из которых преобразует половину элементов в массиве. Таким образом, одна подзадача преобразует элементы от 0 до $N/2$, а другая — элементы от $N/2$ до N . В свою очередь, каждая подзадача может быть сведена к набору подзадач, каждая из которых преобразует половину остальных элементов. Этот процесс деления массива продолжается до тех пор, пока не будет достигнуто пороговое значение, при котором последовательное решение задачи оказывается быстрее, чем дальнейшее ее разделение на подзадачи.

Преимущество стратегии “разделяй и властвуй” заключается в том, что обработка может осуществляться параллельно. Поэтому части массива могут быть обработаны одновременно, вместо того чтобы циклически перебирать весь массив в одном потоке. Безусловно, принцип “разделяй и властвуй” действует во многих случаях и без массива (или коллекции), но наиболее распространенная область его применения подразумевает наличие некоторого типа массива, коллекции или другого группирования данных.

Одним из главных условий успешного применения стратегии “разделяй и властвуй” является правильное определение порогового значения, после которого выполняется последовательная обработка, а не дальнейшее разделение задачи. Как правило, оптимальное пороговое значение получается при профилировании характеристик исполнения. Но даже при использовании порогового значения меньше оптимального все равно произойдет весьма существенное ускорение выполнения задачи. Тем не менее лучше избегать чрезмерно крупных или мелких пороговых значений. На момент написания этой книги в документации API Java на класс `ForkJoinTask<T>` приводится следующее эмпирически выведенное правило: задача должна выполняться где-то за 100–10000 этапов вычисления.

Важно также иметь в виду, что на оптимальное пороговое значение влияет время, которое отнимают вычисления. Если каждый этап вычислений достаточно продолжителен, то предпочтительнее устанавливать малые пороговые значения, и, наоборот, если каждый этап вычислений очень короткий, то большие пороговые значения могут обеспечить лучшие результаты. Для прикладных программ, которые должны быть запущены на выполнение в системе с известным количеством процессоров, сведения о количестве процессоров можно использовать для обоснования решения о пороговом значении. Но для прикладных программ, которые будут выполняться в разных системах, возможности которых заранее неизвестны, практически невозможно сделать никаких предположений о вычислительных мощностях исполняющей среды.

И еще одно замечание: несмотря на то что в системе может быть доступно несколько процессоров, другие задачи (и сама операционная система) будут соперничать с прикладной программой за время ЦП. Поэтому не стоит особенно полагаться на то, что у прикладной программы будет неограниченный доступ ко всем имеющимся в системе процессорам. Кроме того, различные процессы в одной той же программе могут показать разные характеристики времени выполнения из-за отличий в загрузженности задачами.

Первый простой пример вилочного соединения

А теперь рассмотрим простой пример, демонстрирующий применение каркаса Fork/Join Framework и стратегии “разделяй и властвуй” на практике. В приведенной ниже программе значения элементов массива типа `double` преобразуются в их квадратные корни. Для этой цели служит класс, производный от класса `RecursiveAction`. Обратите внимание на то, что в данной программе создается свой пул потоков типа `ForkJoinPool`.

```
// Простой пример реализации стратегии "разделяй и властвуй".
// В данном примере применяется класс RecursiveAction
import java.util.concurrent.*;
import java.util.*;

// Класс ForkJoinTask преобразует (через класс RecursiveAction)
// значения элементов массива типа double в их квадратные корни
class SqrtTransform extends RecursiveAction {
    // В данном примере пороговое значение произвольно устанавливается
    // равным 1000. В реальном коде его оптимальное значение может
    // быть определено в результате профилирования исполняющей системы
    // или экспериментально
    final int seqThreshold = 1000;

    // обрабатываемый массив
    double[] data;

    // определить часть обрабатываемых данных
    int start, end;

    SqrtTransform(double[] vals, int s, int e ) {
        data = vals;
        start = s;
        end = e;
    }
    // Этот метод выполняет параллельное вычисление
    protected void compute() {

        // Если количество элементов меньше порогового значения,
        // выполнить дальнейшую обработку последовательно
        if((end - start) < seqThreshold) {
            // преобразовать значение каждого элемента массива
            // в его квадратный корень
            for(int i = start; i < end; i++) {
                data[i] = Math.sqrt(data[i]);
            }
        }
        else {
            // в противном случае продолжить разделение данных на
            // меньшие части

            // найти середину
            int middle = (start + end) / 2;

            // запустить новые подзадачи на выполнение, используя
            // разделенные на части данные
            invokeAll(new SqrtTransform(data, start, middle),
                new SqrtTransform(data, middle, end));
        }
    }
}
```

```

}

// продемонстрировать параллельное выполнение
class ForkJoinDemo {
    public static void main(String args[]) {
        // создать пул задач
        ForkJoinPool fjp = new ForkJoinPool();

        double[] nums = new double[100000];

        // присвоить некоторые значения
        for(int i = 0; i < nums.length; i++)
            nums[i] = (double) i;

        System.out.println("Часть исходной последовательности:");

        for(int i=0; i < 10; i++)
            System.out.print(nums[i] + " ");
        System.out.println("\n");

        SqrtTransform task = new SqrtTransform(nums, 0, nums.length);

        // запустить главную задачу типа ForkJoinTask на выполнение
        fjp.invoke(task);

        System.out.println("Часть преобразованной последовательности " +
            "(с точностью до четырех знаков после десятичной точки):");

        for(int i=0; i < 10; i++)
            System.out.format("%.4f ", nums[i]);
        System.out.println();
    }
}

```

Ниже приведен результат, выводимый данной программой. Как можно заметить, значения элементов массива были преобразованы в их квадратные корни.

Часть исходной последовательности:
0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0

Часть преобразованной последовательности (с точностью до четырех знаков после десятичной точки):
0.0000 1.0000 1.4142 1.7321 2.0000 2.2361 2.4495 2.6458 2.8284 3.0000

Рассмотрим данную программу подробнее. Прежде всего обратите внимание на то, что класс `SqrtTransform` расширяет класс `RecursiveAction`. Как упоминалось ранее, класс `RecursiveAction` расширяет класс `ForkJoinTask` для выполнения задач, которые не возвращают результаты. Затем обратите внимание на конечную переменную `seqThreshold`. Ее значение определяет, когда будет иметь место последовательная обработка. Это значение устанавливается (несколько произвольно) равным 1000. Далее обратите внимание на то, что ссылка на обрабатываемый массив сохраняется в переменной `data` и что поля `start` и `end` служат для указания границ доступности элементов массива.

Основное действие программы происходит в методе `compute()`. Оно начинается с проверки соответствия количества обрабатываемых элементов нижнему пороговому значению для последовательной обработки. Если такое соответствие обнаружено, то эти элементы обрабатываются (в данном примере из их значе-

ний извлекается квадратный корень). А если пороговое значение для последовательной обработки не достигнуто, то вызывается метод `invokeAll()` для запуска на выполнение двух новых подзадач. В данном случае в каждой подзадаче обрабатывается половина элементов массива. Как пояснялось ранее, метод `invokeAll()` ожидает возврата результатов завершения обеих подзадач. По завершении всех рекурсивных вызовов каждый элемент в массиве будет изменен, причем большая часть рассматриваемого здесь действия выполняется параллельно, при условии, что в системе имеется несколько процессоров.

Как упоминалось ранее, начиная с версии JDK 8 строить пул типа `ForkJoinPool` явным образом совсем не обязательно, поскольку для этой цели автоматически доступен общий пул. Кроме того, пользоваться общим пулом намного проще. В частности, вызвав статический метод `commonPool()`, определяемый в классе `ForkJoinPool`, можно получить ссылку на общий пул. Следовательно, приведенный выше пример программы можно переделать, чтобы воспользоваться общим пулом, заменив вызов конструктора класса `ForkJoinPool` на вызов метода `commonPool()`:

```
ForkJoinPool fjp = ForkJoinPool.commonPool();
```

С другой стороны, получать явным образом ссылку на общий пул нет нужды, поскольку вызов метода `invoke()` или `fork()` из класса `ForkJoinTask` для задачи, которая уже не является частью пула, приведет к тому, что она будет автоматически выполнена в общем пуле. Так, из предыдущего примера программы можно полностью исключить переменную `fjp` и запустить задачу на выполнение, воспользовавшись следующей строкой кода:

```
task.invoke();
```

Как упоминалось выше, общий пул относится к усовершенствованиям, внесенным в каркас `Fork/Join Framework` в версии JDK 8 с целью упростить его применение. Более того, применение общего пула оказывается зачастую более предпочтительным, если принять во внимание, что для этого не требуется совместимость с версией JDK 7.

Влияние уровня параллелизма

Прежде чем завершить рассматриваемую здесь тему, важно рассмотреть влияние уровня параллелизма на эффективность выполнения задачи вилочного соединения, а также взаимосвязь между параллелизмом и пороговым значением. Пример программы, приведенный в этом разделе, позволяет экспериментировать с различными уровнями параллелизма и пороговыми значениями. Используя многоядерный компьютер, можно наблюдать в интерактивном режиме результат изменения этих значений.

В предыдущем примере использовался уровень параллелизма по умолчанию. Но ничто не мешает задать требуемый уровень параллелизма. Это можно, в частности, сделать, создав объект типа `ForkJoinPool` с помощью следующего конструктора:

```
ForkJoinPool(int уровень_параллелизма)
```


где параметр *уровень_параллелизма* обозначает устанавливаемый уровень параллелизма. Его значение должно быть больше нуля, но меньше предела реализации.

В приведенном ниже примере программы формируется задача вилочного соединения, в ходе которой преобразуется массив типа `double`. Преобразование выполняется произвольно, но оно организовано таким образом, чтобы потреблять несколько циклов процессора. Это сделано для большей наглядности эффекта от изменения порогового значения или уровня параллелизма. Запуская программу на выполнение, следует указать пороговое значение и уровень параллелизма в командной строке. В итоге программа запустит задачи на выполнение, отображая время, расходуемое на выполнение каждой задачи. Для этой цели вызывается метод `System.nanoTime()`, возвращающий значение высокоточного таймера виртуальной машины JVM.

```
// Простой пример программы, позволяющий экспериментировать с
// эффектом от изменения порогового значения и уровня параллелизма
// выполнения задач в классе ForkJoinTask
import java.util.concurrent.*;

// Класс ForkJoinTask преобразует (через класс RecursiveAction)
// элементы массива типа double
class Transform extends RecursiveAction {

    // Порог последовательного выполнения,
    // устанавливаемый конструктором
    int seqThreshold;

    // Обрабатываемый массив
    double[] data;

    // определить часть обрабатываемых данных
    int start, end;

    Transform(double[] vals, int s, int e, int t) {
        data = vals;
        start = s;
        end = e;
        seqThreshold = t;
    }

    // Этот метод выполняет параллельное вычисление
    protected void compute() {

        // выполнить далее обработку последовательно,
        // если количество элементов ниже порогового значения
        if((end - start) < seqThreshold) {
            // В следующем фрагменте кода элементу по четному
            // индексу присваивается квадратный корень его
            // первоначального значения, а элементу по нечетному
            // индексу – кубический корень его первоначального значения.
            // Этот код предназначен только для потребления времени ЦП,
            // чтобы сделать нагляднее эффект от параллельного выполнения
            for(int i = start; i < end; i++) {
                if((data[i] % 2) == 0)
                    data[i] = Math.sqrt(data[i]);
                else
                    data[i] = Math.cbrt(data[i]);
            }
        }
    }
}
```

```

    }
}
else {
    // В противном случае продолжить разделение данных на
    // меньшие части

    // найти середину
    int middle = (start + end) / 2;

    // запустить новые подзадачи на выполнение,
    // используя разделенные на части данные
    invokeAll(new Transform(data, start, middle, seqThreshold),
              new Transform(data, middle, end, seqThreshold));
}
}
}

// Продемонстрировать параллельное выполнение
class FJExperiment {

    public static void main(String args[]) {
        int pLevel;
        int threshold;

        if(args.length != 2) {
            System.out.println(
                "Использование: FJExperiment параллелизм порог ");
            return;
        }

        pLevel = Integer.parseInt(args[0]);
        threshold = Integer.parseInt(args[1]);

        // Эти переменные используются для измерения
        // времени выполнения задачи
        long beginT, endT;

        // Создать пул задач.
        // Обратите внимание на установку уровня параллелизма
        ForkJoinPool fjp = new ForkJoinPool(pLevel);

        double[] nums = new double[1000000];

        for(int i = 0; i < nums.length; i++)
            nums[i] = (double) i;

        Transform task =
            new Transform(nums, 0, nums.length, threshold);

        // начать измерение времени выполнения задачи
        beginT = System.nanoTime();

        // запустить главную задачу типа ForkJoinTask
        fjp.invoke(task);

        // завершить измерение времени выполнения задачи
        endT = System.nanoTime();

        System.out.println("Уровень параллелизма: " + pLevel);
    }
}

```

```

        System.out.println(
            "Порог последовательной обработки: " + threshold);
        System.out.println(
            "Истекшее время: " + (endT - beginT) + " нс");
        System.out.println();
    }
}

```

Чтобы воспользоваться данной программой, следует указать уровень параллелизма и порог. Разные значения каждого из этих параметров можно опробовать экспериментально, наблюдая за результатами. Не следует, однако, забывать, что данную программу нужно запускать на компьютере, по крайней мере, с двумя процессорами. И даже выполнение этой программы два раза подряд на одном и том же компьютере скорее всего приведет к разным результатам из-за наличия в системе других процессов, потребляющих время ЦП.

Чтобы получить некоторое представление о влиянии параллелизма на производительность программы, сделайте такой эксперимент. Сначала запустите программу по следующей команде:

```
java FJExperiment 1 1000
```

По этой команде устанавливается уровень параллелизма 1 (по существу, последовательное выполнение) и пороговое значение 1000. Ниже приведен примерный результат выполнения данной программы на двухъядерном компьютере.

```

Уровень параллелизма: 1
Порог последовательной обработки: 1000
Истекшее время: 259677487 нс

```

А теперь укажите уровень параллелизма 2 в следующей команде:

```
java FJExperiment 2 1000
```

Ниже приведен примерный результат выполнения данной программы на том же самом двухъядерном компьютере.

```

Уровень параллелизма: 2
Порог последовательной обработки: 1000
Истекшее время: 169254472нс

```

Совершенно очевидно, что благодаря параллелизму значительно уменьшается время выполнения, а следовательно, повышается быстродействие программы. Поэкспериментируйте с изменением порога последовательной обработки и уровня параллелизма на своем компьютере. Полученные результаты могут удивить вас.

Имеются еще два метода, которые могут оказаться полезными для экспериментирования с временными характеристиками выполнения программы, демонстрирующей вилочное соединение. Во-первых, вызвав метод `getParallelism()`, определяемый в классе `ForkJoinPool`, можно получить уровень параллелизма. Ниже приведена общая форма этого метода.

```
int getParallelism()
```

Этот метод возвращает текущий уровень параллелизма. Напомним, что по умолчанию он будет равен количеству доступных процессоров. (Чтобы получить уровень параллелизма для общего пула, можно также вызвать метод `get`

`CommonPoolParallelism()`, внедренный в версии JDK 8.) И во-вторых, вызвав метод `availableProcessors()`, определяемый в классе `Runtime`, можно получить количество процессоров, доступных в системе. Ниже приведена общая форма этого метода. В связи с наличием других системных запросов возвращаемое значение может отличаться после каждого вызова этого метода.

```
int availableProcessors()
```

Пример применения класса `RecursiveTask<V>`

Два приведенных выше примера основаны на классе `RecursiveAction`, а это означает, что в них одновременно выполняются задачи, которые не возвращают результаты. Чтобы сформировать задачу, возвращающую результат, следует воспользоваться классом `RecursiveTask`. В общем, параллельное выполнение задачи, возвращающей результат, организуется так, как было показано выше. Основное отличие состоит в том, что метод `compute()` возвращает результат. Поэтому нужно накопить результаты, чтобы по завершении первого вызова был возвращен общий результат. Другое отличие состоит в том, что запуск подзадачи на выполнение обычно делается явно путем вызова метода `fork()` или `join()`, а не неявно путем вызова, например, метода `invokeAll()`.

В приведенном ниже примере программы демонстрируется применение класса `RecursiveTask`. В этой программе формируется задача `Sum`, возвращающая сумму значений элементов в массиве типа `double`. В данном примере массив состоит из 5000 элементов. Но значение каждого второго элемента этого массива отрицательное. Таким образом, первые элементы данного массива будут иметь значения 0, -1, 2, -3, 4 и т.д. (Для совместимости данной программы с версиями JDK 7 и JDK 8 в ней создается свой пул. В качестве упражнения можете попробовать воспользоваться общим пулом.)

```
// Простой пример применения класса RecursiveTask<V>
import java.util.concurrent.*;

// Класс RecursiveTask, используемый для вычисления суммы
// значений элементов в массиве типа double
class Sum extends RecursiveTask<Double> {

    // Пороговое значение последовательного выполнения
    final int seqThreshold = 500;

    // Обрабатываемый массив
    double[] data;

    // определить часть обрабатываемых данных
    int start, end;

    Sum(double[] vals, int s, int e) {
        data = vals;
        start = s;
        end = e;
    }

    // определить сумму значений элементов в массиве типа double
```

```

protected Double compute() {
    double sum = 0;

    // Если количество элементов ниже порогового значения,
    // то выполнить далее обработку последовательно
    if((end - start) < seqThresHold) {
        // суммировать значения элементов в массиве типа double
        for(int i = start; i < end; i++) sum += data[i];
    }
    else {
        // В противном случае продолжить разделение данных на
        // меньшие части

        // найти середину
        int middle = (start + end) / 2;

        // запустить новые подзадачи на выполнение, используя
        // разделенные на части данные
        Sum subTaskA = new Sum(data, start, middle);
        Sum subTaskB = new Sum(data, middle, end);

        // запустить каждую подзадачу путем разветвления
        subTaskA.fork();
        subTaskB.fork();

        // ожидать завершения подзадач и накопить результаты
        sum = subTaskA.join() + subTaskB.join();
    }
    // вернуть конечную сумму
    return sum;
}

// Продемонстрировать параллельное выполнение
class RecurTaskDemo {
    public static void main(String args[]) {
        // создать пул задач
        ForkJoinPool fjp = new ForkJoinPool();

        double[] nums = new double[5000];

        // инициализировать массив nums чередующимися
        // положительными и отрицательными значениями
        for(int i=0; i < nums.length; i++)
            nums[i] = (double) ((i%2) == 0 ? i : -i) ;

        Sum task = new Sum(nums, 0, nums.length);

        // Запустить задачи типа ForkJoinTask. Обратите
        // внимание на то, что в данном случае метод invoke()
        // возвращает результат
        double summation = fjp.invoke(task);

        System.out.println("Суммирование " + summation);
    }
}

```

Эта программа выводит следующий результат:

Суммирование -2500.0

У данной программы имеются некоторые интересные особенности. Прежде всего обратите внимание на то, что для выполнения обеих подзадач вызывается метод `fork()`:

```
subTaskA.fork();
subTaskB.fork();
```

В данном случае метод `fork()` используется потому, что он запускает подзадачу на выполнение, но не ожидает ее завершения. (Таким образом, он запускает подзадачу на выполнение асинхронно.) А для получения результата выполнения каждой подзадачи вызывается метод `join()`, как показано ниже.

```
sum = subTaskA.join() + subTaskB.join();
```

В данной строке кода ожидается завершение каждой подзадачи, а затем полученные результаты суммируются и присваиваются переменной `sum`. Таким образом, результаты выполнения всех подзадач складываются в вычисляемую итоговую сумму. И наконец, метод `compute()` завершается, возвращая значение переменной `sum`, которое станет окончательной итоговой суммой после возврата из первого вызова.

Имеются и другие способы асинхронного выполнения подзадач. Например, в следующем фрагменте кода метод `fork()` вызывается для запуска подзадачи `subTaskA`, а метод `invoke()` — для запуска и ожидания завершения подзадачи `subTaskB`:

```
subTaskA.fork();
sum = subTaskA.join() + subTaskB.invoke();
```

В качестве еще одного варианта можно непосредственно вызвать метод `compute()` для подзадачи `subTaskB`, как показано ниже.

```
subTaskA.fork();
sum = subTaskA.join() + subTaskB.compute();
```

Асинхронное выполнение задач

Для инициализации задачи в приведенных ранее примерах программ вызывался метод `invoke()` из класса `ForkJoinPool`. Это общепринятый подход, когда вызывающий поток исполнения должен ожидать завершения задачи (что зачастую и бывает), поскольку метод `invoke()` не завершится до тех пор, пока не завершится задача. Но задачу можно запустить на выполнение асинхронно. При таком подходе вызывающий поток продолжает выполняться. Таким образом, вызывающий поток и задача выполняются одновременно. Чтобы запустить задачу на выполнение асинхронно, следует вызвать метод `execute()`, который также определяется в классе `ForkJoinPool`. Ниже приведены две его общие формы.

```
void execute(ForkJoinTask<?> задача)
void execute(Runnable задача)
```

В обеих формах этого метода задается выполняемая задача. Обратите внимание на то, что вторая форма позволяет определить задачу типа `Runnable`, а не `ForkJoinTask`. Этим наводится своего рода мост между традиционным подходом к многопоточности в Java и новым каркасом `Fork/Join Framework`. Следует, одна-

ко, иметь в виду, что потоки исполнения, используемые пулом типа `ForkJoinPool`, являются потоковыми демонами. Следовательно, они завершаются по окончании основного потока исполнения. Это означает, что основной поток исполнения, возможно, придется поддерживать в активном состоянии до тех пор, пока не завершатся все задачи.

Отмена задачи

Вызвав метод `cancel()`, определенный в классе `ForkJoinTask`, можно отменить задачу. Ниже приведена общая форма этого метода.

```
boolean cancel(boolean прерывание)
```

Этот метод возвращает логическое значение `true`, если задача, для которой он был вызван, успешно отменена, или логическое значение `false`, если задача уже отменена, завершена или не может быть отменена. В настоящее время параметр *прерывание* не используется в стандартной реализации. Как правило, метод `cancel()` вызывается из кода за пределами задачи, поскольку задача может легко отменить себя путем возврата.

Вызвав метод `isCancelled()`, можно выяснить, была ли задача отменена:

```
final boolean isCancelled()
```

Этот метод возвращает логическое значение `true`, если вызывающая задача была отменена до ее завершения, а иначе — логическое значение `false`.

Определение состояния завершения задачи

Кроме только что описанного метода `isCancelled()`, класс `ForkJoinTask` содержит два других метода, позволяющих определить состояние завершения задачи. Первым из них является метод `isCompletedNormally()`, общая форма которого приведена ниже.

```
final boolean isCompletedNormally()
```

Этот метод возвращает логическое значение `true`, если вызывающая задача завершилась нормально, т.е. не сгенерировала исключение, и не была отменена в результате вызова метода `cancel()`, а иначе — логическое значение `false`.

Вторым является метод `isCompletedAbnormally()`, общая форма которого приведена ниже.

```
final boolean isCompletedAbnormally()
```

Этот метод возвращает логическое значение `true`, если вызывающая задача завершилась не нормально, а вследствие отмены или генерирования исключения. В противном случае возвращается логическое значение `false`.

Перезапуск задачи

Обычно перезапустить задачу нельзя. Иными словами, как только задача завершится, она не может быть перезапущена. Тем не менее после завершения задачи

ее состояние можно еще раз инициализировать таким образом, чтобы снова запустить ее на выполнение. Для этого достаточно вызвать метод `reinitialize()`, как показано далее.

```
void reinitialize()
```

Этот метод устанавливает вызывающую задачу в исходное состояние. Но любые изменения, внесенные в какие угодно данные постоянного хранения, обрабатываемые в задаче, не будут отменены. Так, если в задаче изменяется массив, это изменение не будет отменено в результате вызова метода `reinitialize()`.

Предмет дальнейшего изучения

Выше были затронуты лишь самые основы каркаса Fork/Join Framework и наиболее употребительные методы. Но Fork/Join Framework — это функционально насыщенный каркас, предоставляющий дополнительные возможности для расширенного управления параллельностью. Обсуждение всех вопросов и особенностей параллельного программирования и применения каркаса Fork/Join Framework выходит за рамки данной книги, тем не менее ниже рассматриваются другие средства, избранные из этого каркаса.

Другие избранные средства из класса ForkJoinTask

Как упоминалось выше, такие методы, как `invokeAll()` и `fork()`, могут быть вызваны только из класса `ForkJoinTask`. Это правило особенно важно для версии JDK 7, где общий пул не поддерживается. Зачастую соблюсти это правило нетрудно, но иногда приходится иметь дело с кодом, способным выполняться как в самой задаче, так и за ее пределами. Поэтому, вызвав метод `inForkJoinPool()`, можно выяснить, выполняется ли код в пределах задачи.

С помощью метода `adapt()`, определяемого в классе `ForkJoinTask`, можно преобразовать объект типа `Runnable` или `Callable` в объект типа `ForkJoinTask`. У этого метода имеются три общие формы: одна — для преобразования объекта типа `Callable`, другая — для объекта типа `Runnable`, не возвращающего результат; и третья — для объекта типа `Runnable`, возвращающего результат. Если это объект типа `Callable`, то выполняется метод `call()`, а если это объект типа `Runnable` — метод `run()`.

Вызвав метод `getQueuedTaskCount()`, можно получить приблизительное количество задач в очереди вызывающего потока исполнения, а вызвав метод `getSurplusQueuedTaskCount()` — количество таких задач, превышающее количество других потоков исполнения в пуле, которые могли бы перехватить их. Напомним, перехват работы (в данном случае задачи) в каркасе Fork/Join Framework — это единственный способ добиться высокой эффективности. И хотя этот процесс происходит автоматически, иногда сведения о нем могут оказаться полезными для оптимизации производительности.

В классе `ForkJoinTask` определяются приведенные ниже варианты методов `join()` и `invoke()`, имена которых начинаются с префикса `quietly`. По существу, эти методы подобны своим упрощенным аналогам, за исключением того, что они не возвращают значений и не генерируют исключений.

final void quietlyJoin()	Соединяет задачу, но не возвращает результат и не генерирует исключение
final void quietlyInvoke()	Вызывает задачу, но не возвращает результат и не генерирует исключение

Вызвав метод `tryUnfork()`, можно попытаться “отменить вызов” задачи. Иными словами, исключить ее из плана выполнения.

В версии JDK 8 было внедрено несколько методов, поддерживающих дескрипторы, в том числе методы `getForkJoinTaskTag()` и `setForkJoinTaskTag()`. Дескрипторы представляют собой короткие целочисленные значения, связанные с задачами. Они могут оказаться полезными в особых случаях.

Класс `ForkJoinTask` реализует интерфейс `Serializable`. Таким образом, его объект может быть сериализован. Но сериализация не употребляется во время выполнения.

Другие избранные средства из класса `ForkJoinPool`

К числу методов, которые оказываются весьма полезными для настройки приложений вилочных соединений, относится метод `toString()`, переопределяемый в классе `ForkJoinPool`. Этот метод выводит в удобной для пользователя форме отчет о состоянии пула. Чтобы опробовать его на практике, введите приведенный ниже фрагмент кода для запуска и последующего ожидания задачи в класс `FJExperiment` из представленного ранее примера программы экспериментирования с задачами.

```
// Запустить главную задачу типа ForkJoinTask асинхронно
fjp.execute(task);

// отобразить состояние пула во время ожидания
while(!task.isDone()) {
    System.out.println(fjp);
}
```

Запустив эту программу на выполнение, вы увидите на экране ряд сообщений, описывающих состояние пула. Ниже приведен один из примеров такого вывода. Безусловно, у вас вывод может быть другим из-за отличий в количестве процессоров, пороговых значений, загруженности задачами и т.д.

```
java.util.concurrent.ForkJoinPool@141d683[Running, parallelism = 2,
size = 2, active = 0, running = 2, steals = 0, tasks = 0, submissions = 1]
```

Вызвав метод `isQuiescent()`, можно выяснить, не бездействует ли пул в настоящий момент. Этот метод возвращает логическое значение `true`, если в пуле отсутствуют активные потоки, а иначе — логическое значение `false`. Вызвав метод `getPoolSize()`, можно получить количество рабочих потоков исполнения, находящихся в настоящий момент в пуле, а вызвав вызов метод `getActiveThreadCount()` — приблизительное количество активных потоков исполнения в пуле.

Чтобы закрыть пул, следует вызвать метод `shutdown()`. Текущие задачи все еще будут выполняться, но никаких новых задач запущено не будет. Чтобы закрыть пул немедленно, нужно вызвать метод `shutdownNow()`. В данном случае делается попытка отменить текущие задачи. Следует, однако, иметь в виду, что ни один из

упоминаемых здесь методов не оказывает влияния на общий пул. Вызвав метод `isShutdown()`, можно выяснить, закрыт ли пул. Этот метод возвращает логическое значение `true`, если пул закрыт, а иначе — логическое значение `false`. И наконец, чтобы выяснить, закрыт ли пул и все ли задачи завершены, следует вызвать метод `isTerminated()`.

Рекомендации относительно вилочного соединения

В этом разделе даются некоторые рекомендации, помогающие обойти наиболее трудные препятствия, связанные с применением каркаса Fork/Join Framework. Во-первых, старайтесь не указывать слишком низкое пороговое значение для последовательного выполнения задач. Обычно лучше ошибиться в большую, чем в меньшую сторону. Если пороговое значение слишком мало, на формирование и переключение задач может уйти времени больше, чем на их обработку. Во-вторых, обычно лучше выбирать уровень параллелизма, устанавливаемый по умолчанию. Если же указать меньший уровень параллелизма, это может в значительной степени свести на нет все преимущества, которые дает применение каркаса Fork/Join Framework.

Обычно в задаче типа `ForkJoinTask` не должны применяться синхронизированные методы или блоки кода. Кроме того, метод `compute()` обычно применяется вместе с другими средствами синхронизации, например семафорами. (Тем не менее можно воспользоваться новым классом `Phaser`, поскольку он совместим с механизмом вилочного соединения.) Напомним, что в основу класса `ForkJoinTask` положена стратегия “разделяй и властвуй”. Но такой подход обычно не применяется в тех случаях, когда требуется внешняя синхронизация. Кроме того, старайтесь избегать ситуаций, когда ввод-вывод может привести к длительной блокировке. В подобных случаях класс `ForkJoinTask` обычно не обслуживает ввод-вывод. Проще говоря, задача должна выполнять вычисление, которое может быть организовано без внешней блокировки или синхронизации. Это позволит лучше воспользоваться преимуществами каркаса Fork/Join Framework.

И последнее замечание: за исключением необычных обстоятельств не делайте никаких предположений относительно среды выполнения, в которой будет работать написанный вами прикладной код. Это означает, что вы не должны предполагать, что будет доступно конкретное количество процессоров или что на характеристики выполнения вашей прикладной программы не будут оказывать влияние другие одновременно выполняющиеся процессы.

Утилиты параллелизма в сравнении с традиционным подходом к многозадачности в Java

Принимая во внимание эффективность и гибкость утилит параллелизма, естественно задаться следующим вопросом: заменяют ли они собой традиционный подход в многозадачности, принятый в Java? Безусловно, не заменяют!

Первоначальная поддержка многозадачности и встроенные средства синхронизации по-прежнему должны применяться во многих прикладных программах на Java, апплетах и сервлетах. Например, ключевое слово `synchronized`, а также методы `wait()` и `notify()` предоставляют изящные решения широкого ряда задач многопоточной обработки. Но когда требуется дополнительное управление потоками исполнения, на помощь приходят утилиты параллелизма. Кроме того, в каркасе `Fork/Join Framework` предоставляется эффективный способ, позволяющий интегрировать методики параллельного программирования в более сложные прикладные программы.

Среди многих новых средств, внедренных в версии JDK 8, особое место, безусловно, принадлежит лямбда-выражениям и потоковому прикладному программному интерфейсу API. Лямбда-выражения были описаны в главе 15, а в этой главе рассматривается потоковый API. Как станет ясно из материала этой главы, потоковый API разработан с учетом лямбда-выражений. Более того, потоковый API наглядно демонстрирует, насколько лямбда-выражения повышают эффективность Java.

Несмотря на впечатляющую совместимость потокового API с лямбда-выражениями, главной его особенностью является способность выполнять очень сложные операции поиска, фильтрации, преобразования и иного манипулирования данными. Используя потоковый API, можно, например, сформировать последовательности действий, принципиально подобных запросам базы данных, составленным на языке SQL. Более того, подобные действия, как правило, выполняются параллельно, а следовательно, они повышают производительность, особенно при обработке крупных массивов данных. Проще говоря, потоковый API предоставляет мощные средства для обработки данных эффективным, но простым способом.

Прежде всего следует заметить, что в потоковом API применяются одни из самых развитых языковых средств Java. Поэтому для полного уяснения и применения этого прикладного программного интерфейса потребуются твердое знание обобщений и лямбда-выражений, а также основных принципов параллельного выполнения и применения каркаса коллекций Collections Framework (подробнее об этом см. в главах 14, 15, 18 и 28 соответственно).

Основные положения о потоках данных

Прежде всего, само понятие *поток данных* в потоковом API определяется как канал передачи данных. Следовательно, поток данных представляет собой последовательность объектов. Поток данных оперирует источником данных, например массивом или коллекцией. В самом потоке данные не хранятся, а только перемещаются и, возможно, фильтруются, сортируются или обрабатываются иным образом в ходе этого процесса. Но, как правило, действие самого потока данных не видоизменяет их источник. Например, сортировка данных в потоке не изменяет их упорядочение в источнике, а скорее приводит к созданию нового потока данных, дающего отсортированный результат.

На заметку! Следует особо подчеркнуть, что употребляемое здесь понятие *поток данных* отличается от аналогичного понятия, употреблявшегося при описании классов ввода-вывода ранее в данной книге. Несмотря на то что обычные потоки ввода-вывода могут в принципе действовать почти так же, как и потоки данных, определяемые в пакете `java.util.stream`, они не одинаковы. Поэтому под термином *поток данных*, употребляемым в этой главе, подразумеваются объекты, основывающиеся на одном из описываемых здесь типов потоков.

Потоковые интерфейсы

В потоковом API определяется ряд потоковых интерфейсов, входящих в состав пакета `java.util.stream`. В основание их иерархии положен интерфейс `BaseStream`, в котором определяются основные функциональные возможности всех потоков данных. Интерфейс `BaseStream` является обобщенным и определяется следующим образом:

```
interface BaseStream<T, S extends BaseStream<T, S>>
```

где параметр `T` обозначает тип элементов в потоке данных, а параметр `S` — тип потока данных, расширяющего интерфейс `BaseStream`. В свою очередь, интерфейс `BaseStream` расширяет интерфейс `AutoCloseable`, а следовательно, потоком данных можно управлять в блоке оператора `try` с ресурсами. Но, как правило, закрывать приходится только те потоки данных, где источники данных требуют закрытия (например, те потоки, которые связаны с файлами). В то же время потоки, источниками данных для которых обычно служат коллекции, закрывать не нужно. Методы, объявляемые в интерфейсе `BaseStream`, перечислены в табл. 29.1.

Таблица 29.1. Методы из интерфейса `BaseStream`

Метод	Описание
<code>void close()</code>	Закрывает вызывающий поток данных, вызывая любые зарегистрированные обработчики событий закрытия. (Как пояснялось ранее, закрывать требуется лишь немногие потоки данных.)
<code>boolean isParallel()</code>	Возвращает логическое значение <code>true</code> , если вызывающий поток данных является параллельным, или логическое значение <code>false</code> , если он является последовательным
<code>Iterator<T> iterator()</code>	Получает итератор для потока данных и возвращает ссылку на него. (Это оконечная операция.)
<code>S onClose(Runnable обработчик)</code>	Возвращает новый поток данных с указанным обработчиком событий закрытия. Указанный обработчик вызывается при закрытии потока данных. (Это промежуточная операция.)
<code>S parallel()</code>	Возвращает параллельный поток данных, исходя из вызывающего потока данных. Если вызывающий поток данных уже является параллельным, то именно он и возвращается. (Это промежуточная операция.)
<code>S sequential()</code>	Возвращает последовательный поток данных, исходя из вызывающего потока данных. Если вызывающий поток данных уже является последовательным, то именно он и возвращается. (Это промежуточная операция.)

Окончание табл. 29.1

Метод	Описание
Splitterator<T> spliterator()	Получает итератор-разделитель для потока данных и возвращает ссылку на него. (Это оконечная операция.)
S unordered()	Возвращает неупорядоченный поток данных, исходя из вызывающего потока данных. Если вызывающий поток данных уже является неупорядоченным, то именно он и возвращается. (Это промежуточная операция.)

Производными от интерфейса `BaseStream` являются несколько типов интерфейсов. Наиболее употребительным из них является интерфейс `Stream`. Он объявляется следующим образом:

```
interface Stream<T>
```

где параметр `T` обозначает тип элементов в потоке данных. Интерфейс `Stream` является обобщенным и поэтому пригоден для всех ссылочных типов. Помимо методов, наследуемых из интерфейса `BaseStream`, в интерфейсе `Stream` определяются собственные методы. Некоторые из них перечислены в табл. 29.2.

Таблица 29.2. Избранные методы из интерфейса `Stream`

Метод	Описание
<R, A> R collect(Collector<? super T, A, R> <i>функция_накопления</i>)	Накапливает элементы в изменяемом контейнере и возвращает этот контейнер. Называется операцией изменяемого сведения. Здесь <code>R</code> обозначает тип результирующего контейнера; <code>T</code> — тип элемента из вызывающего потока данных, тогда как <code>A</code> — внутренний накапливаемый тип. Указанная <i>функция_накопления</i> обозначает порядок выполнения процесса накопления (Это оконечная операция.)
long count()	Подсчитывает количество элементов в потоке данных и возвращает полученный результат. (Это оконечная операция.)
Stream<T> filter(Predicate<? super T> <i>предикат</i>)	Производит поток данных, содержащий те элементы из вызывающего потока данных, которые удовлетворяют указанному <i>предикату</i> (Это промежуточная операция.)
void forEach(Consumer<? super T> <i>действие</i>)	Выполняет код, обозначаемый указанным <i>действием</i> , для каждого элемента из вызывающего потока данных. (Это оконечная операция.)
<R> Stream<R> map(Function<? super T, ? extends R> <i>функция_отображения</i>)	Применяет указанную <i>функцию_отображения</i> к элементам из вызывающего потока данных, производя новый поток данных, содержащий эти элементы. (Это промежуточная операция.)

Метод	Описание
DoubleStream mapToDouble (ToDoubleFunction <? super T > <i>функция_отображения</i>)	Применяет указанную <i>функцию_отображения</i> к элементам из вызывающего потока данных, производя новый поток данных типа DoubleStream , содержащий эти элементы. (Это промежуточная операция.)
IntStream mapToInt (ToIntFunction <? super T > <i>функция_отображения</i>)	Применяет указанную <i>функцию_отображения</i> к элементам из вызывающего потока данных, производя новый поток данных типа IntStream , содержащий эти элементы. (Это промежуточная операция.)
LongStream mapToLong (ToLongFunction <? super T > <i>функция_отображения</i>)	Применяет указанную <i>функцию_отображения</i> к элементам из вызывающего потока данных, производя новый поток данных типа LongStream , содержащий эти элементы. (Это промежуточная операция.)
Optional < T > max (Comparator <? super T > <i>компаратор</i>)	Обнаруживает и возвращает максимальный элемент в вызывающем потоке данных, используя упорядочение, определяемое указанным <i>компаратором</i> . (Это окончательная операция.)
Optional < T > min (Comparator <? super T > <i>компаратор</i>)	Обнаруживает и возвращает минимальный элемент в вызывающем потоке данных, используя упорядочение, определяемое указанным <i>компаратором</i> . (Это окончательная операция.)
T reduce (T <i>identityVal</i> , BinaryOperator < T > <i>накопитель</i>)	Возвращает результат, исходя из элементов в вызывающем потоке данных. Называется операцией сведения (Это окончательная операция.)
Stream < T > sorted ()	Производит новый поток данных, содержащий элементы из вызывающего потока данных, отсортированные в естественном порядке (Это промежуточная операция.)
Object [] toArray ()	Создает массив из элементов в вызывающем потоке данных (Это окончательная операция.)

Обратите внимание на то, что в обеих приведенных выше таблицах методы обозначены как *оконечные* или *промежуточные* операции. Их отличие состоит в том, что окончательная операция *потребляет* поток данных и дает конечный результат, например, обнаруживает минимальное значение в потоке данных или выполняет некоторое действие, как это делает метод `forEach()`. Если поток данных потреблен, он не может быть использован повторно. А промежуточная операция *производит* поток данных и служит для создания *конвейера* для выполнения последовательности действий. Кроме того, промежуточные операции не выполняются немедленно. Напротив, указанное действие происходит в том случае, когда окончательная операция выполняется в новом потоке данных, созданном промежуточной операцией. Такой механизм называется *отложенным поведением*, а промежуточные

операции — *отложенными*. Благодаря отложенному поведению потоковый API действует более эффективно.

Еще одна особенность потоков данных состоит в том, что одни промежуточные операции выполняются *без сохранения состояния*, а другие — *с сохранением состояния*. В операции без сохранения состояния каждый элемент обрабатывается независимо от остальных. А в операции с сохранением состояния обработка элемента может зависеть от особенностей остальных элементов. Например, сохранение данных является операцией с сохранением состояния, поскольку упорядочение элемента зависит от значений других элементов. Следовательно, метод `sorted()` выполняется с сохранением состояния. Но фильтрация элементов на основании предиката, не имеющего состояния, выполняется без сохранения состояния, поскольку каждый элемент обрабатывается отдельно. Следовательно, метод `filter()` может (и должен) выполняться без сохранения состояния. Отличие операций с сохранением состояния от операций без сохранения состояния особенно важно для параллельной обработки потоков данных, поскольку операцию с сохранением состояния, возможно, придется выполнить не за один, а за несколько проходов.

Интерфейс `Stream` оперирует ссылками на объекты, и поэтому он не может обращаться непосредственно к примитивным типам данных. Для обработки потоков примитивных типов данных в потоковом API определяются следующие интерфейсы:

- `DoubleStream`
- `IntStream`
- `LongStream`

Все эти интерфейсы расширяют интерфейс `BaseStream` и обладают теми же функциональными возможностями, что и интерфейс `Stream`, за исключением того, что они оперируют примитивными, а не ссылочными типами данных. Они предоставляют также служебные методы, например, `boxed()`, упрощающие их применение. В этой главе основное внимание уделяется интерфейсу `Stream`, поскольку потоки объектов употребляются чаще всего. Хотя аналогичным образом могут быть использованы и потоки примитивных типов данных.

Получение потока данных

Получить поток данных можно самыми разными способами. Вероятно, самый распространенный способ получения потока данных из коллекции. В версии JDK 8 интерфейс `Collection` дополнен двумя методами, специально предназначенными для получения потока данных из коллекции. Первый из них называется `stream()`, а его общая форма показана ниже.

```
default Stream<E> stream()
```

В реализации по умолчанию этот метод возвращает последовательный поток данных. Второй метод называется `parallelStream()`, а его общая форма выглядит следующим образом:


```
default Stream<E> parallelStream()
```

В реализации по умолчанию этот метод возвращает параллельный поток данных, если это вообще возможно. А если получить параллельный поток данных нельзя, то вместо него может быть возвращен последовательный поток данных. Параллельные потоки данных поддерживают параллельное выполнение потоковых операций. Благодаря тому что интерфейс `Collection` может быть реализован в классе каждой коллекции, с помощью упомянутых выше методов можно получить поток из класса любой коллекции, в том числе `ArrayList` или `HashSet`.

Поток данных можно также получить из массива, вызвав метод `stream()`, введенный в класс `Arrays`, начиная с версии `JDK 8`. Ниже приведена одна из общих форм этого метода.

```
static <T> Stream<T> stream(T[] массив)
```

Этот метод возвращает последовательный поток данных элементов заданного массива. Так, если имеется массив `addresses` типа `Address`, то в следующей строке кода получается поток для его элементов:

```
Stream<Address> addrStrm = Arrays.stream(addresses);
```

Определяется также несколько перегружаемых вариантов метода `stream()`, в том числе для обработки массивов примитивных типов данных. Они возвращают поток данных типа `IntStream`, `DoubleStream` или `LongStream`.

Потоки данных можно получить и целым рядом других способов. Например, многие потоковые операции возвращают новый поток данных, а для ввода данных из источника такой поток можно получить, вызвав метод `lines()` для буферизированного потока чтения типа `BufferedReader`. Но каким бы образом такой поток ни был получен, им можно воспользоваться таким же образом, как и любым другим потоком данных.

Простой пример потока данных

Прежде чем продолжить дальше, рассмотрим пример, в котором применяются потоки данных. В приведенной ниже программе сначала создается списочный массив `myList` типа `ArrayList`, содержащий коллекцию целочисленных значений, которые автоматически упаковываются в объекты ссылочного типа `Integer`. Затем в этой программе получается поток данных, использующий массив `myList` в качестве источника данных. А далее в ней демонстрируются различные потоковые операции.

```
// Продемонстрировать несколько потоковых операций

import java.util.*;
import java.util.stream.*;

class StreamDemo {

    public static void main(String[] args) {
        // создать списочный массив значений типа Integer
        ArrayList<Integer> myList = new ArrayList<>( );
        myList.add(7);
        myList.add(18);
```

```

myList.add(10);
myList.add(24);
myList.add(17);
myList.add(5);

System.out.println("Исходный список: " + myList);

// получить поток элементов списочного массива
Stream<Integer> myStream = myList.stream();

// получить минимальное и максимальное значения, вызвав
// методы min(), max(), isPresent() и get()
Optional<Integer> minVal = myStream.min(Integer::compare);
if(minVal.isPresent()) System.out.println(
    "Минимальное значение: " + minVal.get());

// непременно получить новый поток данных, поскольку
// предыдущий вызов метода min() стал оконечной операцией,
// употребившей поток данных
myStream = myList.stream();
Optional<Integer> maxVal = myStream.max(Integer::compare);
if(maxVal.isPresent()) System.out.println(
    "Максимальное значение: " + maxVal.get());

// отсортировать поток данных, вызвав метод sorted()
Stream<Integer> sortedStream = myList.stream().sorted();

// отобразить отсортированный поток данных, вызвав метод forEach()
System.out.print("Отсортированный поток данных: ");
sortedStream.forEach((n) -> System.out.print(n + " "));
System.out.println();

// вывести только нечетные целочисленные значения,
// вызвав метод filter()
Stream<Integer> oddVals =
    myList.stream().sorted().filter((n) -> (n % 2) == 1);
System.out.print("Нечетные значения: ");
oddVals.forEach((n) -> System.out.print(n + " "));
System.out.println();

// вывести только те нечетные целочисленные значения,
// которые больше 5. Обратите внимание на конвейеризацию
// обеих операций фильтрации
oddVals = myList.stream().filter((n) -> (n % 2) == 1)
    .filter((n) -> n > 5);
System.out.print("Нечетные значения больше 5: ");
oddVals.forEach((n) -> System.out.print(n + " "));
System.out.println();
}
}

```

Ниже приведен результат, выводимый данной программой.

```

Исходный список: [7, 18, 10, 24, 17, 5]
Минимальное значение: 5
Максимальное значение: 24
Отсортированный поток данных: 5 7 10 17 18 24
Нечетные значения: 5 7 17
Нечетные значения больше 5: 7 17

```

Рассмотрим каждую потоковую операцию из данной программы в отдельности. После создания списочного массива типа `ArrayList` в данной программе вызывается метод `stream()` для получения потока элементов этого массива:

```
Stream<Integer> myStream = myList.stream();
```

Как пояснялось ранее, в интерфейсе `Collection` теперь определяется метод `stream()` для получения потока данных из вызывающей коллекции. Благодаря тому что интерфейс `Collection` реализуется классом каждой коллекции, вызвав метод `stream()`, можно получить поток данных для коллекции любого типа, в том числе и `ArrayList`. В данном случае ссылка на получаемый поток данных присваивается переменной экземпляра `myStream`.

Далее в рассматриваемой здесь программе получается минимальное значение, обнаруживаемое в потоке данных (разумеется, оно является минимальным и в источнике данных). Полученное минимальное значение затем отображается, как показано ниже.

```
Optional<Integer> minVal = myStream.min(Integer::compare);
if(minVal.isPresent()) System.out.println(
    "Минимальное значение: " + minVal.get());
```

Как следует из табл. 29.2, метод `min()` объявляется следующим образом:

```
Optional<T> min(Comparator<? super T> компаратор)
```

Обратите прежде всего внимание на параметр *компаратор* метода `min()`. Обозначаемый им компаратор служит для сравнения двух элементов в потоке данных. В данном примере методу `min()` передается ссылка на метод `compare()` из класса `Integer`. Этот метод служит для реализации компаратора типа `Comparator`, способного сравнивать два объекта типа `Integer`. Обратите далее внимание на то, что метод `min()` возвращает объект типа `Optional`. Класс `Optional` подробно описывается в главе 19, а здесь вкратце поясняется принцип его действия. Этот класс является обобщенным, входит в состав пакета `java.util` и объявляется следующим образом:

```
class Optional<T>
```

где параметр **T** обозначает тип элемента. Экземпляр класса `Optional` может содержать значение типа `T` или же быть пустым. Вызвав метод `isPresent()`, можно определить, имеется ли значение в данном объекте. Если он содержит значение, то его можно получить, вызвав метод `get()`. В данном примере возвращается объект, содержащий минимальное значение из потока данных в виде объекта типа `Integer`.

И последнее замечание: метод `min()` выполняет окончательную операцию, употребляющую поток данных. Поэтому по его завершении потоком данных, на который ссылается переменная экземпляра `myStream`, нельзя воспользоваться снова.

В следующем фрагменте кода получается и выводится максимальное значение, обнаруживаемое в потоке данных:

```
myStream = myList.stream();
Optional<Integer> maxVal = myStream.max(Integer::compare);
if(maxVal.isPresent()) System.out.println(
    "Максимальное значение: " + maxVal.get());
```

И в этом случае переменной `myStream` присваивается поток данных, возвращаемый методом `myList.stream()`. Как пояснялось ранее, это требуется для того, чтобы получить новый поток данных. Ведь предыдущий поток данных употреблен при вызове метода `min()`. И только после этого вызывается метод `max()`, чтобы по-

лучить максимальное значение. Как и метод `min()`, этот метод возвращает объект типа `Optional`, значение которого получается в результате вызова метода `get()`.

Далее в рассматриваемой здесь программе получается отсортированный поток данных, как показано в приведенной ниже строке кода.

```
Stream<Integer> sortedStream = myList.stream().sorted();
```

В этой строке кода метод `sorted()` вызывается для потока данных, возвращаемого в результате вызова метода `myList.stream()`. А поскольку метод `sorted()` выполняет промежуточную операцию, то в конечном итоге получается новый поток данных, который присваивается переменной `sortedStream`. Содержимое отсортированного потока данных выводится с помощью метода `forEach()` следующим образом:

```
sortedStream.forEach((n) -> System.out.print(n + " "));
```

В этой строке кода метод `forEach()` выполняет операцию над каждым элементом потока данных. В частности, он вызывает метод `System.out.print()` для каждого элемента из отсортированного потока данных, хранящегося в переменной `sortedStream`. Эта операция выполняется лямбда-выражением. Ниже приведена общая форма метода `forEach()`.

```
void forEach(Consumer<? super T> действие)
```

Здесь `Consumer` — это обобщенный функциональный интерфейс, определяемый в пакете `java.util.function`. Он содержит абстрактный метод `accept()`, объявляемый следующим образом:

```
void accept(T ссылка_на_объект)
```

Лямбда-выражение, указываемое при вызове метода `forEach()`, предоставляет реализацию метода `accept()`. Метод `forEach()` выполняет окончательную операцию, и поэтому по его завершении поток данных оказывается употребленным.

Далее отсортированный поток данных фильтруется методом `filter()`, чтобы в нем остались только нечетные значения:

```
Stream<Integer> oddVals =  
    myList.stream().sorted().filter((n) -> (n % 2) == 1);
```

Фильтрация потока данных методом `filter()` выполняется по указанному предикату. Этот метод возвращает новый поток данных, содержащий только те элементы из исходного потока, которые удовлетворяют указанному предикату. Ниже приведена общая форма метода `filter()`.

```
Stream<T> filter(Predicate<? super T> предикат)
```

Здесь `Predicate` — это обобщенный функциональный интерфейс, определяемый в пакете `java.util.function`. Он содержит абстрактный метод `test()`, объявляемый следующим образом:

```
boolean test(T ссылка_на_объект)
```

Этот метод возвращает логическое значение `true`, если указанная `ссылка_на_объект` удовлетворяет предикату, а иначе — логическое значение `false`. А реализуется метод `test()` в лямбда-выражении, передаваемом методу `filter()`. И по-

сколько метод `filter()` выполняет промежуточную операцию, то он возвращает новый поток данных, содержащий отфильтрованные элементы (в данном случае нечетные целочисленные значения). Эти элементы отображаются далее методом `forEach()`, как и прежде.

Метод `filter()` или любая другая промежуточная операция возвращает новый поток данных. Поэтому отфильтрованный поток может быть отфильтрован еще раз, что и демонстрируется в следующем фрагменте кода, где производится новый поток данных, содержащий только те нечетные целочисленные значения, которые больше 5. Обратите внимание на то, что обоим фильтрам передаются лямбда-выражения.

```
oddVals = myList.stream().filter((n) -> (n % 2) == 1)
                    .filter((n) -> n > 5);
```

Операции сведения

Рассмотрим подробнее методы `min()` и `max()` из предыдущего примера. Оба метода выполняют окончательные операции и возвращают результат, исходя из элементов в потоке данных. По терминологии потокового API эти методы представляют собой операции сведения, поскольку каждый из них сводит поток данных к единственному значению (в данном случае — максимальному и минимальному соответственно). В потоковом API такие операции сведения называются *особыми*, поскольку они выполняют особую функцию. Помимо методов `min()` и `max()`, имеются и другие методы, выполняющие особые операции сведения. Например, метод `count()` подсчитывает количество элементов в потоке данных. Но в методе `reduce()` понятие сведения обобщается. Вызывая метод `reduce()`, можно вернуть значение из потока данных по любому произвольному критерию. По определению все операции сведения являются окончательными.

В интерфейсе `Stream` определяются три варианта метода `reduce()`. Ниже приведены общие формы двух из них.

```
Optional<T> reduce(BinaryOperator<T> накопитель)
T reduce(T значение_идентичности, BinaryOperator<T> накопитель)
```

В первой форме возвращается объект типа `Optional`, содержащий полученный результат, а во второй форме — объект типа `T`, т.е. типа элемента из потока данных. В обеих формах указанный *накопитель* обозначает функцию, оперирующую двумя значениями и получающую результат. Во второй форме *значение_идентичности* обозначает такое значение, что операция накопления, включающая *значение_идентичности* и любой элемент из потока данных, дает в итоге тот же самый элемент без изменения. Так, если выполняется операция сложения, то значение идентичности равно нулю, поскольку $0 + x = x$. А если выполняется операция умножения, то значение идентичности равно 1, поскольку $1 * x = x$.

Интерфейс `BinaryOperator` является функциональным и объявляется в пакете `java.util.function`. Он расширяет функциональный интерфейс `BiFunction`. В интерфейсе `BiFunction` определяется следующий абстрактный метод:

```
R apply(T значение1, U значение2)
```

где параметр **R** обозначает тип результата; параметр **T** — тип первого операнда; параметр **U** — тип второго операнда. Следовательно, метод `apply()` применяет функцию к своим операндам (*значение1* и *значение2*) и возвращает результат. Когда функциональный интерфейс `BinaryOperator` расширяет функциональный интерфейс `BiFunction`, то все параметры типа в нем обозначают один и тот же тип. Следовательно, в интерфейсе `BinaryOperator` метод `apply()` объявляется следующим образом:

T apply(T значение1, T значение2)

По отношению к методу `reduce()` параметр *значение1* метода `apply()` будет содержать предыдущий результат, тогда как параметр *значение2* — следующий элемент. При первом вызове данного метода параметр *значение2* будет содержать значение идентичности или первый элемент в зависимости от применяемого варианта метода `reduce()`.

Следует, однако, иметь в виду, что операция накопления должна удовлетворять трем ограничениям. Она должна быть:

- без сохранения состояния;
- без вмешательства;
- ассоциативная.

Как пояснялось ранее, операция *без сохранения состояния* означает, что она опирается на сведения о состоянии. Следовательно, каждый элемент из потока данных обрабатывается отдельно. Операция *без вмешательства* означает, что источник данных не видоизменяется самой операцией. И наконец, операция должна быть *ассоциативной*. В данном случае понятие *ассоциативная* операция употребляется в его обычном для арифметики значении. Это означает, что если ассоциативный оператор применяется в последовательности операций, то не имеет никакого значения, какая именно пара операндов обрабатывается первой. Например, вычисление выражения

```
(10 * 2) * 7
```

дает такой же результат, как и вычисление этого выражения:

```
10 * (2 * 7)
```

Ассоциативность имеет особое значение для правильного применения операций сведения в параллельных потоках данных, обсуждаемых в следующем разделе. В следующем примере программы демонстрируется применение рассмотренных выше вариантов метода `reduce()`:

```
// Продемонстрировать применение метода reduce()
```

```
import java.util.*;
import java.util.stream.*;
```

```
class StreamDemo2 {
```

```
    public static void main(String[] args) {
```

```
        // создать список объектов типа Integer
```

```

ArrayList<Integer> myList = new ArrayList<>( );

myList.add(7);
myList.add(18);
myList.add(10);
myList.add(24);
myList.add(17);
myList.add(5);

// Два способа получения результата перемножения целочисленных
// элементов списка myList с помощью метода reduce()
Optional<Integer> productObj = myList.stream().reduce((a,b) -> a*b);
if(productObj.isPresent())
    System.out.println("Произведение в виде объекта типа Optional: "
        + productObj.get());

int product = myList.stream().reduce(1, (a,b) -> a*b);
System.out.println(
    "Произведение в виде значения типа int: " + product);
}
}

```

Как показано ниже, в обоих вариантах применения метода `reduce()` получается одинаковый результат.

```

Произведение в виде объекта типа Optional: 2570400
Произведение в виде значения типа int: 2570400

```

Сначала в данной программе применяется первый вариант метода `reduce()` с лямбда-выражением для получения произведения двух числовых значений. В связи с тем что поток в данном примере содержит объекты типа `Integer`, они автоматически распаковываются перед операцией умножения и снова упаковываются перед возвратом результата. Оба перемножаемых значения представляют текущий результат и следующий элемент в потоке данных. А конечный результат возвращается в виде объекта типа `Optional`. Выводимое значение получается в результате вызова метода `get()` для возвращаемого объекта.

Далее в данной программе применяется второй вариант метода `reduce()`, при вызове которого значение идентичности указывается явным образом: для операции умножения оно равно 1. Обратите внимание на то, что результат возвращается в виде объекта, тип которого соответствует типу элемента из потока данных (в данном случае — `Integer`).

Столь простые операции сведения, как умножение, удобны в качестве примеров, но этим применение операций сведения, конечно, не ограничивается. Так, если обратиться к предыдущему примеру программы, то получить произведение только четных целочисленных значений можно следующим образом:

```

int evenProduct = myList.stream().reduce(1, (a,b) -> {
    if(b%2 == 0) return a*b; else return a;
});

```

Обратите особое внимание на лямбда-выражение. Если параметр `b` имеет четное числовое значение, то возвращается произведение `a * b`, а иначе — значение параметра `a`. И это вполне допустимо, поскольку параметр `a` содержит текущий результат, а параметр `b` — следующий элемент из потока данных, как пояснялось ранее.

Параллельные потоки данных

Прежде чем продолжить рассмотрение потокового API, следует остановиться на параллельных потоках данных. Как отмечалось ранее в данной книге, параллельное выполнение кода на многоядерных процессорах позволяет добиться значительного повышения производительности. Вследствие этого параллельное программирование стало важной частью современного арсенала средств программистов. Но в то же время параллельное программирование — дело непростое и чреватое ошибками. Поэтому одно из преимуществ библиотеки потоков данных заключается в том, что она позволяет просто и надежно организовать параллельное выполнение некоторых операций.

Запросить параллельную обработку потока данных совсем не трудно. Для этого достаточно воспользоваться параллельным потоком данных. Как упоминалось ранее, чтобы получить параллельный поток данных, можно, в частности, вызвать метод `parallelStream()`, определенный в классе `Collection`. С другой стороны, можно вызвать метод `parallel()` в последовательном потоке данных. Метод `parallel()` определяется в интерфейсе `BaseStream` следующим образом:

```
S parallel()
```

Этот метод возвращает параллельный поток данных, исходя из вызывающего последовательного потока данных. (Если же он вызывается в потоке данных, который уже является параллельным, то возвращается вызывающий поток.) Следует, однако, иметь в виду, что даже при наличии параллельного потока данных искомый параллелизм достигается только в том случае, если его поддерживает исполняющая среда.

Как только будет получен параллельный поток данных, последующие операции в этом потоке могут выполняться параллельно, при условии, что параллелизм поддерживается исполняющей средой. Например, первая операция сведения, выполняемая методом `reduce()` в предыдущей программе, может быть распараллелена, если вызов метода `stream()` заменить на вызов метода `parallelStream()`, как показано ниже. Результат окажется тем же самым, но операция умножения может быть распараллелена в разных потоках исполнения.

```
Optional<Integer> productObj = myList.parallelStream().reduce((a,b) -> a*b);
```

Как правило, любая операция в параллельном потоке данных должна выполняться без сохранения состояния. Она должна быть также ассоциативной и без вмешательства. Этим гарантируется, что результаты выполнения операций в параллельном потоке данных остаются такими же, как и результаты выполнения аналогичных операций в последовательном потоке данных.

Для применения параллельных потоков данных особенно полезной может оказаться приведенный ниже вариант метода `reduce()`. Этот вариант позволяет указать порядок объединения частичных результатов.

```
<U> U reduce(U значение_идентичности, BiFunction<U, ? super T, U>  
            накопитель BinaryOperator<U> объединитель)
```


В данном варианте метода `reduce()` функции накопителя и объединителя одинаковы. И это приводит к ошибке, ведь когда объединяются два частичных результата, перемножаются их квадратные корни, а не сами частичные результаты.

Любопытно, что если изменить параллельный поток данных на последовательный в предыдущем вызове метода `reduce()`, то выполнение данной операции даст правильный результат, поскольку в этом случае объединять оба частичных результата не нужно. Ошибка возникает лишь в том случае, если применяется параллельный поток данных.

Вызвав метод `sequential()`, определяемый в интерфейсе `BaseStream`, можно заменить параллельный поток данных последовательным. Ниже приведена общая форма этого метода. Как правило, параллельный поток данных заменяется последовательным, и наоборот, по мере надобности.

S `sequential()`

Организуя параллельное выполнение в потоке данных, следует также принимать во внимание порядок следования элементов. Потоки данных могут быть как упорядоченными, так и неупорядоченными. Если источник данных упорядочен, то, как правило, упорядочен и поток данных. Но иногда можно добиться повышения производительности, если применяемый поток данных неупорядочен. В этом случае каждую часть потока данных можно обрабатывать отдельно, не согласуя ее с остальными частями. В тех случаях, когда порядок следования операций не имеет особого значения, можно выбрать режим работы без упорядочения, вызвав метод `unordered()`:

S `unordered()`

И наконец, при выполнении метода `forEach()` упорядоченность параллельного потока может не сохраняться. Если же при выполнении операции над каждым элементом в параллельном потоке данных требуется сохранить его упорядоченность, то лучше воспользоваться методом `forEachOrdered()`. Этот метод применяется таким же образом, как и метод `forEach()`.

Отображение

Нередко элементы одного потока данных требуется отобразить на элементы другого потока. Например, из базы данных, содержащей имя, номер телефона и адрес электронной почты в одном потоке данных, требуется отобразить только имя и адрес электронной почты в другом потоке. А в качестве другого примера можно привести преобразование, которое требуется выполнить только над некоторыми элементами в потоке данных. Операции отображения весьма распространены, и поэтому в потоковом API предоставляется их встроенная поддержка. Самое общее отображение выполняется методом `map()`. Ниже приведена его общая форма.

<R> Stream<R> map(Function<? super T, ? extends R> функция_отображения)

Здесь параметр `R` обозначает тип элементов из нового потока данных; параметр `T` — тип элементов из вызывающего потока данных; параметр *функция_отображения* — экземпляр функционального интерфейса `Function`, выполняю-

щий отображение. Функция отображений должна выполнять операцию без сохранения состояния и без вмешательства. А метод `map()` выполняет промежуточную операцию, поскольку он возвращает новый поток данных.

Интерфейс `Function` является функциональным и объявляется в пакете `java.util.function` следующим образом:

```
Function<T, R>
```

где параметр **T** по отношению к методу `map()` обозначает тип элемента, а параметр **R** — результат отображения. В функциональном интерфейсе `Function` объявляется следующий абстрактный метод:

```
R apply(T значение)
```

где параметр *значение* обозначает ссылку на отображаемый объект. В итоге возвращается результат отображения.

Ниже приведен простой пример применения метода `map()`. Это переделанная версия программы из предыдущего примера. Как и прежде, в программе вычисляется произведение квадратных корней элементов из списочного массива типа `ArrayList`. Но в данной версии программы квадратные корни элементов сначала отображаются на новый поток данных, а затем для вычисления их произведения вызывается метод `reduce()`.

```
// Отобразить один поток данных на другой
```

```
import java.util.*;
import java.util.stream.*;
```

```
class StreamDemo4 {

    public static void main(String[] args) {

        // Список числовых значений типа double
        ArrayList<Double> myList = new ArrayList<>( );

        myList.add(7.0);
        myList.add(18.0);
        myList.add(10.0);
        myList.add(24.0);
        myList.add(17.0);
        myList.add(5.0);

        // Отобразить квадратные корни элементов из списка myList
        // на новый поток данных
        Stream<Double> sqrtRootStrm =
            myList.stream().map((a) -> Math.sqrt(a));

        // получить произведение квадратных корней
        double productOfSqrRoots = sqrtRootStrm.reduce(1.0, (a,b) -> a*b);

        System.out.println(
            "Произведение квадратных корней равно " + productOfSqrRoots);
    }
}
```

Эта версия программы выводит такой же результат, как и предыдущая ее версия. А отличается она от предыдущей версии лишь тем, что преобразование

(т.е. вычисление произведения квадратных корней) выполняется во время отображения, а не во время сведения. Благодаря этому для вычисления произведения можно воспользоваться формой метода `reduce()` с двумя параметрами, поскольку для этого больше не требуется предоставлять отдельную функцию объединителя.

Ниже приведен пример программы, где для создания нового потока данных, содержащего только избранные поля из исходного потока, используется метод `map()`. В данном примере исходный поток содержит объекты типа `NamePhoneEmail`, состоящие из имен, номеров телефонов и адресов электронной почты. На новый поток данных объектов типа `NamePhone` отображаются только имена и номера телефонов, тогда как адреса электронной почты отбрасываются.

```
// Применить метод map() для создания нового потока данных,
// содержащего только избранные элементы из исходного потока

import java.util.*;
import java.util.stream.*;

class NamePhoneEmail {
    String name;
    String phonenum;
    String email;

    NamePhoneEmail(String n, String p, String e) {
        name = n;
        phonenum = p;
        email = e;
    }
}

class NamePhone {
    String name;
    String phonenum;

    NamePhone(String n, String p) {
        name = n;
        phonenum = p;
    }
}

class StreamDemo5 {

    public static void main(String[] args) {

        // Список имен, номеров телефонов и адресов электронной почты
        ArrayList<NamePhoneEmail> myList = new ArrayList<>();
        myList.add(new NamePhoneEmail("Ларри", "555-5555",
                                         "Larry@HerbSchildt.com"));
        myList.add(new NamePhoneEmail("Джеймс", "555-4444",
                                         "James@HerbSchildt.com"));
        myList.add(new NamePhoneEmail("Мэри", "555-3333",
                                         "Mary@HerbSchildt.com"));

        System.out.println("Исходные элементы из списка myList: ");
        myList.stream().forEach( (a) -> {
            System.out.println(a.name + " " + a.phonenum + " " + a.email);
        });
        System.out.println();
    }
}
```

```
// отобразить на новый поток данных
// только имена и номера телефонов
Stream<NamePhone> nameAndPhone = myList.stream().map(
    (a) -> new NamePhone(a.name, a.phonenum)
);

System.out.println("Список имен и номеров телефонов: ");
nameAndPhone.forEach( (a) -> {
    System.out.println(a.name + " " + a.phonenum);
});
}
```

Ниже приведен результат, выводимый данной программой. Он позволяет сверить исходные элементы списка с отображаемыми.

Исходные элементы из списка myList:
 Ларри 555-5555 Larry@HerbSchildt.com
 Джеймс 555-4444 James@HerbSchildt.com
 Мэри 555-3333 Mary@HerbSchildt.com

Список имен и номеров телефонов:
 Ларри 555-5555
 Джеймс 555-4444
 Мэри 555-3333

Несколько промежуточных операций можно объединить в единый конвейер, что позволяет составлять довольно эффективные действия. Например, в следующей строке кода для получения нового потока данных, содержащего только элементы имени и номера телефона, совпадающие с именем "Джеймс", сначала вызывается метод `filter()`, а затем метод `map()`:

```
Stream<NamePhone> nameAndPhone = myList.stream().
    filter((a) -> a.name.equals("Джеймс")).
    map((a) -> new NamePhone(a.name, a.phonenum));
```

Такая операция фильтрации весьма характерна для составления запросов базы данных. Приобретая некоторый опыт обращения с потоковым API, вы непременно обнаружите, что из подобных цепочек операций можно составлять довольно сложные запросы, объединения и выборки данных в потоке данных.

Помимо описанной выше формы, имеются еще три формы метода `map()`. Все они возвращают поток данных примитивного типа и приведены ниже.

```
IntStream mapToInt(ToIntFunction<? super T> функция_отображения)
LongStream mapToLong(ToLongFunction<? super T> функция_отображения)
DoubleStream mapToDouble(ToDoubleFunction<? super T> функция_отображения)
```

Каждая задаваемая *функция_отображения* должна реализовывать абстрактный метод, определяемый в заданном интерфейсе и возвращающий значение указанного типа. Например, в интерфейсе `ToDoubleFunction` определяется метод `applyAsDouble(T значение)`, который должен возвращать значение своего параметра как `double`.

В приведенном ниже примере программы применяется поток данных примитивного типа. Сначала в этой программе создается списочный массив типа `ArrayList`, состоящий из объектов типа `Double`. Затем для создания потока данных типа `IntStream`, содержащего максимально допустимый предел каждого числового значения, вызывается метод `stream()`, а далее — метод `mapToInt()`.

```
// Отобразить поток данных типа Stream на поток данных типа IntStream

import java.util.*;
import java.util.stream.*;

class StreamDemo6 {

    public static void main(String[] args) {

        // Список значений типа double
        ArrayList<Double> myList = new ArrayList<>( );

        myList.add(1.1);
        myList.add(3.6);
        myList.add(9.2);
        myList.add(4.7);
        myList.add(12.1);
        myList.add(5.0);

        System.out.print("Исходные значения из списка myList: ");
        myList.stream().forEach( (a) -> {
            System.out.print(a + " ");
        });
        System.out.println();

        // Отобразить максимально допустимый предел каждого
        // значения из списка myList на поток данных типа IntStream
        IntStream cStrm = myList.stream().mapToInt((a) -> (int) Math.ceil(a));

        System.out.print("Максимально допустимые пределы значений" +
            "из списка myList: ");
        cStrm.forEach( (a) -> {
            System.out.print(a + " ");
        });
    }
}
```

Ниже приведен результат, выводимый данной программой. Поток данных, получаемый методом `mapToInt()`, содержит максимально допустимые пределы исходных значений элементов из списка `myList`.

```
Исходные значения из списка myList: 1.1 3.6 9.2 4.7 12.1 5.0
Максимально допустимые пределы значений из списка myList: 2 4 10 5 13 5
```

Завершая обсуждение темы отображения, следует заметить, что в потоковом API предоставляются также методы, поддерживающие *плоское отображение*. К их числу относятся методы `flatMap()`, `flatMapToInt()`, `flatMapToLong()` и `flatMapToDouble()`. Методы плоского отображения предназначены для выхода из тех ситуаций, когда каждый элемент в исходном потоке данных отображается на более чем один элемент в итоговом потоке данных.

Накопление

В приведенных выше примерах можно получить поток данных из коллекции, что, как правило, и делается. Но иногда требуется совсем противоположное: полу-

чить коллекцию из потока данных. Для выполнения подобного действия в потоковом API предоставляется метод `collect()`. У этого метода имеются две общие формы. Ниже приведена одна из них, употребляемая в этом разделе.

```
<R, A> R collect(Collector<? super T, A, R> функция_накопления)
```

Здесь параметр **R** обозначает тип получаемого результата; параметр **T** — тип элемента из вызывающего потока данных; параметр **A** — тип накапливаемых внутри данных; параметр *функция_накопления* — порядок обработки коллекции. Метод `collect()` выполняет окончательную операцию.

Интерфейс `Collector` объявляется в пакете `java.util.stream`, как показано ниже.

```
interface Collector<T, A, R>
```

Параметры **T**, **A** и **R** имеют описанное выше назначение. В интерфейсе `Collector` определяется ряд методов, но в этой главе не демонстрируется их реализация. Вместо этого используются два предопределенных накопителя, предоставляемых в классе `Collectors`, входящем в состав пакета `java.util.stream`.

В классе `Collectors` определяется ряд статических методов накопления, которыми можно воспользоваться в готовом виде. Ниже приведены общие формы двух употребляемых далее методов `toList()` и `toSet()`.

```
static <T> Collector<T, ?, List<T>> toList()
static <T> Collector<T, ?, Set<T>> toSet()
```

Метод `toList()` возвращает накопитель, предназначенный для накопления элементов в списке типа `List`, а метод `toSet()` — накопитель, предназначенный для накопления элементов во множестве типа `Set`. Например, для накопления элементов в списке типа `List` можно вызвать метод `collect()` следующим образом:

```
collect(Collectors.toList())
```

В приведенном ниже примере программы все сказанное выше о накоплении демонстрируется на практике. Это версия программы из предыдущего примера, переделанная таким образом, чтобы накапливать имена и номера телефонов в списке типа `List` и множестве типа `Set`.

```
// Использовать метод collect() для создания списка типа List
// и множества типа Set из потока данных
```

```
import java.util.*;
import java.util.stream.*;

class NamePhoneEmail {
    String name;
    String phonenum;
    String email;

    NamePhoneEmail(String n, String p, String e) {
        name = n;
        phonenum = p;
        email = e;
    }
}

class NamePhone {
```

```
String name;
String phonenum;

NamePhone(String n, String p) {
    name = n;
    phonenum = p;
}

class StreamDemo7 {

    public static void main(String[] args) {

        // Список имен, номеров телефонов и адресов электронной почты
        ArrayList<NamePhoneEmail> myList = new ArrayList<>();

        myList.add(new NamePhoneEmail("Ларри", "555-5555",
                                         "Larry@HerbSchildt.com"));
        myList.add(new NamePhoneEmail("Джеймс", "555-4444",
                                         "James@HerbSchildt.com"));
        myList.add(new NamePhoneEmail("Мэри", "555-3333",
                                         "Mary@HerbSchildt.com"));

        // отобразить только имена и номера телефонов на новый поток данных
        Stream<NamePhone> nameAndPhone = myList.stream().map(
            (a) -> new NamePhone(a.name, a.phonenum)
        );

        // вызвать метод collect(), чтобы составить список типа List
        // из имен и номеров телефонов
        List<NamePhone> npList = nameAndPhone.collect(Collectors.toList());

        System.out.println("Имена и номера телефонов в списке типа List:");
        for(NamePhone e : npList)
            System.out.println(e.name + ": " + e.phonenum);

        // получить другое отображение имен и номеров телефонов
        nameAndPhone = myList.stream().map(
            (a) -> new NamePhone(a.name, a.phonenum)
        );

        // а теперь создать множество типа Set, вызвав метод collect()
        Set<NamePhone> npSet = nameAndPhone.collect(Collectors.toSet());

        System.out.println(
            "\nИмена и номера телефонов в множестве типа Set:");
        for(NamePhone e : npSet)
            System.out.println(e.name + ": " + e.phonenum);
    }
}
```

Эта программа выводит следующий результат:

Имена и номера телефонов в списке типа List:
Ларри: 555-5555
Джеймс: 555-4444
Мэри: 555-3333

Имена и номера телефонов в множестве типа Set:
Джеймс: 555-4444
Ларри: 555-5555
Мэри: 555-3333

В следующей строке кода из данной программы накапливаются имена и номера телефонов в списке типа `List` с помощью метода `toList()`:

```
List<NamePhone> npList = nameAndPhone.collect(Collectors.toList());
```

После выполнения этой строки кода коллекция `npList` может быть использована аналогично любой другой коллекции типа `List`. Например, ее можно перебрать в цикле `for` в стиле `for each`, как показано в следующей строке кода:

```
for (NamePhone e : npList)
    System.out.println(e.name + ": " + e.phonenumber);
```

Аналогичным образом осуществляется и создание множества типа `Set` в результате вызова метода `collect(Collectors.toSet())`. Возможность перемещать данные из коллекции в поток, а затем обратно в коллекцию является одной из самых сильных сторон потокового API. Это позволяет оперировать коллекцией через поток данных, а затем реорганизовывать его в виде коллекции. Более того, потоковые операции могут при соответствующих условиях выполняться параллельно.

Форма метода `collect()`, использованная в предыдущем примере, достаточно удобна и нередко удовлетворяет требованиям программистов. Но имеется и другая, приведенная ниже форма этого метода, предоставляющая больше возможностей управлять процессом накопления.

```
<R> R collect(Supplier<R> адресат, BiConsumer<R, ? super T>
              накопитель, BiConsumer<R, R> объединитель)
```

Здесь параметр *адресат* обозначает порядок создания объекта, содержащего результат. Например, чтобы воспользоваться связным списком типа `LinkedList` в качестве результирующей коллекции, следует указать ее конструктор. Функция *накопителя* добавляет элемент к результату, а функция *объединителя* служит для объединения двух частичных результатов. Следовательно, эти функции действуют таким же образом, как и в методе `reduce()`. Но обе функции должны выполняться без сохранения состояния и без вмешательства, а также должны быть ассоциативными.

Следует заметить, что параметр *адресат* относится к типу `Supplier`. Это функциональный интерфейс, объявляемый в пакете `java.util.function`. В этом функциональном интерфейсе определяется единственный метод `get()` без параметров, и в данном случае он возвращает объект типа `R`. Следовательно, по отношению к методу `collect()` метод `get()` возвращает ссылку на изменяемый объект хранения, например коллекцию.

Следует также иметь в виду, что параметры *накопитель* и *объединитель* относятся к типу `BiConsumer`. Это функциональный интерфейс, определяемый в пакете `java.util.function`. В этом функциональном интерфейсе определяется абстрактный метод `accept()`:

```
void accept(T объект1, U объект2)
```

Этот метод выполняет определенного рода операцию над указанными объектом1 и объектом2. По отношению к *накопителю* указанный *объект1* обозначает целевую коллекцию, а *объект2* — две объединяемые коллекции.

Используя описанную только что форму метода `collect()`, можно указать связный список типа `LinkedList` в качестве целевой коллекции в предыдущем примере программы следующим образом:

```
LinkedList<NamePhone> npList = nameAndPhone.collect(
    () -> new LinkedList<>(),
    (list, element) -> list.add(element),
    (listA, listB) -> listA.addAll(listB));
```

Обратите внимание на то, что первым аргументом в методе `collect()` является лямбда-выражение, возвращающее новый связный список типа `LinkedList`. В качестве второго аргумента указывается выражение со стандартным методом коллекции `add()`, вводящим элемент в список, а в качестве третьего аргумента — выражение с методом `addAll()`, объединяющим два связных списка. Любопытно, что для ввода элемента в список можно воспользоваться любым методом, определенным в классе `LinkedList`. Например, для ввода элементов в начале списка можно вызвать метод `addFirst()` следующим образом:

```
(list, element) -> list.addFirst(element)
```

Нетрудно догадаться, что в качестве аргументов метода `collect()` далеко не всегда нужно указывать лямбда-выражение. Зачастую достаточно указать ссылку на метод или конструктор. Если снова обратиться к предыдущему примеру программы, то в ее исходный код можно ввести приведенную ниже строку, в которой создается множество типа `HashSet`, содержащее все элементы из потока `nameAndPhone`. Обратите внимание на то, что в качестве первого аргумента указывается ссылка на конструктор класса `HashSet`, а в качестве второго и третьего аргументов — ссылки на методы `add()` и `addAll()` из класса `HashSet` соответственно.

```
HashSet<NamePhone> npSet = nameAndPhone.collect(HashSet::new,
                                                HashSet::add,
                                                HashSet::addAll);
```

И последнее замечание: в терминологии потокового API метод `collect()` выполняет операцию, называемую *изменяемым сведением*. Дело в том, что в результате сведения получается изменяемый объект хранения, например коллекция.

Итераторы и потоки данных

Несмотря на то что поток данных не является объектом их хранения, для перебора его элементов можно воспользоваться итератором почти так же, как и для перебора элементов коллекции. В потоковом API поддерживаются два типа итераторов. Первым из них является традиционный итератор типа `Iterator`, а вторым — итератор-разделитель типа `Splititerator`, внедренный в версии JDK 8. Последний обеспечивает немало преимуществ в тех случаях, когда он применяется в параллельных потоках данных.

Применение итератора в потоке данных

Как упоминалось выше, итератор применяется в потоке данных почти так же, как и в коллекции. Подробнее итераторы рассматриваются в главе 18, но здесь все же стоит сделать их краткий обзор. Итераторы являются объектами классов, реализующих интерфейс `Iterator`, объявляемый в пакете `java.util`. В этом ин-

терфейсе имеются два метода — `hasNext()` и `next()`. Если для перебора имеется следующий элемент, то метод `hasNext()` возвращает логическое значение `true`, а иначе — логическое значение `false`, тогда как метод `next()` возвращает следующий элемент в итерации.

На заметку! В версии JDK 8 внедрены следующие дополнительные типы итераторов для обработки потоков примитивных типов данных: `PrimitiveIterator`, `PrimitiveIterator.OfDouble`, `PrimitiveIterator.OfLong` и `PrimitiveIterator.OfInt`. Все эти итераторы расширяют интерфейс `Iterator` и в целом действуют таким же образом, как и итераторы, непосредственно опирающиеся на интерфейс `Iterator`.

Чтобы получить итератор для потока данных, следует вызвать метод `iterator()` для этого потока. Ниже приведена общая форма этого метода, употребляемая в потоке данных типа `Stream`.

```
Iterator<T> iterator()
```

Здесь параметр **T** обозначает тип элемента. (Потоки примитивных типов данных возвращают данные соответствующего примитивного типа.)

В приведенном ниже примере программы демонстрируется перебор элементов потока с помощью итератора. В данном примере перебираются символьные строки в списочном массиве `ArrayList`, но этот процесс аналогичен для потока данных любого типа.

```
// Применить итератор в потоке данных

import java.util.*;
import java.util.stream.*;

class StreamDemo8 {

    public static void main(String[] args) {

        // создать список символьных строк
        ArrayList<String> myList = new ArrayList<>( );
        myList.add("Альфа");
        myList.add("Бета");
        myList.add("Гамма");
        myList.add("Дельта");
        myList.add("Кси");
        myList.add("Омега");

        // получить поток данных для списочного массива
        Stream<String> myStream = myList.stream();

        // получить итератор для потока данных
        Iterator<String> itr = myStream.iterator();

        // перебрать элементы в потоке данных
        while(itr.hasNext())
            System.out.println(itr.next());
    }
}
```

Ниже приведен результат, выводимый данной программой.

Альфа
Бета
Гамма
Дельта
Кси
Омега

Применение итератора-разделителя

Итератор-разделитель типа `SplitIterator` служит альтернативой обычному итератору типа `Iterator`, особенно для параллельной обработки. В общем, итератор-разделитель сложнее, чем обычный итератор. Более подробно он рассматривается в главе 18, но здесь все же стоит напомнить вкратце его особенности. В интерфейсе `SplitIterator` определяется несколько методов, но на практике применяются только три из них. Первым из них является метод `tryAdvance()`, выполняющий действие над следующим элементом и продвигающим итератор дальше. Ниже приведена общая форма этого метода.

`boolean tryAdvance(Consumer<? super T> действие)`

Здесь параметр *действие* обозначает конкретное действие, выполняемое над следующим элементом в итерации. Метод `tryAdvance()` возвращает логическое значение `true`, если имеется следующий элемент, или логическое значение `false`, если элементов больше не осталось. Как пояснялось ранее в этой главе, в функциональном интерфейсе `Consumer` объявляется единственный метод `accept()`, принимающий элемент типа `T` в качестве своего аргумента и возвращающий значение типа `void`, а по существу, ничего не возвращающий.

Благодаря тому что метод `tryAdvance()` возвращает логическое значение `false`, если больше не остается элементов для обработки, не составляет особого труда организовать цикл итерации, как показано в следующем примере:

```
while(splitItr.tryAdvance( // выполнить здесь действие ));
```

Заданное действие выполняется над следующим элементом до тех пор, пока метод `tryAdvance()` возвращает логическое значение `true`. Итерация завершается, когда метод `tryAdvance()` возвращает логическое значение `false`. Обратите внимание, как в одном методе `tryAdvance()` сочетаются функции, выполняемые методами `hasNext()` и `next()`, предоставляемыми интерфейсом `Iterator`. Благодаря этому повышается эффективность процесса итерации.

Ниже приведена переделанная версия программы из предыдущего примера. В этой версии вместо обычного итератора типа `Iterator` применяется итератор-разделитель типа `SplitIterator`. Эта версия программы выводит такой же результат, как и предыдущая ее версия.

```
// Применить итератор-разделитель в потоке данных

import java.util.*;
import java.util.stream.*;

class StreamDemo9 {

    public static void main(String[] args) {
```

```
// создать список символьных строк
ArrayList<String> myList = new ArrayList<>( );
myList.add("Альфа");
myList.add("Бета");
myList.add("Гамма");
myList.add("Дельта");
myList.add("Кси");
myList.add("Омега");

// получить поток данных для списочного массива
Stream<String> myStream = myList.stream();

// получить итератор-разделитель
Spliterator<String> splitItr = myStream.splitItr();

// перебрать элементы в потоке данных
while(splitItr.tryAdvance((n) -> System.out.println(n)));
}
}
```

Иногда некоторое действие требуется выполнить над всеми элементами сразу, а не над каждым из них по очереди. На этот случай в интерфейсе `Spliterator` предоставляется метод `forEachRemaining()`. Ниже приведена его общая форма.

default void forEachRemaining(Consumer<? super T> действие)

Этот метод выполняет заданное *действие* над каждым необработанным элементом и затем производит возврат. Так, если обратиться к предыдущему примеру программы, то оставшиеся в потоке символьные строки можно вывести следующим образом:

```
splitItr.forEachRemaining((n) -> System.out.println(n));
```

Обратите внимание на то, что этот метод вообще исключает потребность в циклическом переборе каждого элемента по очереди. И это еще одно преимущество итератора-разделителя типа `Spliterator`.

В интерфейсе `Spliterator` имеется еще один метод, называемый `trySplit()` и представляющий особый интерес. Этот метод разделяет итерируемые элементы на две части, возвращая новый итератор-разделитель типа `Spliterator` для одной из этих частей, тогда как другая часть остается доступной по исходному итератору-разделителю. Ниже приведена общая форма метода `trySplit()`.

Spliterator<T> trySplit()

Если разделить вызывающий итератор-разделитель типа `Spliterator` не удастся, то возвращается пустое значение `null`, а иначе — ссылка на возвращаемую часть. В качестве примера ниже приведена другая версия программы из предыдущего примера, где демонстрируется применение метода `trySplit()`.

```
// Продемонстрировать применение метода trySplit()

import java.util.*;
import java.util.stream.*;

class StreamDemo10 {

    public static void main(String[] args) {
```

```
// создать список символьных строк
ArrayList<String> myList = new ArrayList<>( );
myList.add("Альфа");
myList.add("Бета");
myList.add("Гамма");
myList.add("Дельта");
myList.add("Кси");
myList.add("Омега");

// получить поток данных для списочного массива
Stream<String> myStream = myList.stream();

// получить итератор-разделитель
Spliterator<String> splitItr = myStream.splitterator();

// а теперь разделить первый итератор
Spliterator<String> splitItr2 = splitItr.trySplit();

// использовать сначала итератор splitItr2, если
// нельзя разделить итератор splitItr
if(splitItr2 != null) {
    System.out.println("Результат, выводимый итератором splitItr2: ");
    splitItr2.forEachRemaining((n) -> System.out.println(n));
}

// а теперь воспользоваться итератором splitItr
System.out.println("\nРезультат, выводимый итератором splitItr: ");
splitItr.forEachRemaining((n) -> System.out.println(n));
}
}
```

Ниже приведен результат, выводимый данной программой.

Результат, выводимый итератором splitItr2:

Альфа
Бета
Гамма

Результат, выводимый итератором splitItr:

Дельта
Кси
Омега

В данном простом примере разделение итератора-разделителя типа `Spliterator` не имеет особого смысла, но оно может иметь *большое значение* при параллельной обработке крупных массивов данных. На практике все-таки лучше воспользоваться каким-нибудь другим методом из интерфейса `Stream` вместе с параллельным потоком данных, чем разделять вручную итератор-разделитель типа `Spliterator`. Последнее следует делать лишь в тех случаях, когда не подходит ни один из predefined методов.

Дальнейшее изучение потокового API

В этой главе были рассмотрены лишь некоторых из основных средств потокового API и продемонстрированы приемы из применения, хотя этих средств намного больше в данном прикладном программном интерфейсе. Прежде всего следует упомянуть ряд других методов, предоставляемых в интерфейсе `Stream`. Эти методы могут оказаться полезными в следующих случаях.

- Чтобы выяснить, удовлетворяет ли один или больше элементов указанному предикату, следует воспользоваться методом `allMatch()`, `anyMatch()` или `noneMatch()`.
- Чтобы получить количество элементов в потоке, следует вызвать метод `count()`.
- Чтобы получить поток, содержащий только однозначные элементы, нужно вызвать метод `distinct()`.
- Чтобы создать поток, содержащий указанный ряд элементов, следует воспользоваться методом `of()`.

И последнее замечание: потоковый API является эффективным дополнением Java. Этот прикладной программный интерфейс, скорее всего, будет со временем расширен дополнительными функциональными возможностями. Поэтому рекомендуется периодически заглядывать в документацию на потоковый API.

Регулярные выражения и другие пакеты

Первоначально язык Java включал восемь пакетов, называемых *базовым API*. С каждым последующим выпуском этот API пополнялся новыми пакетами. Ныне API Java содержит большое количество пакетов. Многие из них являются специализированными и не рассматриваются в данной книге. Тем не менее в этой главе будут рассмотрены пакеты `java.util.regex`, `java.lang.reflect`, `java.rmi` и `java.text`, поддерживающие обработку регулярных выражений, рефлекссию, удаленный вызов методов и форматирование текста соответственно. А в конце главы будет представлен новый прикладной программный интерфейс API даты и времени, входящий в пакет `java.time`, и подчиненные ему подпакеты.

Пакет *регулярных выражений* позволяет выполнять сложные операции сопоставления с шаблонами. Он подробно рассматривается в этой главе на многочисленных примерах его применения. *Рефлексией* называется способность программного обеспечения к самоанализу. Рефлексия является неотъемлемой частью технологии Java Beans, которая рассматривается в главе 37. *Удаленный вызов методов* (Remote Method Invocation — RMI) позволяет создавать на Java прикладные программы, распределяемые на нескольких машинах. В этой главе представлен простой пример построения архитектуры “клиент–сервер” с использованием удаленного вызова методов. Возможности *форматирования текста*, доступные в пакете `java.text`, находят немало примеров применения. И наконец, новый API даты и времени предлагает современный подход к обработке даты и времени.

Пакеты из базового API

В табл. 30.1 перечислены все пакеты из базового API, доступные на момент написания данной книги в пространстве имен `java`, с указанием их назначения.

Таблица 30.1. Пакеты из базового API

Пакет	Основное назначение
<code>java.applet</code>	Поддерживает построение апплетов
<code>java.awt</code>	Предоставляет средства для построения ГПИ
<code>java.awt.color</code>	Поддерживает цветовые пространства и профили
<code>java.awt.datatransfer</code>	Осуществляет обмен данными через системный буфер обмена

Пакет	Основное назначение
<code>java.awt.dnd</code>	Поддерживает операции перетаскивания
<code>java.awt.event</code>	Организует обработку событий
<code>java.awt.font</code>	Представляет различные типы шрифтов
<code>java.awt.geom</code>	Позволяет работать с геометрическими формами
<code>java.awt.im</code>	Позволяет вводить японские, китайские и корейские символы в компоненты редактирования текста
<code>java.awt.im.spi</code>	Поддерживает альтернативные устройства ввода
<code>java.awt.image</code>	Организует обработку изображений
<code>java.awt.image.renderable</code>	Поддерживает независимое воспроизведение изображений
<code>java.awt.print</code>	Поддерживает общие возможности печати
<code>java.beans</code>	Позволяет создавать компоненты программного обеспечения
<code>java.beans.beancontext</code>	Предоставляет исполняющую среду для компонентов Java Beans
<code>java.io</code>	Организует ввод-вывод данных
<code>java.lang</code>	Предоставляет базовые функциональные возможности
<code>java.lang.annotation</code>	Поддерживает аннотации (метаданные)
<code>java.lang.instrument</code>	Поддерживает инструментальные средства для разработки программ
<code>java.lang.invoke</code>	Поддерживает динамические языки
<code>java.lang.management</code>	Поддерживает управление исполняющей средой
<code>java.lang.ref</code>	Позволяет взаимодействовать с системой сборки "мусора"
<code>java.lang.reflect</code>	Анализирует код во время выполнения
<code>java.math</code>	Организует обработку больших целых и десятичных чисел
<code>java.net</code>	Поддерживает работу в сети
<code>java.nio</code>	Пакет верхнего уровня для классов системы ввода-вывода NIO. Инкапсулирует буфера
<code>java.nio.channels</code>	Инкапсулирует каналы, применяемые в системе ввода-вывода NIO
<code>java.nio.channels.spi</code>	Предоставляет поставщики услуг для каналов
<code>java.nio.charset</code>	Инкапсулирует наборы символов
<code>java.nio.charset.spi</code>	Предоставляет поставщики услуг для наборов символов
<code>java.nio.file</code>	Поддерживает систему ввода-вывода NIO для файлов
<code>java.nio.file.attribute</code>	Поддерживает атрибуты файлов в системе ввода-вывода NIO
<code>java.nio.file.spi</code>	Предоставляет поставщики услуг системы ввода-вывода NIO для файлов
<code>java.rmi</code>	Обеспечивает удаленный вызов методов
<code>java.rmi.activation</code>	Активизирует постоянные объекты
<code>java.rmi.dgc</code>	Управляет распределенной сборкой "мусора"

Окончание табл. 30.1

Пакет	Основное назначение
<code>java.rmi.registry</code>	Отображает имена на ссылки, которые делаются на удаленные объекты
<code>java.rmi.server</code>	Поддерживает удаленный вызов методов
<code>java.security</code>	Организует обработку сертификатов, ключей, сверток, подписей и выполнение других функций, связанных с соблюдением безопасности
<code>java.security.acl</code>	Поддерживает списки управления доступом
<code>java.security.cert</code>	Выполняет синтаксический анализ и управление сертификатами
<code>java.security.interfaces</code>	Определяет интерфейсы для ключей DSA (Digital Signature Algorithm – алгоритм цифровой подписи)
<code>java.security.spec</code>	Определяет параметры ключей и алгоритмов
<code>java.sql</code>	Организует взаимодействие с базой данных SQL (Structured Query Language – язык структурированных запросов)
<code>java.text</code>	Организует форматирование и поиск текста, а также манипулирование им
<code>java.text.spi</code>	Предоставляет поставщики услуг для классов форматирования текста из пакета <code>java.text</code>
<code>java.time</code>	Обеспечивает основную поддержку нового API даты и времени (добавлен в версии JDK 8)
<code>java.time.chrono</code>	Поддерживает другие календари, кроме григорианского (добавлен в версии JDK 8)
<code>java.time.format</code>	Поддерживает форматирование даты и времени (добавлен в версии JDK 8)
<code>java.time.temporal</code>	Поддерживает расширенные функции обработки даты и времени (добавлен в версии JDK 8)
<code>java.time.zone</code>	Поддерживает часовые пояса (добавлен в версии JDK 8)
<code>java.util</code>	Содержит общие утилиты
<code>java.util.concurrent</code>	Поддерживает утилиты параллелизма
<code>java.util.concurrent.atomic</code>	Поддерживает атомарные (т.е. неделимые) операции над переменными, не прибегая к блокировкам
<code>java.util.concurrent.locks</code>	Поддерживает синхронизированные блокировки
<code>java.util.jar</code>	Создает и читает архивные JAR-файлы
<code>java.util.logging</code>	Поддерживает протоколирование сведений о выполнении программы
<code>java.util.prefs</code>	Инкапсулирует сведения о пользовательских настройках
<code>java.util.regex</code>	Поддерживает обработку регулярных выражений
<code>Java.util.spl</code>	Предоставляет поставщики услуг для вспомогательных классов из пакета <code>java.util</code>
<code>java.util.zip</code>	Читает и записывает сжатые и несжатые архивные файлы формата ZIP

Обработка регулярных выражений

В пакете `java.util.regex` поддерживается обработка регулярных выражений. Употребляемый здесь термин *регулярное выражение* обозначает строку, описывающую последовательность символов. Это общее описание называется *шаблоном* и может быть впоследствии использовано для поиска совпадений в других последовательностях символов. В регулярных выражениях допускается определять метасимволы, наборы символов и различные кванторы. Таким образом, можно задать регулярное выражение, представляющее общую форму, совпадающую с разными конкретными последовательностями символов.

Обработка регулярных выражений поддерживается двумя классами: `Pattern` и `Matcher`. Эти классы действуют совместно. Регулярное выражение определяется в классе `Pattern`, а сопоставление последовательности символов с шаблоном осуществляется средствами класса `Matcher`.

Класс `Pattern`

В классе `Pattern` конструкторы не определяются. Вместо этого для составления шаблона вызывается фабричный метод `compile()`. Ниже приведена одна из общих форм этого метода.

```
static Pattern compile(String шаблон)
```

Здесь параметр *шаблон* обозначает регулярное выражение, которое требуется использовать. Метод `compile()` преобразует в шаблон символьную строку, определяемую параметром *шаблон*, для сопоставления средствами класса `Matcher`. Этот метод возвращает объект типа `Pattern`, содержащий шаблон.

Как только объект типа `Pattern` будет получен, его можно использовать для создания объекта типа `Matcher`. Для этого вызывается фабричный метод `matcher()`, определяемый в классе `Pattern`. Ниже приведена его общая форма.

```
Matcher matcher(CharSequence строка)
```

Здесь параметр *строка* обозначает последовательность символов, сопоставляемую с шаблоном и называемую *входной последовательностью*. В интерфейсе `CharSequence` определяется набор символов, доступных только для чтения. Среди прочих, он реализуется в классе `String`. Таким образом, методу `matcher()` можно передать символьную строку.

Класс `Matcher`

У этого класса отсутствуют конструкторы. Вместо этого для создания объекта класса `Matcher` вызывается фабричный метод `matcher()`, определяемый в классе `Pattern`, как пояснялось выше. Как только объект класса `Matcher` будет создан, его методы можно использовать для выполнения различных операций сопоставления с шаблоном.

Самым простым для сопоставления с шаблоном является метод `matches()`, который просто определяет, совпадает ли последовательность символов с шаблоном. Общая форма этого метода приведена ниже.

```
boolean matches()
```

Этот метод возвращает логическое значение `true`, если последовательность символов совпадает с шаблоном, а иначе — логическое значение `false`. Следует, однако, иметь в виду, что с шаблоном должна совпадать *вся* последовательность символов, а не только ее часть (т.е. подпоследовательность).

Чтобы определить, совпадает ли с шаблоном подпоследовательность из входной последовательности символов, следует вызвать метод `find()`. Ниже приведена одна из его общих форм.

```
boolean find()
```

Этот метод возвращает логическое значение `true`, если подпоследовательность совпадает с шаблоном, а иначе — логическое значение `false`. Метод `find()` можно вызывать неоднократно, чтобы находить все совпадающие подпоследовательности. При каждом вызове метода `find()` сравнение начинается с того места, где было завершено предыдущее сравнение.

Символьную строку, содержащую последнюю совпавшую последовательность, можно получить, вызвав метод `group()`. Ниже приведена одна из его общих форм.

```
String group()
```

Этот метод возвращает совпавшую символьную строку. Если ни одного совпадения не обнаружено, генерируется исключение типа `IllegalStateException`.

Вызвав метод `start()`, можно получить индекс текущего совпадения во входной последовательности символов, а индекс, следующий после текущего совпадения, — вызвав метод `end()`. Ниже приведены общие формы этих методов. В отсутствие совпадений оба эти метода генерируют исключение типа `IllegalStateException`.

```
int start()  
int end()
```

Каждую совпавшую последовательность символов можно заменить другой последовательностью, вызвав метод `replaceAll()`. Его общая форма выглядит следующим образом:

```
String replaceAll(String новая_строка)
```

где параметр *новая_строка* определяет новую последовательность символов, которая будет заменять последовательности, совпавшие с шаблоном. Обновленная входная последовательность будет возвращена в виде символьной строки.

Синтаксис регулярных выражений

Прежде чем продемонстрировать применение классов `Pattern` и `Matcher` на практике, следует пояснить, каким образом составляется регулярное выражение. Хотя ни одно из правил составления регулярных выражений нельзя назвать

сложным, их очень много, и поэтому описать полностью все эти правила в одной главе просто невозможно. Тем не менее ниже описываются некоторые из наиболее распространенных синтаксических конструкций регулярных выражений.

В общем, регулярное выражение состоит из обычных символов, классов символов (наборов символов), метасимволов и кванторов. Обычный символ сопоставляется в исходном виде. Так, если шаблон содержит пару символов "xy", то с этим шаблоном может совпасть только входная последовательность "xy". Символы вроде новой строки и табуляции указываются с помощью стандартных управляющих последовательностей, начинающихся со знака обратной косой черты (\). Например, символ новой строки обозначается управляющей последовательностью \n. В терминологии регулярных выражений обычный символ иначе называется *литералом*.

Класс символов является набором символов. Класс символов можно задать, заключив символы этого класса в квадратные скобки. Например, класс символов [wxyz] совпадает с символами w, x, y или z. Чтобы задать обратный набор символов, перед ними следует указать знак ^. Например, класс символов [^wxyz] совпадает с любым символом, кроме w, x, y и z. Диапазон символов указывается с помощью дефиса. Так, класс символов [1-9] совпадает с цифрами от 1 до 9.

Метасимволом служит знак точки (.), совпадающий с любым символом. Таким образом, шаблон ". ", состоящий только из знака точки, будет совпадать с любой из следующих (и других) входных последовательностей: "A", "a", "x" и т.д.

Квантор определяет, сколько раз совпадает выражение. В табл. 30.2 перечислены кванторы, применяемые в регулярных выражениях.

Таблица 30.2. Кванторы регулярных выражений

Квантор	Описание
+	Обозначает совпадение один раз или больше
*	Обозначает совпадение нуль или раз больше
?	Обозначает совпадение нуль или один раз

Например, шаблон "x+" будет совпадать с последовательностями символов "x", "xx", "xxx" и т.п. И наконец, следует иметь в виду, что если регулярное выражение составлено неверно, то будет сгенерировано исключение типа `PatternSyntaxException`.

Примеры, демонстрирующие совпадение с шаблоном

Чтобы стало понятнее, каким образом происходит сопоставление с шаблоном в регулярном выражении, рассмотрим несколько примеров. В первом приведенном ниже примере осуществляется поиск совпадения с литеральным шаблоном.

```
// Пример простого сопоставления с шаблоном
import java.util.regex.*;

class RegExpr {
    public static void main(String args[]) {
        Pattern pat;
```

```
Matcher mat;
boolean found;

pat = Pattern.compile("Java");
mat = pat.matcher("Java");
found = mat.matches(); // проверить на совпадение

System.out.println("Проверка совпадения Java с Java:");
if(found) System.out.println("Совпадает");
else System.out.println("Не совпадает");

System.out.println();

System.out.println("Проверка совпадения Java с Java 8:");
mat = pat.matcher("Java 8"); // создать новый сопоставитель
                               // с шаблоном
found = mat.matches(); // проверить на совпадение

if(found) System.out.println("Совпадает");
else System.out.println("Не совпадает");
}
}
```

Ниже приведен результат, выводимый данной программой.

Проверка совпадения Java с Java:
Совпадает

Проверка совпадения Java с Java 8:
Не совпадает

Проанализируем эту программу. Сначала в ней создается шаблон, состоящий из последовательности символов "Java". Затем для данного шаблона создается объект типа `Matcher` с входной последовательностью "Java". Далее вызывается метод `matches()`, с помощью которого определяется, совпадает ли входная последовательность с шаблоном. Входная последовательность совпадает с шаблоном, и поэтому метод `matches()` возвращает логическое значение `true`. После этого создается новый объект типа `Matcher` с входной последовательностью "Java 8" и снова вызывается метод `matches()`. В этом случае входная последовательность отличается от шаблона, и поэтому совпадение не обнаруживается. Напомним, что метод `matches()` возвращает логическое значение `true` только в том случае, если входная последовательность полностью совпадает с шаблоном. Он не возвратит логическое значение `true` только потому, что с шаблоном совпадает подпоследовательность, а не вся входная последовательность в целом.

Чтобы выяснить, содержит ли входная последовательность подпоследовательность, совпадающую с шаблоном, достаточно вызвать метод `find()`. Рассмотрим следующий пример программы:

```
// Использовать метод find() для нахождения подпоследовательности
import java.util.regex.*;

class RegExpr2 {
    public static void main(String args[]) {
        Pattern pat = Pattern.compile("Java");
        Matcher mat = pat.matcher("Java 8");

        System.out.println("Поиск Java в Java 8:");
```

```

        if(mat.find()) System.out.println(
                               "Подпоследовательность найдена");
        else System.out.println("Совпадения отсутствуют");
    }
}

```

Ниже приведен результат, выводимый данной программой. В данном случае метод `find()` осуществляет поиск подпоследовательности "Java" во входной последовательности символов.

Поиск Java в Java 8:
Подпоследовательность найдена

Метод `find()` можно использовать для поиска во входящей последовательности повторяющихся совпадений с шаблоном, поскольку каждый вызов метода `find()` начинается с того места, где был завершен предыдущий. Так, в следующем примере программы обнаруживаются два совпадения с шаблоном "test":

```

// Использовать метод find() для нахождения нескольких
// подпоследовательностей символов
import java.util.regex.*;

class RegExpr3 {
    public static void main(String args[]) {
        Pattern pat = Pattern.compile("test");

        Matcher mat = pat.matcher("test 1 2 3 test");
        while(mat.find()) {
            System.out.println(
                "Подпоследовательность test найдена по индексу "
                + mat.start());
        }
    }
}

```

Ниже приведены результаты, выводимые данной программой.

Подпоследовательность test найдена по индексу 0
Подпоследовательность test найдена по индексу 11

Как следует из результатов, в данной программе обнаружены два совпадения. Для получения индекса каждого совпадения в ней используется метод `start()`.

Применение метасимволов и кванторов

В предыдущем примере программы была продемонстрирована общая методика применения классов `Pattern` и `Matcher`, но она не раскрывает полностью их возможности. Подлинное преимущество, которое дает обработка регулярных выражений, невозможно ощутить, не применяя метасимволы и кванторы. Рассмотрим сначала следующий пример, где квантор `+` применяется для сопоставления с любой произвольной последовательностью символов `W`:

```

// Применить квантор
import java.util.regex.*;

class RegExpr4 {
    public static void main(String args[]) {

```

```
Pattern pat = Pattern.compile("W+");
Matcher mat = pat.matcher("W WW WWW");

while(mat.find())
    System.out.println("Совпадение: " + mat.group());
}
```

Ниже приведены результаты, выводимые данной программой. Как следует из результатов, шаблон "W+" в регулярном выражении совпадает с последовательностью символов W любой длины.

```
Совпадение: W
Совпадение: WW
Совпадение: WWW
```

В приведенном ниже примере программы метасимвол используется для составления шаблона, который должен сопоставляться с любой последовательностью, начинающейся с символа e и оканчивающейся символом d. Для этой цели в шаблоне указывается не только метасимвол точки, но и квантор +.

```
// Применить метасимвол и квантор
import java.util.regex.*;

class RegExpr5 {
    public static void main(String args[]) {
        Pattern pat = Pattern.compile("e.+d");
        Matcher mat = pat.matcher("extend cup end table");

        while(mat.find())
            System.out.println("Совпадение: " + mat.group());
    }
}
```

Приведенный ниже результат выполнения этой программы может показаться неожиданным.

```
Совпадение: extend cup end
```

При сопоставлении входной последовательности с шаблоном было обнаружено только одно совпадение. И это самая длинная последовательность, которая начинается с символа e и оканчивается символом d, хотя предполагалось обнаружить два совпадения: "extend" и "end". Более длинная последовательность была обнаружена потому, что по умолчанию метод find() обнаруживает совпадение с шаблоном самой длинной последовательности. Такое совпадение называется *строгим*. Но можно задать и *нестрогое* совпадение, если ввести в шаблон квантор ?, как показано в приведенной ниже версии программы из предыдущего примера. В итоге получается шаблон для сопоставления с самой короткой входной последовательностью.

```
// Применить квантор ?
import java.util.regex.*;

class RegExpr6 {
    public static void main(String args[]) {
        // составить шаблон для нестрогого совпадения
        Pattern pat = Pattern.compile("e.+?d");
        Matcher mat = pat.matcher("extend cup end table");
```



```

        while(mat.find())
            System.out.println("Совпадение: " + mat.group());
    }
}

```

Ниже приведены результаты, выводимые данной программой.

```

Совпадение: extend
Совпадение: end

```

Как следует из результатов, шаблон "e.+?d" совпадает с более короткой входной последовательностью, которая начинается с символа e и оканчивается символом d. Таким образом, обнаруживаются два совпадения.

Обращение с классами символов

Иногда возникает потребность обнаружить совпадение с последовательностью, состоящей из одного или нескольких произвольно расположенных символов, являющихся частью определенного набора символов. Например, чтобы обнаружить совпадение с целыми словами, придется искать совпадение с любой последовательностью букв алфавита. Это проще всего сделать с помощью класса символов, определяющего их набор. Напомним, чтобы создать класс символов, следует заключить сопоставляемые символы в квадратные скобки. Например, для сопоставления со строчными буквами от a до z служит класс символов [a-z]. В следующем примере программы демонстрируется применение класса символов:

```

// Применить класс символов
import java.util.regex.*;

class RegExpr7 {
    public static void main(String args[]) {
        // составить шаблон для сопоставления со словами,
        // набранными строчными буквами
        Pattern pat = Pattern.compile("[a-z]+");
        Matcher mat = pat.matcher("this is a test.");

        while(mat.find())
            System.out.println("Совпадение: " + mat.group());
    }
}

```

Ниже приведены результаты выполнения данной программы.

```

Совпадение: this
Совпадение: is
Совпадение: a
Совпадение: test

```

Применение метода `replaceAll()`

Метод `replaceAll()`, определяемый в классе `Matcher`, позволяет выполнять полноценные операции поиска и замены, в которых используются регулярные выражения. Так, в следующем примере программы каждая входная последовательность "Jon" заменяется выходной последовательностью "Eric":

```

// Применить метод replaceAll()
import java.util.regex.*;

```

```
class RegExpr8 {
    public static void main(String args[]) {
        String str = "Jon Jonathan Frank Ken Todd";

        Pattern pat = Pattern.compile("Jon.*? ");
        Matcher mat = pat.matcher(str);

        System.out.println("Исходная последовательность: " + str);
        str = mat.replaceAll("Eric ");

        System.out.println("Измененная последовательность: " + str);
    }
}
```

Ниже приведены результаты выполнения данной программы.

Исходная последовательность: Jon Jonathan Frank Ken Todd
Измененная последовательность: Eric Eric Frank Ken Todd

Регулярное выражение "Jon.*?" выполняет сопоставление с любой символьной строкой, начинающейся с последовательности символов Jon, после которой следует нуль или больше символов, и оканчивающейся пробелом, поэтому его можно применить для сопоставления и замены обоих имен Jon и Jonathan именем Eric. Такую замену нельзя было бы произвести без сопоставления с шаблоном.

Применение метода `split()`

С помощью метода `split()`, определяемого в классе `Pattern`, можно свести входную последовательность к отдельным лексемам. Ниже приведена одна из общих форм этого метода.

`String[] split(CharSequence строка)`

Метод `split()` обрабатывает входную последовательность, обозначаемую параметром строка, сводя ее к отдельным лексемам с разделителями, указываемыми в шаблоне. Так, в следующем примере программы обнаруживаются лексемы, разделяемые пробелами, запятыми, точками и знаками восклицания:

```
// Использовать метод split()
import java.util.regex.*;

class RegExpr9 {
    public static void main(String args[]) {

        // составить шаблон для сопоставления со словами,
        // набранными строчными буквами
        Pattern pat = Pattern.compile("[ ,.!]*");

        String strs[] = pat.split("one two,alpha9 12!done.");

        for(int i=0; i < strs.length; i++)
            System.out.println("Следующая лексема: " + strs[i]);
    }
}
```

Ниже приведены результаты выполнения данной программы.

```

Следующая лексема: one
Следующая лексема: two
Следующая лексема: alpha9
Следующая лексема: 12
Следующая лексема: done

```

Как следует из результатов, входная последовательность сводится к отдельным лексемам. Обратите внимание на отсутствие разделителей в выводимых результатах.

Два варианта сопоставления с шаблоном

Несмотря на то что описанные способы сопоставления с шаблонами отличаются высокой степенью гибкости и высокой производительностью, имеются два других варианта, которые могут оказаться полезными в определенных обстоятельствах. Так, если требуется только одноразовое сопоставление с шаблоном, то для этой цели можно воспользоваться методом `matches()`, определяемым в классе `Pattern`. Ниже приведена общая форма этого метода.

```
static boolean matches(String шаблон, CharSequence строка)
```

Метод `matches()` возвращает логическое значение `true`, если заданный шаблон совпадает с указанной строкой, а иначе — логическое значение `false`. Этот метод автоматически компилирует заданный шаблон, а затем осуществляет поиск на совпадение с ним. Для неоднократного применения шаблона вызов метода `matches()` оказывается менее эффективным, чем отдельная компиляция шаблона и сопоставление с ним с помощью методов, определяемых в классе `Matcher`, как пояснялось ранее.

Сопоставление с шаблоном можно также выполнить с помощью метода `matches()`, реализуемого в классе `String`. Ниже приведена общая форма этого метода. Если вызывающая строка совпадает с регулярным выражением, обозначаемым параметром *шаблон*, то метод `matches()` возвращает логическое значение `true`, а иначе — логическое значение `false`.

```
boolean matches(String шаблон)
```

Дальнейшее изучение регулярных выражений

Представленный выше краткий обзор регулярных выражений лишь отчасти раскрывает их подлинный потенциал. Синтаксический анализ, манипулирование и разбиение текста на лексемы является неотъемлемой частью программирования, и поэтому подсистема регулярных выражений в Java может стать удобным и полезным инструментальным средством. В связи с этим рекомендуется уделить время дальнейшему изучению свойств регулярных выражений. Поэкспериментируйте с различными видами шаблонов и входных последовательностей. Уяснив, каким образом осуществляется сопоставление с шаблонами в регулярных выражениях, вы найдете подобный прием вполне пригодным для решения многих задач программирования.

Рефлексия

Рефлексия — это способность программного обеспечения к самоанализу. Такая способность обеспечивается пакетом `java.lang.reflect` и элементами класса `Class`. Рефлексия является важным языковым средством, особенно для компонентов Java Beans. С ее помощью можно динамически анализировать компоненты программного обеспечения и описывать их свойства во время выполнения, а не компиляции. Например, с помощью рефлексии можно определить, какие методы, конструкторы и поля поддерживаются в конкретном классе. Обсуждение темы рефлексии было начато в главе 12, а в этой главе оно будет продолжено.

В состав пакета `java.lang.reflect` входит несколько интерфейсов. Особый интерес представляет интерфейс `Member`, в котором определяются методы, позволяющие получать сведения о поле, конструкторе или методе отдельного класса. В этом пакете имеется также восемь классов. Все они перечислены в табл. 30.3.

Таблица 30.3. Классы из пакета `java.lang.reflect`

Класс	Основное назначение
AccessibleObject	Позволяет обходить стандартные проверки управления доступом
Array	Позволяет динамически создавать массивы и манипулировать ими
Constructor	Предоставляет сведения о конструкторе
Field	Предоставляет сведения о поле
Method	Предоставляет сведения о методе
Modifier	Предоставляет сведения о модификаторах доступа к классу и его членам
Parameter	Предоставляет сведения о параметрах (добавлен в версии JDK 8)
Proxy	Поддерживает динамические прокси-классы
ReflectPermission	Разрешает рефлексии закрытых или защищенных членов класса

В приведенном ниже примере прикладной программы демонстрируется простой вариант применения свойств рефлексии в Java. Эта программа выводит на экран конструкторы, поля и методы из класса `java.awt.Dimension`. Сначала в этой программе вызывается метод `forName()` из класса `Class` для получения объекта класса `java.awt.Dimension`. Как только он будет получен, вызываются методы `getConstructors()`, `getFields()` и `getMethods()` для анализа объекта этого класса. Они возвращают массивы типа `Constructor`, `Field` и `Method`, содержащие сведения об этом объекте.

В классах `Constructor`, `Field` и `Method` определяется ряд методов, которые можно использовать для получения сведений об объекте. Вам придется изучить эти классы самостоятельно, но в каждом из них поддерживается метод `toString()`. Как показано в данной программе, указывать объекты типа `Constructor`, `Field` и `Method` в качестве параметров метода `println()` совсем не трудно.

```
// Продемонстрировать применение рефлексии
import java.lang.reflect.*;
public class ReflectionDemol {
    public static void main(String args[]) {
```

```

try {
    Class<?> c = Class.forName("java.awt.Dimension");
    System.out.println("Конструкторы:");
    Constructor constructors[] = c.getConstructors();
    for(int i = 0; i < constructors.length; i++) {
        System.out.println(" " + constructors[i]);
    }

    System.out.println("Поля:");
    Field fields[] = c.getFields();
    for(int i = 0; i < fields.length; i++) {
        System.out.println(" " + fields[i]);
    }

    System.out.println("Методы:");
    Method methods[] = c.getMethods();
    for(int i = 0; i < methods.length; i++) {
        System.out.println(" " + methods[i]);
    }
} catch(Exception e) {
    System.out.println("Исключение: " + e);
}
}
}

```

Ниже приведен примерный результат, выводимый данной программой (у вас он может оказаться иным).

Конструкторы:

```

public java.awt.Dimension(int, int)
public java.awt.Dimension()
public java.awt.Dimension(java.awt.Dimension)

```

Поля:

```

public int java.awt.Dimension.width
public int java.awt.Dimension.height

```

Методы:

```

public int java.awt.Dimension.hashCode()
public boolean java.awt.Dimension.equals(java.lang.Object)
public java.lang.String java.awt.Dimension.toString()
public java.awt.Dimension java.awt.Dimension.getSize()
public void java.awt.Dimension.setSize(double, double)
public void java.awt.Dimension.setSize(java.awt.Dimension)
public void java.awt.Dimension.setSize(int, int)
public double java.awt.Dimension.getHeight()
public double java.awt.Dimension.getWidth()
public java.lang.Object java.awt.geom.Dimension2D.clone()
public void java.awt.geom.Dimension2D.setSize(java.awt.geom.Dimension2D)
public final native java.lang.Class java.lang.Object.getClass()
public final native void java.lang.Object.wait(long)
    throws java.lang.InterruptedException
public final void java.lang.Object.wait()
    throws java.lang.InterruptedException
public final void java.lang.Object.wait(long, int)
    throws java.lang.InterruptedException
public final native void java.lang.Object.notify()
public final native void java.lang.Object.notifyAll()

```

В следующем примере программы возможности рефлексии в Java используются для получения открытых методов класса. Сначала в этой программе реализуется класс A. Метод `getClass()` вызывается по ссылке на объект этого класса и воз-

возвращает объект типа `Class` для описания класса `A`. Метод `getDeclaredMethods()` возвращает массив объектов типа `Method`, описывающий только методы, объявленные в классе `A`. В этот массив не включаются методы, унаследованные от суперклассов, например от класса `Object`.

После этого обрабатывается каждый элемент массива `methods`. В частности, метод `getModifiers()` возвращает значение типа `int`, содержащее признаки, описывающие модификаторы доступа, применяемые к этому элементу. В классе `Modifier` предоставляется ряд методов типа `isX`, перечисленных в табл. 30.4 и предназначенных для проверки заданного значения. Например, статический метод `isPublic()` возвращает логическое значение `true`, если в его аргумент включается модификатор доступа `public`, а иначе — логическое значение `false`.

Таблица 30.4. Методы из класса `Modifier`, определяющие модификатор доступа

Методы	Описание
<code>static boolean isAbstract(int значение)</code>	Возвращает логическое значение true , если заданное значение имеет установленный признак abstract , а иначе — логическое значение false
<code>static boolean isFinal(int значение)</code>	Возвращает логическое значение true , если заданное значение имеет установленный признак final , а иначе — логическое значение false
<code>static boolean isInterface(int значение)</code>	Возвращает логическое значение true , если заданное значение имеет установленный признак interface , а иначе — логическое значение false
<code>static boolean isNative(int значение)</code>	Возвращает логическое значение true , если заданное значение имеет установленный признак native , а иначе — логическое значение false
<code>static boolean isPrivate(int значение)</code>	Возвращает логическое значение true , если заданное значение имеет установленный признак private , а иначе — логическое значение false
<code>static boolean isProtected(int значение)</code>	Возвращает логическое значение true , если заданное значение имеет установленный признак protected , а иначе — логическое значение false
<code>static boolean isPublic(int значение)</code>	Возвращает логическое значение true , если заданное значение имеет установленный признак public , а иначе — логическое значение false
<code>static boolean isStatic(int значение)</code>	Возвращает логическое значение true , если заданное значение имеет установленный признак static , а иначе — логическое значение false
<code>static boolean isStrict(int значение)</code>	Возвращает логическое значение true , если заданное значение имеет установленный признак strict , а иначе — логическое значение false
<code>static boolean isSynchronized(int значение)</code>	Возвращает логическое значение true , если заданное значение имеет установленный признак synchronized , а иначе — логическое значение false

Методы	Описание
static boolean isTransient(int значение)	Возвращает логическое значение true , если заданное значение имеет установленный признак transient , а иначе — логическое значение false
static boolean isVolatile(int значение)	Возвращает логическое значение true , если заданное значение имеет установленный признак volatile , а иначе — логическое значение false

Если метод оказывается открытым, то его имя получается с помощью метода `getName()` и затем выводится на экран. Ниже приведен исходный код из данного примера программы.

```
// Показать открытые методы
import java.lang.reflect.*;
public class ReflectionDemo2 {
    public static void main(String args[]) {
        try {
            A a = new A();
            Class<?> c = a.getClass();
            System.out.println("Открытые методы:");
            Method methods[] = c.getDeclaredMethods();
            for(int i = 0; i < methods.length; i++) {
                int modifiers = methods[i].getModifiers();
                if(Modifier.isPublic(modifiers)) {
                    System.out.println(" " + methods[i].getName());
                }
            }
        } catch (Exception e) {
            System.out.println("Исключение: " + e);
        }
    }
}

class A {
    public void a1() {
    }
    public void a2() {
    }
    protected void a3() {
    }
    private void a4() {
    }
}
```

Эта программа выводит следующий результат:

```
Открытые методы:
a1
a2
```

Класс `Modifier` содержит также ряд статических методов, возвращающих тип модификаторов, которые могут быть применены к определенному типу элемента программы. Ниже приведены общие формы этих методов.

```
static int classModifiers()
static int constructorModifiers()
static int fieldModifiers()
```

```
static int interfaceModifiers()  
static int methodModifiers()  
static int parameterModifiers() (добавлен в версии JDK 8.)
```

Например, метод `methodModifiers()` возвращает модификаторы, которые могут быть применены к методу. Каждый метод возвращает признаки, скомпонованные в виде значения типа `int` и указывающие допустимые модификаторы. Значения модификаторов определяются константами `PROTECTED`, `PUBLIC`, `PRIVATE`, `STATIC`, `FINAL` и прочими, определенными в классе `Modifier`.

Удаленный вызов методов

Механизм удаленного вызова методов (RMI) позволяет объекту Java, выполняющемуся на одной машине, вызывать метод объекта Java, выполняющегося на другой машине. Это очень важный механизм, поскольку он позволяет разрабатывать распределенные приложения. Подробное обсуждение механизма RMI выходит за рамки данной книги, но в приведенном ниже упрощенном примере демонстрируются основные принципы его действия.

Простое приложение “клиент–сервер”, использующее механизм RMI

В этом разделе представлена пошаговая процедура разработки простого приложения “клиент–сервер” с использованием механизма RMI. Сервер получает запрос от клиента, обрабатывает его и возвращает результат. В данном примере приложения запрос состоит из двух чисел. Сервер суммирует их и возвращает полученный результат.

Шаг 1. Ввод и компиляция исходного кода

В рассматриваемом здесь приложении используются четыре исходных файла. В первом файле, именуемом `AddServerIntf.java`, определяется удаленный интерфейс, предоставляемый сервером. Он содержит один метод, принимающий два параметра типа `double` и возвращающий их сумму. Все удаленные интерфейсы должны расширять интерфейс `Remote`, входящий в пакет `java.rmi`. Никакие члены в интерфейсе `Remote` не определяются. Его основное назначение — указать на то, что в интерфейсе используются удаленные методы. Все удаленные методы могут генерировать исключение типа `RemoteException`, как показано ниже.

```
import java.rmi.*;  
public interface AddServerIntf extends Remote {  
    double add(double d1, double d2) throws RemoteException;  
}
```

Во втором исходном файле, `AddServerImpl.java`, реализуется удаленный интерфейс. Реализовать метод `add()` несложно. Удаленные объекты обычно относятся к классам, расширяющим интерфейс `UnicastRemoteObject`, как показано ниже. Этот интерфейс обеспечивает функциональные возможности, требующиеся для того, чтобы сделать доступными объекты на удаленных машинах.


```
import java.rmi.*;
import java.rmi.server.*;
public class AddServerImpl extends UnicastRemoteObject
implements AddServerIntf {
    public AddServerImpl() throws RemoteException {
    }
    public double add(double d1, double d2) throws RemoteException {
        return d1 + d2;
    }
}
```

Третий исходный файл, `AddServer.java`, содержит главную программу для серверной машины. Ее основное назначение — обновлять реестр RMI на этой машине, как показано ниже. Это делается с помощью метода `rebind()` из класса `Naming`, входящего в пакет `java.rmi`. Этот метод связывает имя со ссылкой на объект. В качестве первого параметра в методе `rebind()` указывается символьная строка, присваивающая серверу имя "AddServer", а в качестве второго параметра — ссылка на экземпляр класса `AddServerImpl`.

```
import java.net.*;
import java.rmi.*;
public class AddServer {
    public static void main(String args[]) {
        try {
            AddServerImpl addServerImpl = new AddServerImpl();
            Naming.rebind("AddServer", addServerImpl);
        } catch (Exception e) {
            System.out.println("Исключение: " + e);
        }
    }
}
```

В четвертом исходном файле, `AddClient.java`, реализуется клиентская часть распределенного приложения. Этому файлу требуются три аргумента командной строки. В первом из них указывается IP-адрес или имя серверной машины, а во втором и третьем параметрах — два суммируемых числа.

Приложение начинается с формирования символьной строки в соответствии с синтаксисом URL. В этом URL используется протокол `rmi`. Символьная строка включает в себя IP-адрес или имя сервера и подстроку "AddServer". Затем вызывается метод `lookup()` из класса `Naming`. Этот метод принимает URL по протоколу `rmi` в качестве единственного параметра и возвращает ссылку на объект типа `AddServerIntf`. Все последующие вызовы удаленных методов могут быть направлены этому объекту. Далее выводятся аргументы данной прикладной программы и вызывается удаленный метод `add()`, возвращающий результат суммирования, который затем выводится на экран, как показано ниже. Введя весь исходный код, воспользуйтесь компилятором `javac`, чтобы скомпилировать четыре созданных вами исходных файла.

```
import java.rmi.*;
public class AddClient {
    public static void main(String args[]) {
        try {
            String addServerURL = "rmi://" + args[0] + "/AddServer";
            AddServerIntf addServerIntf =
                (AddServerIntf) Naming.lookup(addServerURL);
```

```

        System.out.println("Первое число: " + args[1]);
        double d1 = Double.valueOf(args[1]).doubleValue();
        System.out.println("Второе число: " + args[2]);
        double d2 = Double.valueOf(args[2]).doubleValue();
        System.out.println("Сумма: " + addServerIntf.add(d1, d2));
    } catch (Exception e) {
        System.out.println("Исключение: " + e);
    }
}
}
}

```

Шаг 2. Создание, если требуется, заглушки вручную

В контексте RMI *заглушка* представляет собой объект Java, находящийся на клиентском машине. Назначение заглушки – представить те же самые интерфейсы, что и на удаленном сервере. Вызовы удаленных методов, иницилируемые клиентом, направляются заглушке, которая взаимодействует с остальными частями системы RMI для составления запроса, направляемого дистанционной машине.

Удаленный метод может принимать в качестве параметров значения примитивных типов или объекты. В последнем случае указываемый объект может иметь ссылки на другие объекты. Вся эти данные должны быть отправлены удаленной машине. Следовательно, объект, передаваемый в качестве аргумента при вызове удаленного метода, должен быть сериализован и отправлен удаленной машине. Как пояснялось в главе 20, при сериализации рекурсивно обрабатываются также все объекты, на которые делаются ссылки.

Если клиенту требуется вернуть ответ, весь процесс происходит в обратном порядке. Следует иметь в виду, что сериализация и десериализация выполняются и при возвращении объектов клиенту.

До версии Java 5 заглушки приходилось создавать вручную, используя компилятор `rmic`, а в современных версиях Java эта стадия уже необязательна. Но если вы работаете в устаревшей среде, то для создания заглушки можете воспользоваться компилятором `rmic` следующим образом:

```
rmic AddServerImpl
```

По этой команде создается файл `AddServerImpl_Stub.class`. Используя компилятор `rmic`, включите текущий каталог в переменную окружения `CLASSPATH`.

Шаг 3. Установка файлов на серверной и клиентской машинах

Скопируйте файлы `AddClient.class`, `AddServerImpl_Stub.class` (если требуется) и `AddServerIntf.class` в каталог на клиентской машине, а файлы `AddServerIntf.class`, `AddServerImpl.class`, `AddServerImpl_Stub.class` (если требуется) и `AddServer.class` – в каталог на серверной машине.

На заметку! Механизм RMI поддерживает технологии динамической загрузки классов, но они не применяются в рассматриваемом здесь примере. Вместо этого все файлы, используемые в приложениях “клиент–сервер”, должны быть установлены вручную на соответствующих машинах.

Шаг 4. Запуск реестра RMI на серверной машине

В комплект JDK входит утилита **rmiregistry**, которая выполняется на стороне сервера. Она преобразует имена в ссылки на объекты. Сначала проверьте, включен ли каталог, где находятся ваши файлы, в переменную окружения `CLASSPATH`. Затем запустите реестр RMI из командной строки по следующей команде:

```
start rmiregistry
```

По завершении этой команды на экране появится новое окно. Это окно должно быть открыто на весь период экспериментирования с рассматриваемым здесь примером приложения, использующего механизм RMI.

Шаг 5. Запуск сервера

Серверная часть рассматриваемого здесь приложения запускается из командной строки по следующей команде:

```
java AddServer
```

Напомним, что в классе `AddServer` получается экземпляр класса `AddServerImpl`, который регистрируется под именем "AddServer".

Шаг 6. Запуск клиента

Для выполнения клиентской части `AddClient` рассматриваемого здесь приложения требуется указать следующие три аргумента: имя или IP-адрес серверной машины и два числа, которые должны быть просуммированы. Клиентскую часть данного приложения можно вызвать из командной строки в одном из следующих форматов:

```
java AddClient server1 8 9
java AddClient 11.12.13.14 8 9
```

В первом формате указывается имя сервера, а во втором формате — его IP-адрес (11.12.13.14). Рассматриваемое здесь приложение можно попытаться выполнить и без удаленного сервера. Для этого достаточно установить все части этого приложения на одном и том же компьютере и запустить сначала утилиту **rmiregistry**, затем серверную часть `AddServer`, а после этого клиентскую часть `AddClient` следующим образом:

```
java AddClient 127.0.0.1 8 9
```

где 127.0.0.1 обозначает адрес обратной связи локальной машины. Благодаря этому адресу можно испытать весь механизм RMI, не устанавливая сервер на удаленной машине. Но в любом случае результат выполнения данного приложения будет следующим:

```
Первое число: 8
Второе число: 9
Сумма: 17.0
```

На заметку! Применяя механизм RMI на практике, вероятнее всего, придется установить на сервере диспетчер безопасности.

Форматирование даты и времени средствами пакета `java.text`

В пакете `java.text` предоставляются средства для форматирования, синтаксического анализа, поиска и манипулирования текстом. В этом разделе рассматриваются два наиболее употребительных класса из этого пакета, предназначенных для форматирования даты и времени. Следует, однако, сразу же заметить, что в рассматриваемом далее новом API даты и времени предлагается современный подход к обработке даты и времени, который поддерживает также форматирование. Разумеется, рассматриваемые здесь классы будут еще некоторое время применяться в унаследованном коде.

Класс `DateFormat`

Класс `DateFormat` является абстрактным и позволяет форматировать и анализировать дату и время. В частности, метод `getDateInstance()` возвращает экземпляр класса `DateFormat`, способный форматировать информацию о дате. Имеются следующие общие формы этого метода:

```
static final DateFormat getDateInstance()
static final DateFormat getDateInstance(int стиль)
static final DateFormat getDateInstance(int стиль,
                                         Locale региональные_настройки)
```

Параметр *стиль* может принимать значение одной из следующих констант: `DEFAULT`, `SHORT`, `MEDIUM`, `LONG` или `FULL`. Это константы типа `int`, определяемые в классе `DateFormat`. Они позволяют представить различные подробности, касающиеся даты. Параметр *региональные_настройки* принимает одну из статических ссылок, определяемых в классе `Locale`, как поясняется в главе 19. Если параметры *стиль* и/или *региональные_настройки* не указаны, то используются значения, устанавливаемые по умолчанию.

К числу наиболее употребительных из класса `DateFormat` относится метод `format()`. У него имеется несколько перегружаемых форм, одна из которых приведена ниже.

```
final String format(Date d)
```

В качестве аргумента этого метода указывается объект типа `Date`, представляющий отображаемую дату. Метод `format()` возвращает символьную строку, содержащую отформатированную информацию о дате.

В приведенном ниже примере программы демонстрируется форматирование сведений о дате. Сначала в этой программе создается объект класса `Date`. В этом объекте хранятся сведения о текущих дате и времени. Затем сведения о дате выводятся отформатированными в разных стилях и с учетом региональных особенностей представления дат.

```
// Продемонстрировать различные форматы дат
import java.text.*;
import java.util.*;
```

```

public class DateFormatDemo {
    public static void main(String args[]) {
        Date date = new Date();
        DateFormat df;

        df = DateFormat.getDateInstance(DateFormat.SHORT, Locale.JAPAN);
        System.out.println("Япония: " + df.format(date));

        df = DateFormat.getDateInstance(DateFormat.MEDIUM, Locale.KOREA);
        System.out.println("Корея: " + df.format(date));

        df = DateFormat.getDateInstance(DateFormat.LONG, Locale.UK);
        System.out.println("Великобритания: " + df.format(date));

        df = DateFormat.getDateInstance(DateFormat.FULL, Locale.US);
        System.out.println("США: " + df.format(date));
    }
}

```

Ниже приведен примерный результат, выводимый данной программой.

```

Япония: 14/01/01
Корея: 2014. 1. 1
Великобритания: 01 January 2014
США: Saturday, January 1, 2014

```

Метод `getTimeInstance()` возвращает экземпляр класса `DateFormat`, способный форматировать сведения о времени. Этот метод имеет следующие общие формы:

```

static final DateFormat getTimeInstance()
static final DateFormat getTimeInstance(int стиль)
static final DateFormat getTimeInstance(int стиль,
                                         Locale региональные_настройки)

```

Параметр *стиль* может принимать значение одной из следующих констант: `DEFAULT`, `SHORT`, `MEDIUM`, `LONG` и `FULL`. Это константы типа `int`, определяемые в классе `DateFormat`. Они позволяют представить различные подробности, касающиеся времени. Параметр *региональные_настройки* принимает одну из статических ссылок, определяемых в классе `Locale`, как поясняется в главе 19. Если параметры *стиль* и/или *региональные_настройки* не указаны, то используются значения, устанавливаемые по умолчанию.

В приведенном ниже примере программы демонстрируется форматирование сведений о времени. Сначала в этой программе создается объект класса `Date`. В этом объекте хранятся сведения о текущих дате и времени. Затем сведения о времени выводятся отформатированными в разных стилях и с учетом региональных особенностей представления времени.

```

// Продемонстрировать различные форматы времени
import java.text.*;
import java.util.*;
public class TimeFormatDemo {
    public static void main(String args[]) {
        Date date = new Date();
        DateFormat df;
        df = DateFormat.getTimeInstance(DateFormat.SHORT, Locale.JAPAN);
        System.out.println("Япония: " + df.format(date));
        df = DateFormat.getTimeInstance(DateFormat.LONG, Locale.UK);
        System.out.println("Великобритания: " + df.format(date));
        df = DateFormat.getTimeInstance(DateFormat.FULL, Locale.CANADA);
    }
}

```

```
        System.out.println("Канада: " + df.format(date));  
    }  
}
```

Ниже приведен примерный результат, выводимый данной программой.

Япония: 13:06
Великобритания: 13:06:53 CST
Канада: 1:06:53 o'clock PM CST

В классе `DateFormat` имеется также метод `getDateTImeInstance()`, способный форматировать сведения о дате и времени. Можете поэкспериментировать с ним самостоятельно.

Класс `SimpleDateFormat`

Класс `SimpleDateFormat` является конкретным подклассом, производным от класса `DateFormat`. Он позволяет определять собственные шаблоны форматирования, чтобы использовать их для отображения сведений о дате и времени. Ниже показан один из конструкторов этого класса.

`SimpleDateFormat(String форматирующая_строка)`

Параметр *форматирующая_строка* обозначает способ отображения даты и времени. В следующей строке кода демонстрируется один из примеров применения конструктора класса `SimpleDateFormat`:

```
SimpleDateFormat sdf = SimpleDateFormat("dd MMM yyyy hh:mm:ss zzz");
```

Символы, указываемые в форматирующей строке, определяют порядок отображения сведений о дате и времени. Все эти символы и их описание приведены в табл. 30.5.

Таблица 30.5. Символы форматирующей строки для класса `SimpleDateFormat`

Символ	Описание
a	АМ (до полудня) или РМ (после полудня)
d	День месяца (1–31)
h	Часы в формате АМ/РМ (1–12)
k	Часы в формате суток (1–24)
m	Минуты (0–59)
s	Секунды (0–59)
w	Неделя в году (1–52)
Y	Год
z	Часовой пояс
D	День в году (1–366)
E	День недели (например, четверг)
F	День недели в месяце

Символ	Описание
G	Эра (AD — после Рождества Христова, или нашей эры, BC — до Рождества Христова, или до нашей эры)
H	Часы в сутках (0–23)
K	Часы в формате AM/PM (0–11)
M	Месяц
S	Миллисекунды в секунде
W	Неделя в месяце (1–5)
X	Часовой пояс в формате по стандарту ISO 8601
Y	Неделя в году
Z	Часовой пояс в формате по стандарту RFC 822

Как правило, количество повторений символа определяет способ представления даты. Текстовая информация отображается в сокращенной форме, если буква шаблона повторяется меньше четырех раз. В противном случае используется не-сокращенная форма. Например, шаблон **zzzz** позволяет отобразить часовой пояс Pacific Daylight Time (тихоокеанское время), а шаблон **zzz** — этот же часовой пояс в сокращенной форме PDT.

Количество повторений буквы шаблона определяет количество цифр в числовом представлении времени. Например, шаблон **hh:mm:ss** позволяет представить время в формате 01:51:15, тогда как шаблон **h:m:s** отображает то же самое время в формате 1:51:15.

И наконец, символы **M** или **MM** обозначают отображение месяца в виде одной или двух цифр. Три или больше повторения символа **M** обуславливают отображение месяца в виде текстовой строки.

В следующей программе демонстрируется применение класса `SimpleDateFormat`:

```
// Продемонстрировать применение класса SimpleDateFormat
import java.text.*;
import java.util.*;

public class SimpleDateFormatDemo {
    public static void main(String args[]) {
        Date date = new Date();
        SimpleDateFormat sdf;
        sdf = new SimpleDateFormat("hh:mm:ss");
        System.out.println(sdf.format(date));
        sdf = new SimpleDateFormat("dd MMM yyyy hh:mm:ss zzz");
        System.out.println(sdf.format(date));
        sdf = new SimpleDateFormat("E MMM dd yyyy");
        System.out.println(sdf.format(date));
    }
}
```

Ниже приведен примерный результат, выводимый данной программой.

```
01:30:51
01 Jan 2014 01:30:51 CST
Wed Jan 01 2014
```

API даты и времени, внедренный в версии JDK 8

В главе 19 был представлен давно устоявшийся в Java подход к обработке даты и времени средствами классов `Calendar` и `GregorianCalendar`. На момент написания данной книги этот традиционный подход был по-прежнему широко распространен, и поэтому он должен быть известен программирующим на Java. Но в версии JDK 8 в Java был предложен другой подход к обработке даты и времени, который определен в пакетах, перечисленных в табл. 30.6.

Таблица 30.6. Пакеты нового API даты и времени

Пакет	Описание
<code>java.time</code>	Предоставляет классы верхнего уровня для поддержки даты и времени
<code>java.time.chrono</code>	Поддерживает другие календари, кроме григорианского
<code>java.time.format</code>	Поддерживает форматирование даты и времени
<code>java.time.temporal</code>	Поддерживает расширенные функциональные возможности обработки даты и времени
<code>java.time.zone</code>	Поддерживает часовые пояса

В этих новых пакетах определяется большое количество классов, интерфейсов и перечислений, обеспечивающих обширную и подробную поддержку операций обработки даты и времени. Такое большое количество элементов, составляющих новый API даты и времени, поначалу выглядит немного пугающе. Но этот API хорошо организован и логически структурирован. Его размеры отражают степень детализации и гибкость управления, предоставляемые для обработки даты и времени. В рамках данной книги просто невозможно описать каждый элемент этого обширного API, поэтому ниже будут рассмотрены лишь самые основные его классы. В ходе обсуждения этих классов станет ясно, что их достаточно для применения во многих случаях.

Основные классы даты и времени

В пакете `java.time` определяется несколько классов верхнего уровня, упрощающих доступ к дате и времени. К их числу относятся три класса: `LocalDate`, `LocalTime` и `LocalDateTime`. Как следует из имен этих классов, они инкапсулируют локальную дату, локальное время, локальные дату и время соответственно. С помощью этих классов нетрудно получить текущие дату и время, отформатировать дату и время, сравнить даты и время, а также выполнить над ними другие операции.

Класс `LocalDate` инкапсулирует дату, для представления которой используется выбираемый по умолчанию григорианский календарь, как определено в стандарте ISO 8601. Класс `LocalTime` инкапсулирует время по стандарту ISO 8601, а класс `LocalDateTime` инкапсулирует дату и время. Эти классы содержат большое количество методов, предоставляющих доступ к составляющим даты и времени, позволяющих сравнивать даты и время, складывать и вычитать составляющие даты или времени и выполнять прочие операции над ними. Усвоив один из этих методов,

нетрудно овладеть остальными благодаря принятым для них общими условными обозначениями.

В классах `LocalDate`, `LocalTime` и `LocalDateTime` открытые конструкторы не применяются. Вместо этого для получения экземпляра соответствующего класса служит фабричный метод. Одним из удобных для этих целей является метод `now()`, определяемый во всех трех классах. Он возвращает текущую системную дату и/или время. В каждом из трех рассматриваемых здесь классов определяется несколько форм этого метода, но здесь будет использована самая простая из них. Так, ниже приведена общая форма метода `now()`, определенная в классе `LocalDate`.

```
static LocalDate now()
```

Форма этого метода определяется в классе `LocalTime` следующим образом:

```
static LocalTime now()
```

А его форма в классе `LocalDateTime` определяется так:

```
static LocalDateTime now()
```

В каждом случае метод `now()` возвращает соответствующий объект, который может быть отображен в своей исходной удобочитаемой форме, например методом `println()`. Но в то же время имеется возможность полностью управлять процессом форматирования даты и времени.

В приведенном ниже примере программы демонстрируется применение классов `LocalDate` и `LocalTime` для получения и отображения текущих даты и времени. Обратите внимание на вызов метода `now()` для извлечения текущих даты и времени.

```
// Простой пример применения классов LocalDate и LocalTime  
import java.time.*;
```

```
class DateTimeDemo {  
    public static void main(String args[]) {  
  
        LocalDate curDate = LocalDate.now();  
        System.out.println(curDate);  
  
        LocalTime curTime = LocalTime.now();  
        System.out.println(curTime);  
    }  
}
```

Ниже приведен примерный результат, выводимый данной программой. Полученный результат отражает стандартный формат, который получают дата и время. (В следующем разделе будет показано, как указать другой формат.)

```
2014-01-01  
14:03:41.436
```

В предыдущем примере программы отображаются текущие дата и время, но это было бы проще сделать средствами класса `LocalDateTime`. В этом случае потребовалось бы создать единственный экземпляр данного класса и сделать единственный вызов его метода `now()`, как показано ниже.

```
LocalDateTime curDateTime = LocalDateTime.now();  
System.out.println(curDateTime);
```

При таком подходе дата и время выводятся по умолчанию вместе. Ниже приведен пример такого вывода.

```
2014-01-01T14:04:56.799
```

Следует также заметить, что из экземпляра класса `LocalDateTime` можно также получить ссылку на составляющую даты или времени, вызвав метод `toLocalDate()` или `toLocalTime()` соответственно. Ниже приведены общие формы этих методов. Каждый из них возвращает ссылку на указанную составляющую.

```
LocalDate toLocalDate()
LocalTime toLocalTime()
```

Форматирование даты и времени

Стандартные форматы даты и времени, продемонстрированные в предыдущих примерах, оказываются пригодными лишь в некоторых случаях, а зачастую требуется указывать другой формат. Правда, сделать это совсем не трудно, поскольку в классах `LocalDate`, `LocalTime` и `LocalDateTime` для этой цели предоставляется метод `format()`. Ниже приведена общая форма этого метода, где параметр *форматирующий_объект* обозначает экземпляр класса `DateTimeFormatter`, предоставляющий требуемый формат.

```
String format(DateTimeFormatter форматирующий_объект)
```

Класс `DateTimeFormatter` входит в состав пакета `java.time.format`. Для получения экземпляра класса `DateTimeFormatter`, как правило, вызывается один из фабричных методов. Ниже приведены общие формы трех таких методов.

```
static DateTimeFormatter ofLocalizedDate(FormatStyle формат_даты)
static DateTimeFormatter ofLocalizedTime(FormatStyle формат_времени)
static DateTimeFormatter ofLocalizedDateTime(FormatStyle формат_даты,
                                             FormatStyle формат_времени)
```

Безусловно, тип средства форматирования даты и времени, создаваемого в виде экземпляра класса `DateTimeFormatter`, будет зависеть от типа того объекта, которым он оперирует. Так, если требуется отформатировать дату, хранящуюся в экземпляре класса `LocalDate`, то следует вызвать метод `ofLocalizedDate()`. Конкретный формат указывается в виде параметра типа `FormatStyle`.

Перечисление `FormatStyle` входит в состав пакета `java.time.format`. В нем определяются приведенные ниже константы.

```
FULL
LONG
MEDIUM
SHORT
```

Эти константы обозначают уровень детализации отображаемых даты и времени. Следовательно, данная форма класса `DateTimeFormatter` действует аналогично классу `DateFormat` из пакета `java.text`, описанному ранее в этой главе.

В следующем примере программы демонстрируется применение класса `DateTimeFormatter` для отображения текущих даты и времени:

```
// Продемонстрировать применение класса DateTimeFormatter
import java.time.*;
```

```
import java.time.format.*;

class DateTimeDemo2 {
    public static void main(String args[]) {

        LocalDate curDate = LocalDate.now();
        System.out.println(curDate.format(
            DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL)));

        LocalTime curTime = LocalTime.now();
        System.out.println(curTime.format(
            DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT)));
    }
}
```

Ниже приведен примерный результат, выводимый данной программой.

```
Wednesday, January 1, 2014
2:16 PM
```

Иногда требуется формат, отличающийся от тех, что указываются с помощью констант из перечисления `FormatStyle`. Для этой цели можно, в частности, воспользоваться предопределенным средством форматирования, например, `ISO_DATE` или `ISO_TIME` из класса `DateTimeFormatter`. А с другой стороны, можно создать специальный формат, указав шаблон. Для этого достаточно вызвать фабричный метод `ofPattern()` из класса `DateTimeFormatter`. Ниже приведена одна из общих форм этого метода.

```
static DateTimeFormatter ofPattern(String шаблон_форматирования)
```

Здесь параметр *шаблон_форматирования* обозначает символьную строку, содержащую шаблон, требующийся для форматирования даты и времени. Этот метод возвращает объект типа `DateTimeFormatter` для форматирования по заданному шаблону с учетом региональных настроек по умолчанию.

В общем, шаблон состоит из спецификаторов формата, называемых иначе *буквами шаблона*. Каждая буква шаблона заменяется той составляющей даты или времени, которую она обозначает. Полный перечень букв шаблона приведен в описании метода `ofPattern()` из документации на API даты и времени, а в табл. 30.7 перечислены лишь некоторые из них. Следует иметь в виду, что буквы шаблона указываются с учетом регистра.

В общем, конкретный выводимый результат зависит от количества повторений буквы в шаблоне. (Следовательно, класс `DateTimeFormatter` действует в какой-то степени аналогично классу `SimpleDateFormat` из класса `java.text`, описанного ранее в этой главе.) Например, следующий шаблон для вывода даты в апреле месяце:

```
М ММ МММ ММММ
```

позволяет вывести отформатированную дату, как показано ниже. Откровенно говоря, самый лучший способ уяснить назначение каждой буквы шаблона и воздействие различных повторений на вывод — экспериментирование.

```
4 04 Apr April
```

Если букву шаблона требуется вывести как текст, ее следует заключить в одинарные кавычки. Таким образом, все символы, не относящиеся к шаблону, рекомендуется заключать в одинарные кавычки, чтобы избежать осложнений при изменении букв шаблона в последующих версиях Java.

Таблица 30.7. Избранные буквы шаблона

Буква шаблона	Назначение
a	Обозначает время до полудня (AM) или после полудня (PM)
d	День месяца
E	День недели
h	Час в 12-часовом формате
H	Час в 24-часовом формате
M	Месяц
m	Минуты
s	Секунды
y	Год

В следующем примере программы демонстрируется применение шаблона даты и времени:

```
// Создание специального формата даты и времени
import java.time.*;
import java.time.format.*;

class DateTimeDemo3 {
    public static void main(String args[]) {

        LocalDateTime curDateTime = LocalDateTime.now();
        System.out.println(curDateTime.format(
            DateTimeFormatter.ofPattern("MMMM d", ' yyyy h':'mm a')));
    }
}
```

Ниже приведен примерный результат, выводимый данной программой.

January 1, 2014 2:22 PM

В отношении создания специального формата вывода даты и времени следует также заметить следующее: в классах `LocalDate`, `LocalTime` и `LocalDateTime` определяются методы, позволяющие получать различные составляющие даты и времени. Например, метод `getHour()` возвращает час в виде целочисленного значения типа `int`, метод `getMonth()` — месяц в виде значения перечислимого типа `Month`, а метод `getYear()` — год в виде целочисленного значения типа `int`. Используя эти и другие методы, можно сформировать вывод даты и времени вручную. Получаемыми в итоге значениями можно воспользоваться и в других целях, например, при создании специальных таймеров.

Синтаксический анализ символьных строк даты и времени

В классах `LocalDate`, `LocalTime` и `LocalDateTime` предоставляются средства для синтаксического анализа символьных строк даты и/или времени. С этой целью вызывается метод `parse()` для экземпляра одного из упомянутых классов. У этого метода имеются две формы. В первой форме применяется выбираемое по умолчанию средство форматирования и выполняется синтаксический анализ даты и/или времени, отформатированных по стандарту ISO, например, времени в формате `03:31` и даты в формате `2014-08-02`. Ниже приведена общая форма

метода `parse()` для класса `LocalDateTime`. (Его общие формы для других классов аналогичны, за исключением типа возвращаемого объекта.)

```
static LocalDateTime parse(CharSequence строка_даты_времени)
```

Здесь параметр *строка_даты_времени* обозначает символьную строку, содержащую дату и время в надлежащем формате. Если же указан недействительный формат, возникнет ошибка.

Если требуется синтаксический анализ символьной строки даты и/или времени в другом формате, а не по стандарту ISO, то для этой цели можно воспользоваться второй формой метода `parse()`, позволяющей указать собственное средство форматирования. Ниже приведена общая форма этого метода для класса `LocalDateTime`, где *средство_форматирования_даты_времени* обозначает требуемое средство форматирования. (Его общие формы для других классов аналогичны, за исключением типа возвращаемого объекта.)

```
static LocalDateTime parse(CharSequence строка_даты_времени,  
    DateTimeFormatter средство_форматирования_даты_времени)
```

В следующем простом примере демонстрируется синтаксический анализ символьной строки даты и времени с помощью специального средства форматирования:

```
// Пример синтаксического анализа даты и времени
import java.time.*;
import java.time.format.*;

class DateTimeDemo4 {
    public static void main(String args[]) {

        // получить объект типа LocalDateTime, выполнив
        // синтаксический анализ символьной строки даты и времени
        LocalDateTime curDateTime =
            LocalDateTime.parse("June 21, 2014 12:01 AM",
                DateTimeFormatter.ofPattern("MMMM d', ' yyyy hh':'mm a"));

        // а теперь отобразить проанализированные дату и время
        System.out.println(curDateTime.format(
            DateTimeFormatter.ofPattern("MMMM d', ' yyyy h':'mm a")));
    }
}
```

Ниже приведен примерный результат, выводимый данной программой.

```
June 21, 2014 12:01 AM
```

Дальнейшее изучение пакета `java.time`

Начать доскональное изучение всех пакетов даты и времени лучше всего с пакета `java.time`, который содержит немало полезных средств. Начните его изучение с методов, определенных в классах `LocalDate`, `LocalTime` и `LocalDateTime`. В каждом из этих классов имеются методы, позволяющие среди прочего складывать, вычитать, корректировать по заданной составляющей или сравнивать даты и/или время, а также создавать экземпляры, исходя из составляющих даты и/или времени. В числе других классов из пакета `java.time` особый интерес могут представлять `Instant`, `Duration` и `Period`. В частности, класс `Instant` инкапсулирует момент времени; класс `Duration` — протяженность времени; класс `Period` — протяженность даты.

ЧАСТЬ

III

Введение в программирование ГПИ средствами Swing

ГЛАВА 31

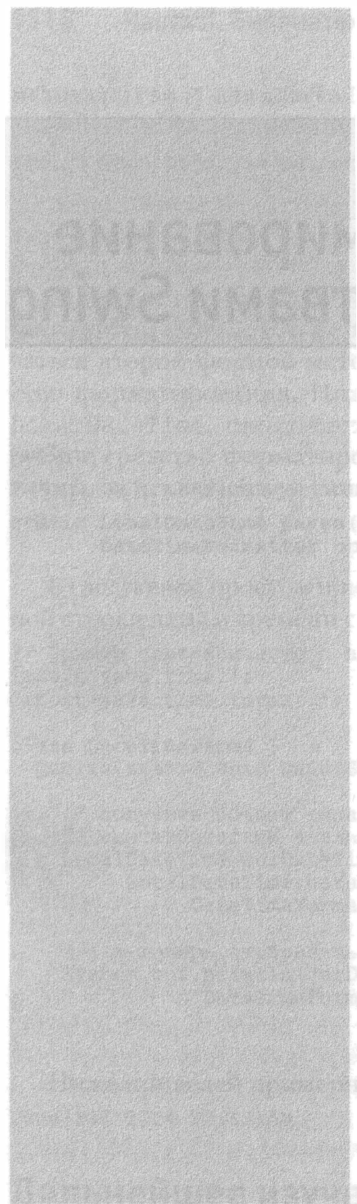
Введение
в библиотеку Swing

ГЛАВА 32

Исследование
библиотеки Swing

ГЛАВА 33

Введение в меню Swing



Введение в библиотеку Swing

Как было показано в части II данной книги, с помощью классов из библиотеки AWT можно создавать графические пользовательские интерфейсы (ГПИ). Несмотря на то что библиотека AWT по-прежнему является важной частью Java, ее компоненты применяются для создания ГПИ уже не так широко. Ныне многие программирующие на Java пользуются для этой цели библиотеками Swing и JavaFX. Подробнее библиотека JavaFX рассматривается в части IV данной книги, а в этой части представлено введение в библиотеку Swing. В этой библиотеке предоставляются более эффективные и гибкие компоненты ГПИ, чем в библиотеке AWT. Именно поэтому библиотека Swing уже более десятилетия широко применяется для построения ГПИ прикладных программ на Java.

Рассмотрению библиотеки Swing посвящены три главы части III. В этой главе представлено введение в библиотеку Swing. Сначала в ней поясняются основные принципы построения библиотеки Swing. Затем демонстрируется общая форма прикладных программ на основе Swing, включая приложения и апплеты. А в конце главы поясняется, каким образом выполняется рисование графики средствами библиотеки Swing. Во второй главе из этой части будет рассмотрен ряд наиболее употребительных компонентов библиотеки Swing, а в третьей главе — меню, создаваемые на основе Swing. Следует, однако, иметь в виду, что количество классов и интерфейсов в пакетах Swing довольно велико, поэтому просто невозможно рассмотреть каждый из них подробно в данной книге. (В действительности потребуется отдельная книга, чтобы полностью осветить библиотеку Swing.) Тем не менее в трех главах этой части даются основы того, что нужно знать о библиотеке Swing.

На заметку! Более содержательное описание библиотеки Swing можно найти в другой книге автора — *Swing: A Beginner's Guide*, издательство McGraw-Hill Professional, 2007 г. В русском переводе книга вышла под названием *Swing. Руководство для начинающих* в ИД “Вильямс”, 2007 г.

Происхождение библиотеки Swing

Библиотека Swing появилась в Java не с самого начала. Причиной разработки классов библиотеки Swing стали недостатки, обнаружившиеся в библиотеке AWT — исходной подсистеме Java для построения ГПИ. В библиотеке AWT определяется базовый набор элементов управления, обычных и диалоговых окон, поддержи-

вающий пригодный для применения ГПИ, но имеющий ограниченные возможности. Одна из причин ограниченности библиотеки AWT состоит в том, что она преобразует свои визуальные компоненты в соответствующие им платформенно-зависимые эквиваленты, называемые *равноправными компонентами*. Это означает, что внешний вид компонента определяется конкретной платформой, а не Java. Компоненты библиотеки AWT используют ресурсы платформенно-ориентированного кода, и поэтому они называются *тяжеловесными*.

Использование платформенно-зависимых равноправных компонентов порождает ряд затруднений. Во-первых, в силу отличий в операционных системах компонент может выглядеть или даже вести себя по-разному на разных платформах. Такая потенциальная изменчивость шла вразрез с главным принципом Java: “написано однажды, работает везде”. Во-вторых, внешний вид каждого компонента был фиксированным, поскольку все зависело от конкретной платформы, а изменить его было очень трудно, если вообще возможно. И в-третьих, применение тяжеловесных компонентов влекло за собой новые неприятные ограничения. Например, тяжеловесный компонент всегда имеет прямоугольные очертания и является непрозрачным.

Вскоре после выпуска первоначальной версии Java стало очевидно, что ограничения, присущие библиотеке AWT, оказались настолько серьезными, что требовался более совершенный подход к построению ГПИ. Поэтому в 1997 году появилась библиотека Swing как часть набора библиотек классов Java Foundation Classes (JFC). Первоначально они были доступны в виде отдельной библиотеки, входившей в состав версии Java 1.1. А в версии Java 1.2 библиотека Swing, а также все компоненты, входившие в состав JFC, были полностью интегрированы в Java.

Построение библиотеки Swing на основе библиотеки AWT

Прежде чем продолжить дальше, нужно сделать следующее важное замечание: несмотря на то, что библиотека Swing снимает некоторые ограничения, присущие библиотеке AWT, она *не заменяет* ее. Напротив, библиотека Swing построена на основе библиотеки AWT. Именно поэтому библиотека AWT до сих пор является важной составной частью Java. Кроме того, в библиотеке Swing применяется тот же самый механизм обработки событий, что и в библиотеке AWT. Поэтому, прежде чем воспользоваться библиотекой Swing, следует усвоить хотя бы основные принципы работы библиотеки AWT. (Библиотеке AWT посвящены главы 25 и 26, а обработке событий — глава 24.)

Главные особенности библиотеки Swing

Как упоминалось выше, библиотека Swing была создана с целью преодолеть ограничения, присущие библиотеке AWT. Этому удалось достичь благодаря двум главным ее особенностям: легковесным компонентам и подключаемому стилю оформления. Совместно они предлагают изящное и простое в употреблении ре-

шение недостатков библиотеки AWT. Именно эти две особенности и определяют сущность библиотеки Swing. Каждая из них рассматривается ниже в отдельности.

Легковесные компоненты Swing

За редким исключением, компоненты библиотеки Swing являются *легковесными*. Это означает, что они написаны исключительно на Java и не преобразуются в равноправные компоненты для конкретной платформы. Следовательно, легковесные компоненты являются более эффективными и гибкими. Более того, легковесные компоненты не преобразуются в равноправные платформенно-ориентированные компоненты, поэтому внешний вид каждого компонента определяется библиотекой Swing, а не базовой операционной системой. Это означает, что каждый компонент будет действовать одинаково на всех платформах.

Подключаемый стиль оформления

В библиотеке Swing поддерживается *подключаемый стиль оформления* (PLAF). Каждый компонент библиотеки Swing воспроизводится кодом, написанным на Java, а не платформенно-ориентированными равноправными компонентами, поэтому стиль оформления компонента находится под управлением библиотеки Swing. Это означает, что стиль оформления можно отделить от логики компонента, что и делается в библиотеке Swing, что дает следующее преимущество: изменить способ воспроизведения компонента, не затрагивая остальные его свойства. Иными словами, новый стиль оформления любого компонента можно “подключить” без побочных эффектов в коде, использующем данный компонент. Более того, можно определить наборы стилей оформления, чтобы представить разные стили оформления ГПИ. Чтобы воспользоваться определенным стилем оформления, достаточно “подключить” его. Как только это будет сделано, все компоненты автоматически будут воспроизводиться в подключенном стиле оформления.

У подключаемых стилей оформления имеется ряд важных преимуществ. Например, можно определить такой стиль оформления, который будет одинаковым для всех платформ. С другой стороны, можно определить стиль оформления, характерный для отдельной платформы. Например, если заранее известно, что прикладная программа будет эксплуатироваться только в среде Windows, для ее ГПИ можно определить стиль оформления, характерный для Windows. Кроме того, можно разработать специальный стиль оформления. И наконец, стиль оформления можно динамически изменять во время работы прикладной программы.

В версии Java 8 предоставляются стили оформления *metal*, *Motif* и *Nimbus*, которые доступны всем пользователям библиотеки Swing. Стиль оформления *metal* (металлический) называется также *стилем оформления Java*. Он не зависит от платформы и доступен во всех исполняющих средах Java. Кроме того, этот стиль оформления выбирается по умолчанию. В среде Windows доступен также стиль оформления Windows. Здесь и далее используется стиль оформления *metal*, поскольку он не зависит от конкретной платформы.

Связь с архитектурой MVC

В общем, визуальный компонент определяется тремя отдельными составляющими.

- Внешний вид компонента при воспроизведении на экране.
- Взаимодействие с пользователем.
- Сведения о состоянии компонента.

Независимо от того, какая именно архитектура используется для реализации компонента, она должна неявно включать в себя эти три его составляющие. В течение многих лет свою исключительную эффективность доказала архитектура MVC — “модель–представление–контроллер”.

Успех архитектуры MVC объясняется тем, что каждая часть этой архитектуры соответствует отдельной составляющей компонента. Согласно терминологии MVC, *модель* соответствует сведениям о состоянии компонента. Так, для флажка модель содержит поле, которое показывает, установлен ли флажок. *Представление* определяет порядок отображения компонента на экране, включая любые составляющие представления, зависящие от текущего состояния модели. *Контроллер* определяет порядок реагирования компонента на действия пользователя. Так, если пользователь щелкает на флажке, контроллер реагирует на это действие, изменяя модель, чтобы отразить выбор пользователя (установку или сброс флажка). В итоге представление обновляется. Разделяя компонент на модель, представление и контроллер, можно изменять конкретную реализацию любой из этих составляющих архитектуры MVC, не затрагивая остальные. Например, различные реализации представлений могут воспроизводить один и тот же компонент разными способами, но это не будет оказывать никакого влияния на модель или контроллер.

Хотя архитектура MVC и положенные в ее основу принципы выглядят вполне разумно, высокая степень разделения представления и контроллера не дает никаких преимуществ компонентам библиотеки Swing. Поэтому в библиотеке Swing применяется видоизмененный вариант архитектуры MVC, где представление и контроллер объединены в один логический объект, называемый *представителем пользовательского интерфейса*. В связи с этим подход, применяемый в библиотеке Swing, называется архитектурой “модель–представитель”, или архитектурой “разделяемая модель”. Таким образом, в архитектуре компонентов библиотеки Swing не используется классическая реализация архитектуры MVC, несмотря на то, что первая основывается на последней.

Подключаемый стиль оформления стал возможным в библиотеке Swing благодаря архитектуре “модель–представитель”. Представление и контроллер отделены от модели, поэтому стиль оформления можно изменять, не оказывая влияния на то, как компонент применяется в программе. И наоборот, модель можно настроить, не оказывая влияния на то, как компонент отображается на экране или реагирует на действия пользователя.

Для поддержания архитектуры “модель–представитель” большинство компонентов библиотеки Swing содержит два объекта. Один из них представляет модель, а другой — представитель пользовательского интерфейса. Модели определяются в интерфейсах. Например, модель кнопки определяется в интерфейсе `ButtonModel`. А пред-

ставители пользовательского интерфейса являются классами, наследующими от класса `ComponentUI`. Например, представителем пользовательского интерфейса кнопки является класс `ButtonUI`. Как правило, прикладные программы не взаимодействуют непосредственно с представителями пользовательского интерфейса.

Компоненты и контейнеры

ГПИ, создаваемый средствами Swing, состоит из двух основных элементов: *компонентов* и *контейнеров*. Но это, по существу, концептуальное разделение, поскольку все контейнеры также являются компонентами. Отличие этих двух элементов заключается в их назначении: компонент является независимым визуальным элементом управления вроде кнопки или ползунка, а контейнер содержит группу компонентов. Таким образом, контейнер является особым типом компонента и предназначен для хранения других компонентов. Более того, компонент должен находиться в контейнере, чтобы его можно было отобразить. Так, во всех ГПИ, создаваемых средствами Swing, имеется как минимум один контейнер. А поскольку контейнеры являются компонентами, то один контейнер может содержать другие контейнеры. Благодаря этому в библиотеке Swing можно определить *иерархию вложенности*, на вершине которой должен находиться *контейнер верхнего уровня*. А теперь рассмотрим подробнее компоненты и контейнеры.

Компоненты

В общем, компоненты библиотеки Swing происходят от класса `JComponent`. (Исключением из этого правила являются четыре контейнера верхнего уровня, о которых речь пойдет в следующем разделе.) В классе `JComponent` предоставляются функциональные возможности, общие для всех компонентов. Так, в классе `JComponent` поддерживается подключаемый стиль оформления. Класс `JComponent` наследует классы `Container` и `Component` из библиотеки AWT. Следовательно, компонент библиотеки Swing построен на основе компонента библиотеки AWT и совместим с ним.

Все компоненты Swing представлены классами, определенными в пакете `javax.swing`. Ниже поименно перечислены классы компонентов Swing (включая компоненты, используемые в качестве контейнеров).

JApplet	JButton	JCheckBox	JCheckBoxMenuItem
JColorChooser	JComboBox	JComponent	JDesktopPane
JDialog	JEditorPane	JFileChooser	JFormattedTextField
JFrame	JInternalFrame	JLabel	JLayer
JLayeredPane	JList	JMenu	JMenuBar
JMenuItem	JOptionPane	JPanel	JPasswordField
JPopupMenu	JProgressBar	JRadioButton	JRadioButtonMenuItem
JRootPane	JScrollBar	JScrollPane	JSeparator
JSlider	JSpinner	JSplitPane	JTabbedPane
JTable	JTextArea	TextField	JTextPane
JToggleButton	JToolBar	JToolTip	JTree
JViewport	JWindow		

Обратите внимание на то, что все классы компонентов начинаются с буквы **J**. Например, класс для создания метки называется `JLabel`, класс для создания кнопки — `JButton`, а класс для создания ползунка — `JScrollBar`.

Контейнеры

В библиотеке Swing определены два типа контейнеров. К первому типу относятся контейнеры верхнего уровня, представленные классами `JFrame`, `JApplet`, `JWindow` и `JDialog`. Классы этих контейнеров не наследуют от класса `JComponent`, но они наследуют от классов `Component` и `Container` из библиотеки AWT. В отличие от остальных компонентов Swing, которые являются легковесными, компоненты верхнего уровня являются тяжеловесными. Поэтому в библиотеке Swing контейнеры являются особым случаем компонентов.

Судя по названию, контейнер верхнего уровня должен находиться на вершине иерархии контейнеров. Контейнер верхнего уровня не содержится ни в одном из других контейнеров. Более того, каждая иерархия вложенности должна начинаться с контейнера верхнего уровня. Таким контейнером в прикладных программах чаще всего является класс `JFrame`, а в апплетах — класс `JApplet`.

Ко второму типу контейнеров, поддерживаемых в библиотеке Swing, относятся легковесные контейнеры. Они наследуют от класса `JComponent`. Примером легковесного контейнера служит класс `JPanel`, который представляет контейнер общего назначения. Легковесные контейнеры нередко применяются для организации и управления группами связанных вместе компонентов, поскольку легковесный контейнер может находиться в другом контейнере. Следовательно, легковесные контейнеры вроде класса `JPanel` можно применять для создания подгрупп связанных вместе элементов управления, содержащихся во внешнем контейнере.

Панели контейнеров верхнего уровня

Каждый контейнер верхнего уровня определяет ряд *панелей*. На вершине иерархии панелей находится корневая панель в виде экземпляра класса `JRootPane`. Класс `JRootPane` представляет легковесный контейнер, предназначенный для управления остальными панелями. Он также помогает управлять дополнительной, хотя и не обязательной строкой меню. Панели, составляющие корневую панель, называются *прозрачной панелью*, *панелью содержимого* и *многослойной панелью* соответственно.

Прозрачная панель является панелью верхнего уровня. Она находится над всеми панелями и покрывает их полностью. По умолчанию эта панель представлена прозрачным экземпляром класса `JPanel`. Прозрачная панель позволяет управлять событиями от мыши, оказывающими влияние на весь контейнер в целом, а не на отдельный элемент управления, или, например, рисовать поверх любого другого компонента. Как правило, обращаться к прозрачной панели непосредственно не требуется, но если она все же понадобится, то ее нетрудно обнаружить там, где она обычно находится.

Многослойная панель представлена экземпляром класса `JLayeredPane`. Она позволяет задать определенную глубину размещения компонентов. Глубина определяет степень перекрытия компонентов. (В связи с этим многослойные панели

позволяют задавать упорядоченность компонентов по координате Z, хотя это требуется не всегда.) На многослойной панели находится панель содержимого и дополнительно, хотя и не обязательно, — строка меню.

Несмотря на то что прозрачная и многослойная панели являются неотъемлемыми частями контейнера верхнего уровня и служат для разных целей, большая часть их возможностей скрыта от пользователей. Прикладная программа чаще всего будет обращаться к панели содержимого, поскольку именно на ней обычно располагаются визуальные компоненты. Иными словами, когда компонент (например, кнопка) вводится в контейнер верхнего уровня, он оказывается на панели содержимого. По умолчанию панель содержимого представлена непрозрачным экземпляром класса `JPanel`.

Пакеты библиотеки Swing

Библиотека Swing — это довольно крупная подсистема, в которой задействовано большое количество пакетов. Ниже перечислены пакеты, определенные в библиотеке Swing на момент написания данной книги.

<code>javax.swing</code>	<code>javax.swing.plaf.basic</code>	<code>javax.swing.text</code>
<code>javax.swing.border</code>	<code>javax.swing.plaf.metal</code>	<code>javax.swing.text.html</code>
<code>javax.swing.colorchooser</code>	<code>javax.swing.plaf.multi</code>	<code>javax.swing.text.html.parser</code>
<code>javax.swing.event</code>	<code>javax.swing.plaf.nimbus</code>	<code>javax.swing.text.rtf</code>
<code>javax.swing.filechooser</code>	<code>javax.swing.plaf.synth</code>	<code>javax.swing.tree</code>
<code>javax.swing.plaf</code>	<code>javax.swing.table</code>	<code>javax.swing.undo</code>

Самым главным среди них является пакет `javax.swing`. Его следует импортировать в любую прикладную программу, пользующуюся библиотекой Swing. В этом пакете содержатся классы, реализующие базовые компоненты Swing, в том числе кнопки, метки и флажки.

Простое Swing-приложение

Swing-приложения отличаются от консольных программ и AWT-приложений, демонстрировавшихся ранее в данной книге. Например, в них используются компоненты и иерархии контейнеров не из библиотеки AWT. Кроме того, Swing-приложения предъявляют особые требования, связанные с многопоточной обработкой. Уяснить структуру Swing-приложения лучше всего на конкретном примере. Имеются две разновидности программ на Java, в которых обычно применяется библиотека Swing: настольные приложения и апплеты. В этом разделе будет рассмотрен пример создания Swing-приложения. А создание Swing-апплета обсуждается далее этой главе.

Несмотря на всю краткость рассматриваемого здесь примера программы, он наглядно демонстрирует один из способов разработки Swing-приложения, а также основные средства библиотеки Swing. В данном примере используются два ком-

понента Swing: `JFrame` и `JLabel`. Класс `JFrame` представляет контейнер верхнего уровня, который обычно применяется в Swing-приложениях, а класс `JLabel` — компонент Swing, создающий метку для отображения информации. Метка является самым простым компонентом Swing, поскольку это пассивный компонент. Это означает, что метка не реагирует на действия пользователя. Она служит лишь для отображения выводимых данных. В данном примере контейнер типа `JFrame` служит для хранения метки в виде экземпляра класса `JLabel`. Метка отображает короткое текстовое сообщение.

// Пример простого Swing-приложения

```
import javax.swing.*;

class SwingDemo {

    SwingDemo() {

        // создать новый контейнер типа JFrame
        JFrame jfrm = new JFrame("A Simple Swing Application");
        // Простое Swing-приложение

        // задать исходные размеры фрейма
        jfrm.setSize(275, 100);

        // завершить работу, если пользователь закрывает приложение
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // создать метку с текстом сообщения
        JLabel jlab = new JLabel("Swing means powerful GUIs.");
        // Swing — это мощные ГПИ

        // ввести метку на панели содержимого
        jfrm.add(jlab);

        // отобразить фрейм
        jfrm.setVisible(true);
    }

    public static void main(String args[]) {
        // создать фрейм в потоке диспетчеризации событий
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new SwingDemo();
            }
        });
    }
}
```

Swing-приложения компилируются и выполняются таким же образом, как и остальные приложения Java. Поэтому, чтобы скомпилировать данное Swing-приложение, нужно ввести в командной строке следующую команду:

```
javac SwingDemo.java
```

А для того чтобы запустить Swing-приложение на выполнение, нужно ввести в командной строке такую команду:

```
java SwingDemo
```

Когда Swing-приложение начнет выполняться, в нем будет создано окно, показанное на рис. 31.1.

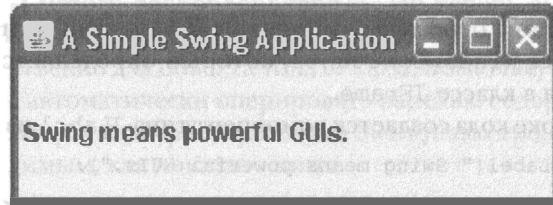


Рис. 31.1. Окно, создаваемое приложением SwingDemo

Пример приложения SwingDemo демонстрирует ряд основных понятий библиотеки Swing, поэтому рассмотрим этот пример построчно. Данное Swing-приложение начинается с импорта пакета `javax.swing`. Как упоминалось ранее, этот пакет содержит компоненты и модели, определяемые в библиотеке Swing. Так, в пакете `javax.swing` определяются классы, реализующие метки, кнопки, текстовые элементы управления и меню. Поэтому этот пакет обычно включается во все программы, пользующиеся библиотекой Swing.

Затем объявляются класс `SwingDemo` и его конструктор, в котором выполняется большинство действий данной программы. Сначала в нем создается экземпляр класса `JFrame`, как показано ниже.

```
JFrame jfrm = new JFrame("A Simple Swing Application");
```

В методе `setSize()`, наследуемом классом `JFrame` от класса `Component` из библиотеки AWT, задаются размеры окна в пикселях. Ниже приведена общая форма этого метода. В данном примере задается ширина окна 275 пикселей, а высота — 100 пикселей.

```
void setSize(int ширина, int высота)
```

Когда закрывается окно верхнего уровня (например, после того, как пользователь щелкнет на кнопке закрытия), по умолчанию окно удаляется с экрана, но работа приложения не прекращается. И хотя такое стандартное поведение иногда оказывается полезным, для большинства приложений оно не подходит. Чаше всего при закрытии окна верхнего уровня требуется завершить работу всего приложения. Это можно сделать двумя способами. Самый простой из них состоит в том, чтобы вызвать метод `setDefaultCloseOperation()`, что и делается в данном приложении следующим образом:

```
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

В результате вызова этого метода приложение полностью завершает свою работу при закрытии окна. Общая форма метода `setDefaultCloseOperation()` выглядит следующим образом:

```
void setDefaultCloseOperation(int что)
```

Значение константы, передаваемое в качестве параметра `что` определяет, что именно происходит при закрытии окна. Помимо значения константы `JFrame.EXIT_ON_CLOSE`, имеются также следующие значения:


```
DISPOSE_ON_CLOSE
HIDE_ON_CLOSE
DO_NOTHING_ON_CLOSE
```

Имена этих констант отражают выполняемые действия. Эти константы объявляются в интерфейсе `WindowConstants`, который определяется в пакете `javax.swing` и реализуется в классе `JFrame`.

В следующей строке кода создается компонент типа `JLabel` из библиотеки `Swing`:

```
JLabel jlab = new JLabel(" Swing means powerful GUIs.");
```

Класс `JLabel` определяет метку — самый простой в употреблении компонент, поскольку он не принимает вводимые пользователем данные, а только отображает информацию в виде текста, значка или того и другого. В данном примере создается метка, которая содержит только текст, передаваемый конструктору класса `JLabel`.

В следующей строке кода метка вводится на панели содержимого фрейма:

```
jfrm.add(jlab);
```

Как пояснялось ранее, у всех контейнеров верхнего уровня имеется панель содержимого, на которой размещаются отдельные компоненты. Следовательно, чтобы ввести компонент во фрейм, его нужно ввести на панели содержимого фрейма. Для этого достаточно вызвать метод `add()` по ссылке на экземпляр класса `JFrame` (в данном случае `jfrm`). Ниже приведена общая форма метода `add()`. Метод `add()` наследуется классом `JFrame` от класса `Container` из библиотеки `AWT`.

Component add(Component компонент)

По умолчанию на панели содержимого, связанной с компонентом типа `JFrame`, применяется граничная компоновка. В приведенной выше форме метода `add()` метка вводится по центру панели содержимого. Другие формы метода `add()` позволяют задать одну из граничных областей. Когда компонент вводится по центру, его размеры автоматически подгоняются таким образом, чтобы он разместился в центре.

Прежде чем продолжить дальше, следует заметить, что до версии `JDK 5` при вводе компонента на панели содержимого метод `add()` нельзя было вызывать непосредственно для экземпляра класса `JFrame`. Вместо этого метод `add()` приходилось вызывать для панели содержимого из объекта типа `JFrame`. Панель содержимого можно было получить в результате вызова метода `getContentPane()` для экземпляра класса `JFrame`. Ниже приведена общая форма метода `getContentPane()`.

Container getContentPane()

Класс `Container` получает ссылку на окно содержимого. И по этой ссылке делается вызов метода `add()`, чтобы ввести компонент на панели содержимого. Следовательно, чтобы ввести метку `jlab` во фрейм `jfrm`, раньше приходилось пользоваться следующим оператором:

```
jfrm.getContentPane().add(jlab); // старый стиль
```

В этом операторе сначала вызывается метод `getContentPane()`, получающий ссылку на панель содержимого, а затем метод `add()`, вводящий указанный компонент (в данном случае метку) в контейнер, связанный с этой панелью. Эта же процедура потребовалась бы и для вызова метода `remove()`, чтобы удалить компонент,

или для вызова метода `setLayout()`, чтобы задать диспетчер компоновки для окна содержимого. Именно поэтому в коде, написанном на Java до версии 5.0, нередко встречаются вызовы метода `getContentPane()`. Но теперь вызывать этот метод больше не нужно. Вместо этого достаточно вызвать методы `add()`, `remove()` и `setLayout()` непосредственно для объекта типа `JFrame`, поскольку они были специально изменены, чтобы автоматически оперировать панелью содержимого.

Последний оператор в конструкторе класса `SwingDemo` требуется для того, чтобы сделать окно видимым, как показано ниже.

```
jfrm.setVisible(true);
```

Метод `setVisible()` наследуется от класса `Component` из библиотеки AWT. Если его аргумент принимает логическое значение `true`, то окно отобразится, а иначе оно будет скрыто. По умолчанию контейнер типа `JFrame` невидим, поэтому нужно вызвать метод `setVisible(true)`, чтобы показать его.

В методе `main()` создается объект типа `SwingDemo`, чтобы отобразить окно и метку. Обратите внимание на то, что конструктор класса `SwingDemo` вызывается в следующих трех строках кода:

```
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        new SwingDemo();
    }
});
```

Выполнение этой последовательности кода приводит к созданию объекта типа `SwingDemo` в *потоке диспетчеризации событий*, а не в главном потоке исполнения данного приложения, потому что Swing-приложения обычно выполняются под управлением событий. А событие происходит, например, когда пользователь взаимодействует с компонентом. Событие передается приложению при вызове обработчика событий, определенного в этом приложении. Но обработчик выполняется в потоке диспетчеризации событий, предоставляемом библиотекой Swing, а не в главном потоке исполнения данного приложения. Таким образом, обработчики событий вызываются в потоке исполнения, который не был создан в приложении, хотя они и определены в нем.

Чтобы избежать осложнений, в том числе вероятной взаимной блокировки, все компоненты ГПИ из библиотеки Swing следует создавать и обновлять из потока диспетчеризации событий, а не из главного потока исполнения данного приложения. Но метод `main()` выполняется в главном потоке исполнения. Следовательно, метод `main()` не может напрямую наследовать объект типа `SwingDemo`. Вместо этого нужно создать объект типа `Runnable`, который выполняется в потоке диспетчеризации событий, и с помощью этого объекта построить ГПИ.

Чтобы построить ГПИ в потоке диспетчеризации событий, следует вызвать один из следующих двух методов, определенных в классе `SwingUtilities`: `invokeLater()` и `invokeAndWait()`. Ниже приведены общие формы этих методов.

```
static void invokeLater(Runnable объект)
static void invokeAndWait(Runnable объект)
        throws InterruptedException, InvocationTargetException
```

Здесь параметр *объект* обозначает объект типа `Runnable`, метод `run()` которого должен вызываться в потоке диспетчеризации событий. Единственное отли-

чие этих двух методов заключается в том, что метод `invokeLater()` выполняет возврат немедленно, а метод `invokeAndWait()` ожидает возврата из метода `obj.run()`. Этими методами можно пользоваться для вызова метода, строящего ГПИ конкретного Swing-приложения, или вызывать их всякий раз, когда требуется изменить состояние ГПИ из кода, не выполняющегося в потоке диспетчеризации событий. Для этих целей обычно вызывается метод `invokeLater()`, как демонстрируется в рассматриваемом здесь примере. А при построении исходного ГПИ для апплета потребуются метод `invokeAndWait()`.

Обработка событий

В предыдущем примере была продемонстрирована основная форма Swing-приложения, но ей недостает одной важной части: обработки событий. Компонент типа `JLabel` не принимает данные, вводимые пользователем, и не генерирует события, и поэтому обработка событий в данном примере не требовалась. Но остальные компоненты библиотеки Swing *реагируют* на вводимые пользователем данные, а следовательно, требуется каким-то образом обработать события, наступающие в результате подобных взаимодействий. Событие происходит, когда, например, срабатывает таймер. Так или иначе, обработка событий занимает большую часть любого Swing-приложения.

Механизм обработки событий, применяемый в библиотеке Swing, ничем не отличается от аналогичного механизма из библиотеки AWT, называемого *моделью делегирования событий*, как пояснялось в главе 24. Как правило, в библиотеке Swing используются те же самые события, что и в библиотеке AWT, и эти события определены в пакете `java.awt.event`. А события, характерные только для библиотеки Swing, определены в пакете `javax.swing.event`.

Несмотря на то что события обрабатываются в библиотеке Swing таким же образом, как и в библиотеке AWT, этот процесс лучше рассмотреть на простом примере. В приведенной ниже программе обрабатывается событие, генерируемое после щелчка на экранной кнопке, определяемой соответствующим компонентом Swing. Результат выполнения данной программы приведен на рис. 31.2.

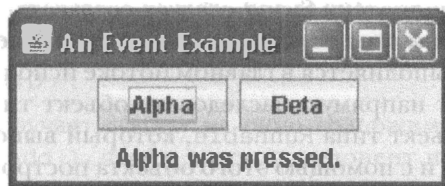


Рис. 31.2. Результат выполнения программы EventDemo

// Обработка события в Swing-приложении

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class EventDemo {
```

```
JLabel jlab;

EventDemo() {

    // создать новый контейнер типа JFrame
    JFrame jfrm = new JFrame("An Event Example");
    // Пример обработки событий

    // определить диспетчер потоковой компоновки типа FlowLayout
    jfrm.setLayout(new FlowLayout());

    // установить исходные размеры фрейма
    jfrm.setSize(220, 90);

    // завершить работу приложения, если пользователь
    // закрывает его окно
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // создать две кнопки
    JButton jbtnAlpha = new JButton("Alpha");
    JButton jbtnBeta = new JButton("Beta");

    // ввести приемник действий от кнопки Alpha
    jbtnAlpha.addActionListener(new ActionListener() {

        public void actionPerformed(ActionEvent ae) {
            jlab.setText("Alpha was pressed.");
            // Нажата кнопка Alpha
        }

    });

    // ввести приемник действий от кнопки Beta
    jbtnBeta.addActionListener(new ActionListener() {

        public void actionPerformed(ActionEvent ae) {
            jlab.setText("Beta was pressed.");
            // Нажата кнопка Beta
        }

    });

    // ввести кнопки на панели содержимого
    jfrm.add(jbtnAlpha);
    jfrm.add(jbtnBeta);

    // создать текстовую метку
    jlab = new JLabel("Press a button.");
    // Метка "Нажмите кнопку."

    // ввести метку на панели содержимого
    jfrm.add(jlab);

    // отобразить фрейм
    jfrm.setVisible(true);
}

public static void main(String args[]) {
    // создать фрейм в потоке диспетчеризации событий
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new EventDemo();
        }
    });
}
}
```

Прежде всего обратите внимание на то, что в данном примере программы теперь импортируются пакеты `java.awt` и `java.awt.event`. Пакет `java.awt` требуется потому, что в нем содержится класс `FlowLayout`, поддерживающий стандартный диспетчер поточной компоновки, который применяется для размещения компонентов во фрейме. (Описание диспетчеров компоновки см. в главе 26.) А пакет `java.awt.event` требуется потому, что в нем определяются интерфейс `ActionListener` и класс `ActionEvent`.

Сначала в конструкторе класса `EventDemo` создается контейнер `jfrm` типа `JFrame`, а затем устанавливается диспетчер компоновки типа `FlowLayout` для панели содержимого контейнера `jfrm`. Напомним, что по умолчанию для размещения компонентов на панели содержимого применяется диспетчер компоновки типа `BorderLayout`, но для данного примера больше подходит диспетчер компоновки типа `FlowLayout`. Обратите внимание на то, что диспетчер компоновки типа `FlowLayout` назначается с помощью следующего оператора:

```
jfrm.setLayout(new FlowLayout());
```

Как упоминалось выше, раньше приходилось явным образом вызывать метод `getContentPane()`, чтобы задать диспетчер компоновки для размещения компонентов на панели содержимого. Но начиная с версии JDK 5 этого больше не нужно делать.

После определения размеров и стандартной операции, выполняемой при закрытии окна, в конструкторе класса `EventDemo` создаются две экранные кнопки, как показано ниже.

```
JButton jbtnAlpha = new JButton("Alpha");
JButton jbtnBeta  = new JButton("Beta");
```

Первая кнопка будет содержать надпись "Alpha", а вторая — надпись "Beta". Экранные кнопки из библиотеки Swing являются экземплярами класса `JButton`. В классе `JButton` предоставляется несколько конструкторов. Здесь используется приведенный ниже конструктор, где параметр *сообщение* обозначает символьную строку, которая будет отображаться в виде надписи на экранной кнопке.

```
JButton(String сообщение)
```

После щелчка на экранной кнопке наступает событие типа `ActionEvent`. Поэтому в классе `JButton` предоставляется метод `addActionListener()`, который служит для ввода приемника подобных событий. (В классе `JButton` предоставляется также метод `removeActionListener()` для удаления приемника событий, но в рассматриваемом здесь примере программы он не применяется.) Как пояснялось в главе 24, в интерфейсе `ActionListener` определяется единственный метод `actionPerformed()`. Ради удобства ниже еще раз приводится его общая форма.

```
void actionPerformed(ActionEvent событие_действия)
```

Этот метод вызывается после щелчка на экранной кнопке. Иными словами, это обработчик, который вызывается, когда наступает событие нажатия экранной кнопки.

Далее вводится приемник событий от двух экранных кнопок, как показано ниже. В данном случае используются анонимные внутренние классы, чтобы предоставить обработчики событий от двух экранных кнопок. Всякий раз, когда нажи-

мается экранная кнопка, символьная строка, отображаемая на месте метки `jlab`, изменяется в зависимости от того, какая кнопка была нажата.

```
// ввести приемник событий действия от кнопки Alpha
jbtnAlpha.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        jlab.setText("Alpha was pressed.");
        // Нажата кнопка Alpha
    }
});

// ввести приемник событий действия от кнопки Beta
jbtnBeta.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        jlab.setText("Beta was pressed.");
        // Нажата кнопка Beta
    }
});
```

Начиная с версии JDK 8 для реализации обработчиков событий можно также воспользоваться лямбда-выражениями. Например, обработчик событий от кнопки **Alpha** можно было бы написать следующим образом:

```
jbtnAlpha.addActionListener( (ae) -> jlab.setText("Alpha was pressed."));
```

Как видите, этот код более краткий. В интересах тех читателей, которые пользуются Java до версии JDK 8, в последующих примерах лямбда-выражения не применяются. Но при написании нового кода следует непременно найти возможности воспользоваться ими.

Затем кнопки вводятся на панели содержимого следующим образом:

```
jfrm.add(jbtnAlpha);
jfrm.add(jbtnBeta);
```

И наконец, на панели содержимого вводится метка `jlab`, и окно приложения становится видимым. Если после запуска данного Swing-приложения щелкнуть на любой из двух экранных кнопок, то на месте метки отобразится сообщение, извещающее, какая именно кнопка была нажата.

И еще одно, последнее замечание: не следует забывать, что все обработчики событий вроде метода `actionPerformed()` вызываются в потоке диспетчеризации событий. Следовательно, возврат из обработчика событий должен быть произведен быстро, чтобы не замедлить выполнение Swing-приложения. Если же при наступлении некоторого события в приложении требуется выполнить операции, отнимающие много времени, то для этой цели следует организовать отдельный поток исполнения.

Создание Swing-аплета

Еще одной разновидностью прикладной программы, где обычно применяется библиотека Swing, является аplet. Аплеты, разработанные на основе библиотеки Swing, подобны апплетам, построенным на основе библиотеки AWT, но у них есть одно существенное отличие: Swing-аplet расширяет класс `JApplet`, а не класс `Applet`. Таким образом, класс `JApplet` включает в себя все функциональные

возможности класса `Applet`, дополняя их поддержкой библиотеки `Swing`. Класс `JApplet` служит контейнером `Swing` верхнего уровня, а это означает, что он *не* наследует от класса `JComponent`, но в то же время включает в себя различные описанные ранее панели. Следовательно, все компоненты вводятся на панели содержимого контейнера типа `JApplet` таким же образом, как и компоненты на панели содержимого контейнера типа `JFrame`.

В `Swing`-апплетах применяются те же самые методы обеспечения жизненного цикла апплета, которые были описаны в главе 23, а именно: `init()`, `start()`, `stop()` и `destroy()`. Естественно, переопределить следует только те методы, которые потребуются в апплете. Рисование графики в библиотеке `Swing` осуществляется иначе, чем в библиотеке `AWT`, и поэтому метод `paint()` обычно не переопределяется в `Swing`-апплете. (О рисовании средствами `Swing` речь пойдет далее в этой главе.)

Следует также иметь в виду, что все взаимодействия с компонентами `Swing` должны происходить в потоке диспетчеризации событий, как пояснялось в предыдущем разделе. Это относится ко всем `Swing`-приложениям.

Ниже представлен пример `Swing`-апплета. Он обладает теми же функциональными возможностями, что и `Swing`-приложение из предыдущего примера, но только является апплетом. На рис. 31.3 показан результат выполнения `Swing`-апплета из данного примера в окне утилиты **appletviewer**.

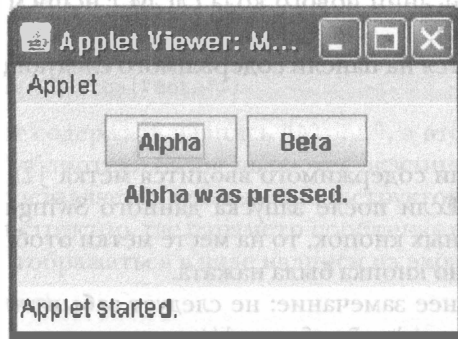


Рис. 31.3. Результат выполнения примера `Swing`-апплета

// Пример простого `Swing`-апплета

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
```

```
/*
```

Этот код HTML можно использовать для запуска апплета:

```
<applet code="MySwingApplet" width=220 height=90>
</applet>
```

```
*/
```

```
public class MySwingApplet extends JApplet {
    JButton jbtnAlpha;
    JButton jbtnBeta;
```

```

JLabel jlab;

// инициализировать апплет
public void init() {
    try {
        SwingUtilities.invokeAndWait(new Runnable () {
            public void run() {
                makeGUI(); // инициализировать ГПИ
            }
        });
    } catch (Exception exc) {
        System.out.println("Can't create because of "+ exc);
        // Нельзя создать из-за исключения указанного типа
    }
}

// В этом апплете не нужно переопределять методы
// start(), stop() или destroy()

// настроить и инициализировать ГПИ
private void makeGUI() {

    // установить для апплета диспетчер потоочной компоновки
    setLayout(new FlowLayout());

    // создать две кнопки
    jbtnAlpha = new JButton("Alpha");
    jbtnBeta = new JButton("Beta");

    // ввести приемник действия от кнопки Alpha
    jbtnAlpha.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent le) {
            jlab.setText("Alpha was pressed.");
            // Нажата кнопка Alpha
        }
    });

    // ввести приемник действия от кнопки Beta
    jbtnBeta.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent le) {
            jlab.setText("Beta was pressed.");
            // Нажата кнопка Beta
        }
    });

    // ввести кнопки на панели содержимого
    add(jbtnAlpha);
    add(jbtnBeta);

    // создать текстовую метку
    jlab = new JLabel("Press a button.");
    // Метка "Нажмите кнопку."

    // ввести метки на панели содержимого
    add(jlab);
}
}

```

В отношении апплетов следует сделать два важных замечания. Во-первых, класс `MySwingApplet` расширяет класс `JApplet`. Как пояснялось ранее, все апплеты, построенные на основе библиотеки `Swing`, расширяют класс `JApplet`, а не класс `Applet`. И во-вторых, метод `init()` инициализирует компоненты `Swing` в пото-

ке диспетчеризации событий, устанавливая вызов метода `makeGUI()`. Обратите внимание на то, что для этой цели служит метод `invokeAndWait()`, а не `invokeLater()`. Метод `invokeAndWait()` следует применять в апплетах потому, что возврат из метода `init()` не должен происходить до тех пор, пока не завершится весь процесс инициализации. По существу, метод `start()` нельзя вызывать до завершения инициализации, а это означает, что прежде нужно полностью построить ГПИ.

В методе `makeGUI()` создаются две кнопки и метка, а также вводятся приемники действий от этих кнопок. И наконец, компоненты вводятся на панели содержимого. Несмотря на всю простоту данного примера, демонстрируемый в нем принцип следует применять при построении любого ГПИ для Swing-апплета.

Рисование средствами Swing

Какими бы эффективными ни были компоненты Swing, построение ГПИ не ограничивается только ими, поскольку библиотека Swing позволяет также выводить информацию непосредственно в область отображения фрейма, панели или одного из компонентов Swing вроде метки типа `JLabel`. Как правило, рисование не выполняется непосредственно на поверхности компонента Swing, тем не менее это можно делать в тех приложениях, где требуется нечто подобное. Чтобы вывести данные прямо на поверхность компонента Swing, следует вызвать один или несколько методов рисования, определенных в библиотеке AWT, например `drawLine()` или `drawRect()`. Таким образом, большинство приемов и методов, описанных в главе 25, распространяются и на библиотеку Swing. Но в подходе к рисованию средствами Swing имеется ряд очень важных отличий, о которых и пойдет речь в этом разделе.

Основы рисования

Подход к рисованию средствами Swing основывается на исходном механизме из библиотеки AWT, но библиотека Swing позволяет более точно управлять этим процессом. Прежде чем приступить к обсуждению особенностей рисования средствами Swing, целесообразно напомнить особенности механизма рисования в библиотеке AWT.

В классе `Component` из библиотеки AWT предоставляется метод `paint()`, предназначенный для рисования выводимых данных прямо на поверхности компонента. Как правило, метод `paint()` не вызывается для этого из прикладной программы. (В действительности вызывать этот метод из прикладной программы приходится в очень редких случаях.) Вместо этого метод `paint()` вызывается исполняющей средой в процессе воспроизведения компонента. Такая ситуация может возникнуть по нескольким причинам. Например, окно, в котором отображается компонент, может быть перекрыто другим окном, а затем появиться снова на экране, или же оно может быть свернуто, а затем восстановлено. Метод `paint()` вызывается и в тех случаях, когда программа начинает выполняться. Если разрабатывается приложение на основе библиотеки AWT, метод `paint()` переопределяется в нем всякий раз, когда данные требуется вывести прямо на поверхности компонента.

Класс `JComponent` наследует от класса `Component`, и поэтому метод `paint()` доступен всем легковесным компонентам Swing. Но переопределять его с целью выводить информацию непосредственно на поверхности компонента *не* придется. Дело в том, что при рисовании средствами библиотеки Swing применяется более изощренный подход с помощью трех разных методов: `paintComponent()`, `paintBorder()` и `paintChildren()`. Эти методы рисуют указанную часть компонента, разделяя процесс рисования на три разных логических действия. В легковесном компоненте исходный метод `paint()` из библиотеки AWT просто вызывает эти методы в упомянутом выше порядке.

Чтобы нарисовать что-нибудь на поверхности компонента Swing, сначала придется создать подкласс компонента, а затем переопределить его метод `paintComponent()`. Этот метод отвечает за прорисовку внутренней области компонента. А два других метода рисования, как правило, переопределять не требуется. Переопределяя метод `paintComponent()`, нужно сначала вызвать метод `super.paintComponent()`, чтобы задействовать часть процесса рисования из суперкласса. (Этого делать не нужно лишь в том случае, если управление отображением компонента осуществляется вручную.) После этого можно вывести отображаемые данные. Ниже приведена общая форма метода `paintComponent()`, где параметр *g* обозначает графическое содержимое выводимых данных.

```
protected void paintComponent(Graphics g)
```

Чтобы нарисовать на поверхности компонента под управлением программы, следует вызвать метод `repaint()`. Этот метод действует в библиотеке Swing таким же образом, как и в библиотеке AWT. Метод `repaint()` определен в классе `Component`. Вызов этого метода приводит к тому, что исполняющая система вызовет метод `paint()`, как только для этого представится возможность. А поскольку процесс рисования отнимает немало времени, то данный механизм позволяет исполняющей системе мгновенно задерживать рисование до тех пор, пока, например, не завершится выполнение другой задачи, имеющей более высокий приоритет. Вызов метода `paint()` в Swing, безусловно, приводит к вызову метода `paintComponent()`. Следовательно, данные, выводимые на поверхность компонента, должны сохраняться прикладной программой до тех пор, пока не будет вызван метод `paintComponent()`. А рисование сохраняемых данных выполняется в переопределяемом методе `paintComponent()`.

Вычисление области рисования

Во время рисования на поверхности компонента нужно аккуратно ограничить область рисования выводимых данных в пределах компонента. Несмотря на то что любые выводимые данные, выходящие за границы компонента, автоматически отсекаются средствами Swing, вполне возможно, что какой-нибудь рисуемый участок окажется прямо на границе, а при ее перерисовке он может быть стерт. Во избежание этого следует вычислить *область рисования* в пределах компонента. Эта область определяется следующим образом: из текущих размеров компонента вычитается пространство, занятое его границами. Следовательно, прежде чем рисовать на поверхности компонента, следует сначала выяснить ширину границы, а затем откорректировать соответственно рисование.

Чтобы выяснить ширину границы компонента, следует вызвать метод `getInsets()`. Ниже приведена общая форма этого метода.

```
Insets getInsets()
```

Этот метод определяется в классе `Container` и переопределяется в классе `JComponent`. Он возвращает объект типа `Insets`, содержащий размеры границ по сторонам компонента. Значения этих размеров можно получить из следующих полей:

```
int top;
int bottom;
int left;
int right;
```

Впоследствии эти значения применяются для вычисления области рисования с учетом ширины и высоты компонента. Ширину и высоту компонента можно выяснить, вызвав методы `getWidth()` и `getHeight()` соответственно. Ниже приведены общие формы этих методов.

```
int getWidth()
int getHeight()
```

Вычитая соответствующие значения размеров границ из ширины и высоты компонента, можно получить ширину и высоту области, в которой предполагается рисование.

Пример рисования

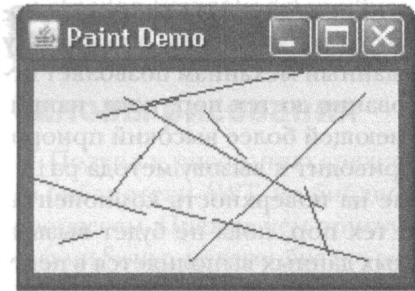


Рис. 31.4. Примерный результат выполнения программы `PaintPanel`

А теперь рассмотрим пример программы, где демонстрируются упомянутые выше методы рисования. В этой программе создается класс `PaintPanel`, расширяющий класс `JPanel`. Объект данного класса служит для отображения линий, конечные точки которых формируются случайным образом. Результат выполнения данной программы приведен на рис. 31.4.

```
// Рисовать линии на панели

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

// Этот класс расширяет класс JPanel. В нем переопределяется
// метод paintComponent(), чтобы произвольно рисовать линии на панели
class PaintPanel extends JPanel {
    Insets ins; // служит для хранения размеров границ панели

    Random rand; // служит для генерирования случайных чисел

    // создать панель
```

```
PaintPanel() {  
    // разместить рамку вокруг панели, определив ее границы  
    setBorder(  
        BorderFactory.createLineBorder(Color.RED, 5));  
  
    rand = new Random();  
}  
  
// переопределить метод paintComponent()  
protected void paintComponent(Graphics g) {  
    // вызывать всегда первым метод из суперкласса  
    super.paintComponent(g);  
    int x, y, x2, y2;  
  
    // получить высоту и ширину компонента  
    int height = getHeight();  
    int width = getWidth();  
  
    // получить размеры границ панели  
    ins = getInsets();  
  
    // нарисовать десять линий, конечные точки которых  
    // формируются произвольно  
    for(int i=0; i < 10; i++) {  
        // получить произвольные координаты, определяющие  
        // конечные точки каждой линии  
        x = rand.nextInt(width-ins.left);  
        y = rand.nextInt(height-ins.bottom);  
        x2 = rand.nextInt(width-ins.left);  
        y2 = rand.nextInt(height-ins.bottom);  
        // нарисовать линию  
        g.drawLine(x, y, x2, y2);  
    }  
}  
}  
  
// продемонстрировать рисование непосредственно на панели  
class PaintDemo {  
    JLabel jlab;  
    PaintPanel pp;  
    PaintDemo() {  
  
        // создать новый контейнер типа JFrame  
        JFrame jfrm = new JFrame("Paint Demo");  
  
        // задать исходные размеры фрейма  
        jfrm.setSize(200, 150);  
  
        // завершить приложение, если пользователь  
        // закроет его окно  
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
  
        // создать панель для рисования  
        pp = new PaintPanel();  
  
        // Ввести эту панель на панели содержимого. В данном  
        // случае применяется граничная компоновка, поэтому  
        // размеры панели будут автоматически подгоняться таким  
        // образом, чтобы она заняла центральную область  
        jfrm.add(pp);  
  
        // отобразить фрейм  
        jfrm.setVisible(true);  
    }  
}
```

```

    }

    public static void main(String args[]) {
        // создать фрейм в потоке диспетчеризации событий
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new PaintDemo();
            }
        });
    }
}

```

А теперь рассмотрим данную программу подробнее. Класс `PaintPanel` расширяет класс `JPanel`, представляющий один из легковесных контейнеров Swing, т.е. компонент, который можно вводить на панели содержимого `JFrame`. Для рисования в классе `PaintPanel` переопределяется метод `paintComponent()`. Это позволяет рисовать прямо на поверхности компонента средствами класса `PaintPanel`. Размеры панели не определены, поскольку в данной программе по умолчанию используется граничная компоновка, а панель вводится по центру. В итоге панель получает такие размеры, которые позволяют разместить ее по центру. При изменении размеров окна соответственно подгоняются размеры панели.

Обратите внимание на то, что в конструкторе класса `PaintPanel` определяется также красная граница толщиной 5 пикселей. Для этого вызывается метод `setBorder()`, общая форма которого показана ниже.

```
void setBorder(Border граница)
```

Интерфейс `Border` из библиотеки Swing инкапсулирует границу, которую можно получить, вызвав один из методов, определенных в классе `BorderFactory`. В данной программе применяется один из таких методов, называемый `createLineBorder()`. Он создает простую линейную границу, как показано ниже, где параметр *цвет* обозначает цвет рамки, а параметр *ширина* — ее ширину в пикселях.

```
static Border createLineBorder(Color цвет, int ширина)
```

В переопределяемом варианте метода `paintComponent()` следует обратить внимание на то, что сначала в нем вызывается метод `super.paintComponent()`. Как пояснялось ранее, это требуется для обеспечения надлежащего рисования. Затем вычисляется ширина и высота панели, а также размеры границ. Эти значения используются для того, чтобы рисуемые линии не выходили за пределы области рисования на панели. Область рисования определяется вычитанием размеров границ из общей ширины и высоты компонента. Вычисления организованы таким образом, чтобы учитывать разные размеры области рисования и границ. Чтобы убедиться в этом, попробуйте изменить размеры окна. Линии по-прежнему будут рисоваться, не выходя за границы панели.

В классе `PaintDemo` сначала создается панель рисования типа `PaintPanel`, а затем она вводится на панели содержимого. При первом отображении вызывается переопределяемый метод `paintComponent()` и рисуются линии. Всякий раз, когда изменяются размеры окна или же оно сворачивается и восстанавливается, рисуется новый ряд линий. Но в любом случае линии не будут выходить за пределы заданной области рисования.

Исследование библиотеки Swing

В предыдущей главе были рассмотрены некоторые основные принципы построения библиотеки Swing и показана общая форма приложений и апплетов, создаваемых на основе библиотеки Swing. В этой главе продолжается исследование библиотеки Swing кратким обзором ее компонентов, в том числе кнопок, флажков, деревьев и таблиц. Компоненты Swing обладают богатыми функциональными возможностями и допускают специальную настройку в широких пределах. К сожалению, в рамках данной книги невозможно описать все особенности и свойства компонентов Swing, поэтому рассмотрим лишь некоторые из них.

Ниже перечислены классы компонентов Swing, рассматриваемых в настоящей главе. Все эти компоненты являются легковесными. Это означает, что все они происходят от класса `JComponent`.

JButton	JCheckBox	JComboBox	JLabel
JList	JRadioButton	JScrollPane	JTabbedPane
JTable	TextField	JToggleButton	JTree

В этой главе рассматривается также класс `ButtonGroup`, инкапсулирующий взаимоисключающий ряд кнопок Swing, а также класс `ImageIcon`, инкапсулирующий графическое изображение. Каждый из этих классов определяется в библиотеке Swing и входит в пакет `javax.swing`.

Следует также иметь в виду, что в этой главе применение компонентов Swing демонстрируется на примерах апплетов, поскольку код апплета более компактный, чем код настольного приложения. Но и на те и на другие в одинаковой степени распространяются рассматриваемые здесь методики построения ГПИ.

Классы `JLabel` и `ImageIcon`

Класс `JLabel` представляет метку – самый простой в употреблении компонент Swing. Этот компонент уже рассматривался в предыдущей главе, и вам должно быть уже известно в общих чертах, как с его помощью создается метка. А в этой главе компонент типа `JLabel` рассматривается более подробно. С помощью компонента типа `JLabel` можно отображать текст и/или значок. Этот компонент является пассивным в том отношении, что он не реагирует на данные, вводимые

пользователем. В классе JLabel определяется несколько конструкторов. Ниже приведены три из них.

```
JLabel (Icon значок)
JLabel (String строка)
JLabel (String строка, Icon значок, int выравнивание)
```

Здесь параметры *строка* и *значок* обозначают соответственно текст и значок, которые будут использоваться в качестве метки, а параметр *выравнивание* — вид выравнивания текста и/или значка по горизонтали в пределах метки. Этот параметр должен принимать одно из значений следующих констант: LEFT, RIGHT, CENTER, LEADING или TRAILING. Наряду с рядом других констант, используемых в классах из библиотеки Swing, эти константы определяются в интерфейсе SwingConstants.

Обратите внимание на то, что значки определяются с помощью объектов типа Icon, относящегося к интерфейсу, определяемому в библиотеке Swing. Получить значок проще всего средствами класса ImageIcon, реализующего интерфейс Icon и инкапсулирующего изображение. Следовательно, объект типа ImageIcon можно передать в качестве параметра типа Icon конструктору класса JLabel. Предоставить изображение можно несколькими способами, включая чтение изображения из файла или его загрузку по указанному URL. Ниже приведен конструктор класса ImageIcon, используемый в примере из этого раздела. Этот конструктор получает изображение из файла, обозначаемого параметром *имя_файла*.

```
ImageIcon (String имя_файла)
```

Значок и текст, связанные с меткой, можно получить с помощью следующих методов:

```
Icon getIcon ()
String getText ()
```

Значок и текст, связанные с меткой, можно установить, вызывая приведенные ниже методы.

```
void setIcon (Icon значок)
void setText (String строка)
```

Здесь параметры *значок* и *строка* обозначают указываемый значок и текст соответственно. Таким образом, с помощью метода setText () можно изменить текст метки во время выполнения программы.

В приведенном ниже примере апплета демонстрируется порядок создания и отображения метки, состоящей из значка и символьной строки. Сначала в данном апплете создается объекта типа ImageIcon для отображения песочных часов из файла изображения hourglass.gif. Этот объект указывается в качестве второго параметра при вызове конструктора класса JLabel. А первый и последний параметры этого конструктора представляют текст метки и его выравнивание. И наконец, созданная метка вводится на панели содержимого. На рис. 32.1 показано, каким образом метка в виде значка отображается в окне апплета из данного примера.

```
// Продемонстрировать применение компонентов типа JLabel и ImageIcon
import java.awt.*;
import javax.swing.*;
```

```

/*
<applet code="JLabelDemo" width=250 height=200>
</applet>
*/
public class JLabelDemo extends JApplet {

    public void init() {
        try {
            SwingUtilities.invokeLater(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
            // Нельзя создать по указанной причине
        }
    }

    private void makeGUI() {

        // создать значок
        ImageIcon ii = new ImageIcon("hourglass.png");

        // создать метку
        JLabel jl = new JLabel("Hourglass", ii, JLabel.CENTER);

        // ввести метку на панели содержимого
        add(jl);
    }
}

```

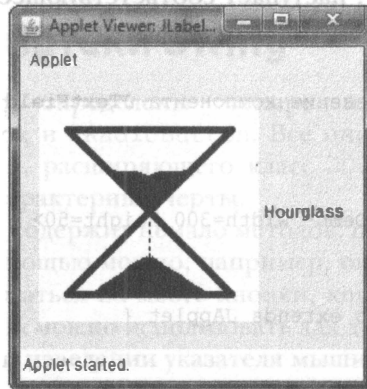


Рис. 32.1. Отображение метки
в окне апплета JLabelDemo

Класс JTextField

Класс `JTextField` представляет простейший текстовый компонент из библиотеки Swing. Это, пожалуй, наиболее употребительный текстовый компонент. Этот

компонент позволяет отредактировать одну строку текста. Класс `JTextField` является производным от класса `JTextComponent`, наделяющим функциональными возможностями текстовые компоненты Swing. В качестве своей модели класс `JTextField` использует интерфейс `Document`. Ниже приведены три конструктора класса `JTextField`.

```
JTextField(int столбцы)
JTextField(String строка, int цвета)
JTextField(String строка)
```

Здесь параметр *строка* обозначает первоначально предоставляемую символьную строку, а параметр *столбцы* — количество столбцов в текстовом поле. Если параметр *строка* не задан, то текстовое поле оказывается исходно пустым. А если не задан параметр *столбцы*, то размеры текстового поля выбираются таким образом, чтобы оно могло уместиться в указанной символьной строке.

Компонент типа `JTextField` генерирует события в ответ на действия пользователя. Например, событие типа `ActionEvent` наступает при нажатии пользователем клавиши `<Enter>`, а событие типа `CaretEvent` — при каждом изменении позиции каретки (т.е. курсора). (Событие типа `CaretEvent` определяется в пакете `javax.swing.event`.) Возможны и другие события. Как правило, эти события не нужно обрабатывать в прикладной программе. Вместо этого достаточно получить символьную строку, находящуюся в данный момент в текстовом поле. Для ее получения следует вызвать метод `getText()`.

В приведенном ниже примере апплета демонстрируется применение компонента типа `JTextField`. В данном апплете сначала создается компонент типа `JTextField`, который затем вводится на панели содержимого. Когда пользователь нажимает клавишу `<Enter>`, наступает соответствующее событие действия. В результате обработки этого события отображается текст в окне состояния, как показана рис. 32.2.

```
// Продемонстрировать применение компонента JTextField
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
   <applet code="JTextFieldDemo" width=300 height=50>
   </applet>
*/

public class JTextFieldDemo extends JApplet {
    JTextField jtf;
    public void init() {
        try {
            SwingUtilities.invokeLater(
                new Runnable() {
                    public void run() {
                        makeGUI( );
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
            // Нельзя создать по указанной причине
        }
    }
}
```

```

    }

    private void makeGUI() {
        // изменить поточную компоновку
        setLayout(new FlowLayout());
        // ввести текстовое поле на панели содержимого
        jtf = new JTextField(15);
        add(jtf);
        jtf.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                // отобразить текст, когда пользователь
                // нажимает клавишу <Enter>
                showStatus(jtf.getText());
            }
        });
    }
}

```

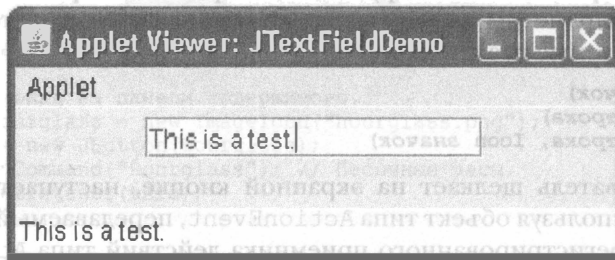


Рис. 32.2. Отображение текста в текстовом поле окна апплета `JTextFieldDemo`

Кнопки из библиотеки Swing

В библиотеке Swing определены четыре класса кнопок: `JButton`, `JToggleButton`, `JCheckBox` и `JRadioButton`. Все они являются производными от класса `AbstractButton`, расширяющего класс `JComponent`. Таким образом, у кнопок имеются общие характерные черты.

Класс `AbstractButton` содержит немало методов, позволяющих управлять поведением кнопок. С их помощью можно, например, определить различные значки, которые будут отображаться на месте кнопки, когда она отключена, нажата или выбрана. Другой значок можно использовать для динамической подстановки, чтобы он отображался при наведении указателя мыши на кнопку. Ниже приведены общие формы методов, с помощью которых можно задавать эти значки.

```

void setDisabledIcon(Icon di)
void setPressedIcon(Icon pi)
void setSelectedIcon(Icon si)
void setRolloverIcon(Icon ri)

```

Параметры *di*, *pi*, *si* и *ri* определяют значки, используемые для обозначения различных состояний. Текст, связанный с кнопкой, можно прочесть и записать с помощью приведенных ниже методов, где параметр *строка* обозначает текст надписи на кнопке.

```
String getText()
void setText(String строка)
```

Модель, применяемая во всех кнопках, определяется в интерфейсе `ButtonModel`. Кнопка генерирует событие действия, когда ее нажимает пользователь. Возможны и другие события. В последующих разделах будут подробнее рассмотрены классы отдельных кнопок.

Класс `JButton`

В классе `JButton` определяются функциональные возможности экранной кнопки. Простая форма конструктора этого класса была представлена в предыдущей главе. Компонент типа `JButton` позволяет связать с экранной кнопкой значок, символьную строку или же и то и другое. Ниже приведены три конструктора класса `JButton`, где параметры *значок* и *строка* обозначают соответственно значок и строку, используемые для кнопки.

```
JButton(Icon значок)
JButton(String строка)
JButton(String строка, Icon значок)
```

Когда пользователь щелкает на экранной кнопке, наступает событие типа `ActionEvent`. Используя объект типа `ActionEvent`, передаваемый методу `actionPerformed()` зарегистрированного приемника действий типа `ActionListener`, можно получить символьную строку с *командой действия*, связанной с данной кнопкой. По умолчанию эта символьная строка отображается в пределах кнопки. Но команду действия можно также задать, вызвав метод `setActionCommand()` для кнопки. А получить команду действия можно, вызвав метод `getActionCommand()` для объекта события. Этот метод объявляется следующим образом:

```
String getActionCommand()
```

Команда действия обозначает кнопку. Так, если в одном приложении используются две кнопки или больше, команда действия позволяет легко определить, какая именно кнопка была нажата.

В предыдущей главе был представлен пример применения текстовой кнопки. А в приведенном ниже примере апплета демонстрируется кнопка в виде значка. В данном примере отображаются четыре экранные кнопки и одна метка. Каждая кнопка отображает значок, представляющий разновидность часов. Когда пользователь щелкает на кнопке, в метке появляется название часов, как показано на рис. 32.3.

```
// Продемонстрировать применение компонента типа JButton в виде значка
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
  <applet code="JButtonDemo" width=250 height=750>
  </applet>
*/

public class JButtonDemo extends JApplet
implements ActionListener {
    JLabel jlab;
```

```
public void init() {
    try {
        SwingUtilities.invokeLater(
            new Runnable() {
                public void run() {
                    makeGUI();
                }
            }
        );
    } catch (Exception exc) {
        System.out.println("Can't create because of " + exc);
        // Нельзя создать по указанной причине
    }
}

private void makeGUI() {

    // изменить поточную компоновку
    setLayout(new FlowLayout());

    // ввести кнопки на панели содержимого.
    ImageIcon hourglass = new ImageIcon("hourglass.png");
    JButton jb = new JButton(hourglass);
    jb.setActionCommand("Hourglass"); // Песочные часы
    jb.addActionListener(this);
    add(jb);

    ImageIcon analog = new ImageIcon("analog.png");
    jb = new JButton(analog);
    jb.setActionCommand("Analog Clock"); // Аналоговые часы
    jb.addActionListener(this);
    add(jb);

    ImageIcon digital = new ImageIcon("digital.png");
    jb = new JButton(digital);
    jb.setActionCommand("Digital Clock"); // Цифровые часы
    jb.addActionListener(this);
    add(jb);

    ImageIcon stopwatch = new ImageIcon("stopwatch.png");
    jb = new JButton(stopwatch);
    jb.setActionCommand("Stopwatch"); // Секундомер
    jb.addActionListener(this);
    add(jb);

    // создать метку и ввести ее на панели содержимого
    JLabel jlab = new JLabel("Choose a Timepiece"); // выбрать часы
    add(jlab);
}

// обработать события от кнопок
public void actionPerformed(ActionEvent ae) {
    jlab.setText("You selected " + ae.getActionCommand());
    // Выбраны указанные часы
}
}
```

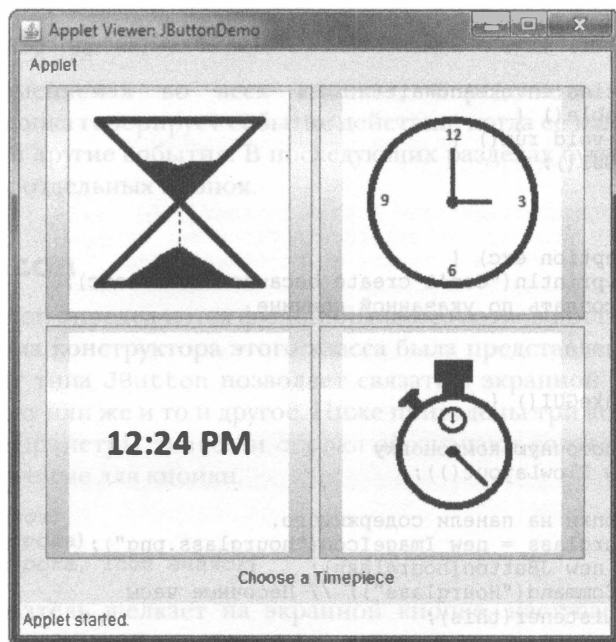


Рис. 32.3. Кнопки выбора часов в окне апплета JButtonDemo

Класс JToggleButton

Полезной разновидностью экранной кнопки является *переключатель*. Эта кнопка похожа на обычную кнопку, но действует иначе, поскольку может находиться в двух состояниях: нажатом и отпущенном. После щелчка на переключателе он остается нажатым, а не отпускается, как обычная экранная кнопка. Если после этого щелкнуть на переключателе еще раз, он отпускается. Таким образом, всякий раз, когда пользователь щелкает на переключателе, он переходит в одно из двух возможных состояний.

Переключатели определяются объектами класса JToggleButton, производного от класса AbstractButton. Помимо создания стандартных переключателей, класс JTogglekеButton служит суперклассом для двух других компонентов Swing, которые также представляют элементы управления, имеющие два состояния. Это классы JCheckBox и JRadioButton. Таким образом, класс JToggleButton определяет базовые функции всех компонентов, имеющих два состояния.

В классе JToggleButton определяется несколько конструкторов. Ниже приведен один из конструкторов, применяемых в примере из этого раздела.

```
JToggleButton(String строка)
```

Этот конструктор создает переключатель с текстом надписи, задаваемым в качестве параметра *строка*. Стандартным является отпущенное состояние переключателя.

чателя. Остальные конструкторы данного класса позволяют создавать переключатели с изображением или текстом надписи и изображением.

В классе `JToggleButton` применяется модель, определяемая во вложенном классе `JToggleButton.ToggleButtonModel`. Как правило, обращаться непосредственно к этой модели не требуется, чтобы воспользоваться стандартным переключателем.

Как и компонент типа `JButton`, компонент типа `JToggleButton` генерирует событие действия всякий раз, когда пользователь щелкает на переключателе. Но, в отличие от класса `JButton`, компонент типа `JToggleButton` генерирует также событие от элемента. Это событие используется теми компонентами, которые действуют по принципу выбора. Если переключатель типа `JToggleButton` нажат, он считается выбранным. Если пользователь отпускает переключатель, выбор отменяется.

Для обработки событий от элементов следует реализовать интерфейс `ItemListener`. Как пояснялось в главе 24, всякий раз, когда генерируется событие от элемента, оно передается методу `itemStateChanged()`, определяемому в интерфейсе `ItemListener`. Из метода `itemStateChanged()` может быть вызван метод `getItem()` для объекта типа `ItemEvent`, чтобы получить ссылку на экземпляр класса `JToggleButton`, сгенерировавший данное событие. Ниже приведена общая форма метода `getItem()`.

Object getItem()

Этот метод возвращает ссылку на кнопку. Эту ссылку следует привести к типу `JToggleButton`.

Чтобы определить состояние переключателя, проще всего вызвать метод `isSelected()`, наследуемый из класса `AbstractButton`, для кнопки, сгенерировавшей событие. Ниже приведена общая форма метода `isSelected()`. Этот метод возвращает логическое значение `true`, если кнопка выбрана, а иначе — логическое значение `false`.

boolean isSelected()

В приведенном ниже примере апплета демонстрируется применение переключателя. Обратите внимание на приемник событий, который просто вызывает метод `isSelected()`, чтобы определить состояние кнопки. На рис. 32.4 показано окно апплета из данного примера с переключателем в нажатом состоянии.

```
// Продемонстрировать применение компонента типа JToggleButton
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
  <applet code="JToggleButtonDemo" width=200 height=80>
  </applet>
*/

public class JToggleButtonDemo extends JApplet {

    JLabel jlab;
    JToggleButton jtnb;
```

```

public void init() {
    try {
        SwingUtilities.invokeLaterAndWait(
            new Runnable() {
                public void run() {
                    makeGUI();
                }
            }
        );
    } catch (Exception exc) {
        System.out.println("Can't create because of " + exc);
        // Нельзя создать по указанной причине
    }
}

private void makeGUI() {
    // изменить поточную компоновку
    setLayout(new FlowLayout());

    // создать метку
    jlab = new JLabel("Button is off."); // Кнопка отпущена

    // создать переключатель
    jtnb = new JToggleButton("On/Off");

    // ввести приемник событий от переключателя
    jtnb.addItemListener(new ItemListener() {
        public void itemStateChanged(ItemEvent ie) {
            if(jtnb.isSelected())
                jlab.setText("Button is on."); // Кнопка нажата
            else
                jlab.setText("Button is off."); // Кнопка отпущена
        }
    });

    // ввести переключатель и метку на панель содержимого
    add(jtnb);
    add(jlab);
}
}

```

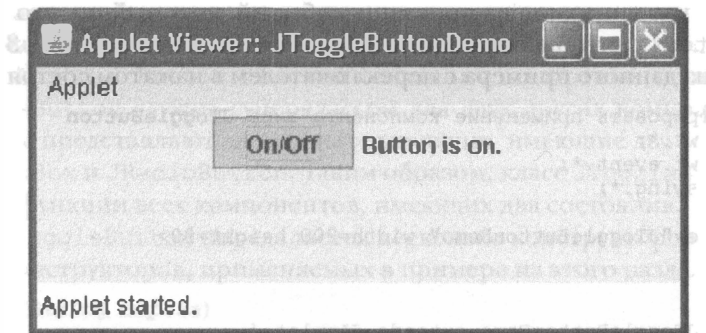


Рис. 32.4. Нажатие переключателя в окне апплета JToggleButtonDemo

Флажки

Класс `JCheckBox` определяет функции флажка. Его суперклассом служит класс `JToggleButton`, поддерживающий кнопки с двумя состояниями, как пояснялось выше. В классе `JCheckBox` определяется ряд конструкторов. Один из них выглядит следующим образом:

`JCheckBox(String строка)`

Этот конструктор создает флажок с текстом метки, определяемым в качестве параметра *строка*. Остальные конструкторы позволяют определить исходное состояние выбора флажка и указать значок.

Если пользователь устанавливает или сбрасывает флажок, генерируется событие типа `ItemEvent`. Чтобы получить ссылку на компонент типа `JCheckBox`, сгенерировавший событие, следует вызвать метод `getItem()` для объекта типа `ItemEvent`, который передается в качестве события от элемента методу `itemStateChanged()`, определяемому в интерфейсе `ItemListener`. Определить выбранное состояние флажка проще всего, вызвав метод `isSelected()` для экземпляра класса `JCheckBox`.

В приведенном ниже примере апплета демонстрируется применение флажков. В окне этого апплета отображаются четыре флажка и метка. Когда пользователь щелкает кнопкой мыши на флажке, наступает событие типа `ItemEvent`. Из метода `itemStateChanged()` вызывается метод `getItem()` для получения ссылки на объект типа `JCheckBox`, сгенерировавший событие от элемента. Далее вызывается метод `isSelected()` с целью определить установленное или сброшенное состояние флажка. А метод `getText()` вызывается с целью получить текст, выводимый на месте метки данного флажка. На рис. 32.5 показано окно апплета из данного примера с одним из установленных флажков.

```
// Продемонстрировать применение компонента JCheckbox
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
  <applet code="JCheckBoxDemo" width=270 height=50>
  </applet>
*/

public class JCheckBoxDemo extends JApplet
implements ItemListener {
    JLabel jlab;

    public void init() {
        try {
            SwingUtilities.invokeAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
        }
    }
}
```



```

        // Нельзя создать по указанной причине
    }
}

private void makeGUI() {
    // изменить поточную компоновку
    setLayout(new FlowLayout());

    // ввести флажки на панели содержимого
    JCheckBox cb = new JCheckBox("C");
    cb.addItemListener(this);
    add(cb);

    cb = new JCheckBox("C++");
    cb.addItemListener(this);
    add(cb);

    cb = new JCheckBox("Java");
    cb.addItemListener(this);
    add(cb);

    cb = new JCheckBox("Perl");
    cb.addItemListener(this);
    add(cb);

    // создать метку и ввести ее на панели содержимого
    jlab = new JLabel("Select languages");
    // Выбор языка программирования
    add(jlab);
}

// обработать событие от флажка
public void itemStateChanged(ItemEvent ie) {
    JCheckBox cb = (JCheckBox)ie.getItem();

    if(cb.isSelected())
        jlab.setText(cb.getText() + " is selected");
        // Выбран указанный язык программирования
    else
        jlab.setText(cb.getText() + " is cleared");
        // Указанный флажок сброшен
}
}

```

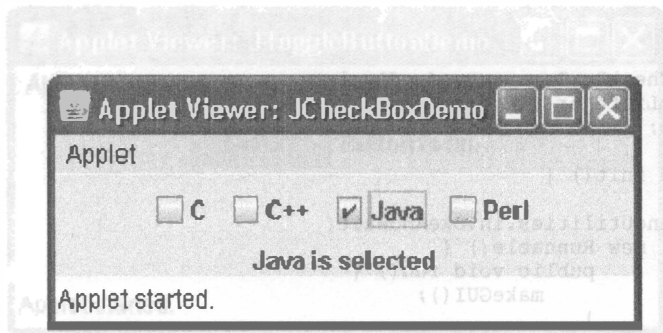


Рис. 32.5. Установка флажка в окне апплета JCheckBoxDemo

Кнопки-переключатели

Кнопки-переключатели образуют группу взаимоисключающих кнопок, из которых можно выбрать только одну. Они поддерживаются в классе `JRadioButton`, расширяющем класс `JToggleButton`. В классе `JRadioButton` предоставляется несколько конструкторов. Ниже приведен конструктор, используемый в примере из этого раздела.

`JRadioButton(String строка)`

Здесь параметр *строка* обозначает метку кнопки-переключателя. Остальные конструкторы позволяют определить исходное состояние кнопки-переключателя и указать для нее значок.

Кнопки-переключатели следует объединить в группу, где можно выбрать только одну из них. Так, если пользователь выбирает какую-нибудь кнопку-переключатель из группы, то кнопка-переключатель, выбранная ранее в этой группе, автоматически выключается. Для создания группы кнопок-переключателей служит класс `ButtonGroup`. С этой целью вызывается его конструктор по умолчанию. После этого в группу можно ввести отдельные кнопки-переключатели с помощью приведенного ниже метода, где параметр *ab* обозначает ссылку на кнопку-переключатель, которую требуется ввести в группу.

`void add(AbstractButton ab)`

Компонент типа `JRadioButton` генерирует события действия, события от элементов и события изменения всякий раз, когда выбирается другая кнопка-переключатель в группе. Зачастую обрабатывается событие действия, а это, как правило, означает необходимость реализовать интерфейс `ActionListener`, в котором определяется единственный метод `actionPerformed()`. В этом методе можно несколькими способами выяснить, какая именно кнопка-переключатель была выбрана. Во-первых, можно проверить ее команду действия, вызвав метод `getActionCommand()`. По умолчанию команда действия аналогична метке кнопки, но, вызвав метод `setActionCommand()` для кнопки-переключателя, можно задать какую-нибудь другую команду действия. Во-вторых, можно вызвать метод `getSource()` для объекта типа `ActionEvent` и проверить ссылку по отношению к кнопкам-переключателям. И наконец, для каждой кнопки можно вызвать свой обработчик событий действия, реализуемый в виде анонимного класса или лямбда-выражения. Не следует, однако, забывать, что всякий раз, когда наступает событие действия, оно означает, что выбранная кнопка-переключатель была изменена и что была выбрана одна и только одна кнопка-переключатель.

В приведенном ниже примере апплета демонстрируется применение кнопок-переключателей. В этом апплете создаются и объединяются в группу три кнопки-переключателя. Как пояснялось ранее, это требуется для того, чтобы они действовали, взаимно исключая друг друга. При выборе кнопки-переключателя наступает событие действия, которое обрабатывается методом `actionPerformed()`. В этом обработчике событий метод `getActionCommand()` получает текст, связанный с кнопкой-переключателем, чтобы отобразить его на месте метки. На рис. 32.6 показано окно апплета из данного примера с одной из выбранных кнопок-переключателей.

```

// Продемонстрировать применение компонента JRadioButton
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
  <applet code="JRadioButtonDemo" width=300 height=50>
  </applet>
*/
public class JRadioButtonDemo extends JApplet
implements ActionListener {
    JLabel jlab;
    public void init() {
        try {
            SwingUtilities.invokeAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
            // Нельзя создать по указанной причине
        }
    }

    private void makeGUI() {
        // изменить поточную компоновку
        setLayout(new FlowLayout());
        // создать кнопки-переключатели и ввести
        // их на панели содержимого
        JRadioButton b1 = new JRadioButton("A");
        b1.addActionListener(this);
        add(b1);

        JRadioButton b2 = new JRadioButton("B");
        b2.addActionListener(this);
        add(b2);

        JRadioButton b3 = new JRadioButton("C");
        b3.addActionListener(this);
        add(b3);
        // определить группу кнопок
        ButtonGroup bg = new ButtonGroup();
        bg.add(b1);
        bg.add(b2);
        bg.add(b3);

        // создать метку и ввести ее на панели содержимого
        jlab = new JLabel("Select One");
        // Выбор одной из кнопок-переключателей
        add(jlab);
    }
    // обработать событие выбора кнопки-переключателя
    public void actionPerformed(ActionEvent ae) {
        jlab.setText("You selected " + ae.getActionCommand());
        // Выбрана указанная кнопка-переключатель
    }
}

```

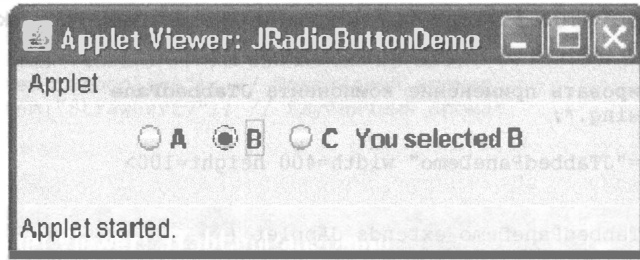


Рис. 32.6. Выбор кнопки-переключателя в окне апплета JRadioButtonDemo

Класс JTabbedPane

Класс JTabbedPane инкапсулирует *панель с вкладками*. Он управляет рядом компонентов, связывая их с помощью вкладок. При выборе вкладки связанный с ней компонент выступает на передний план. Панели с вкладками широко применяются в ГПИ современных приложений, и вам, без сомнения, придется не раз употреблять их в своих прикладных программах. Несмотря на сложную структуру панели с вкладками, создавать и пользоваться ею очень просто.

В классе JTabbedPane определяются три конструктора. В примере из этого раздела используется конструктор по умолчанию, создающий пустой элемент управления с вкладками, располагаемыми вдоль верхнего края панели. Два других конструктора позволяют определить расположение вкладок вдоль одной из четырех сторон окна. В классе JTabbedPane применяется модель, определяемая в классе SingleSelectionModel.

Для ввода вкладок на панели вызывается метод `addTab()`. Ниже приведена одна из общих форм этого метода.

```
void addTab(String имя, Component компонент)
```

Здесь параметр *имя* обозначает указанное имя вкладки, а параметр *компонент* — вводимый на вкладке компонент. Обычно на вкладке вводится компонент типа JPanel, содержащий группу связанных вместе компонентов. Благодаря этому вкладка может содержать ряд компонентов.

В общем, процедура употребления панели с вкладками сводится к следующему.

1. Создать объект класса JTabbedPane.
2. Вызвать метод `addTab()`, чтобы ввести каждую вкладку по отдельности.
3. Ввести панель с вкладками на панели содержимого.

В приведенном ниже примере апплета демонстрируется процесс создания панели с вкладками. Первая вкладка озаглавлена как **Cities** (Города) и содержит четыре кнопки. Каждая кнопка отображает название города. Вторая вкладка озаглавлена как **Colors** (Цвета) и содержит три флажка. Каждый флажок отображает название цвета. И наконец, третья вкладка озаглавлена как **Flavors** (Ароматы) и содержит один комбинированный список. Из этого списка пользователь может выбрать

один из трех ароматов. На рис. 32.7 показано содержимое трех вкладок, выбираемых на панели в окне апплета из данного примера.

```
// Продемонстрировать применение компонента JTabbedPane
import javax.swing.*;
/*
<applet code="JTabbedPaneDemo" width=400 height=100>
</applet>
*/
public class JTabbedPaneDemo extends JApplet {
    public void init() {
        try {
            SwingUtilities.invokeAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
            // Нельзя создать по указанной причине
        }
    }

    private void makeGUI() {
        JTabbedPane jtp = new JTabbedPane();
        jtp.addTab("Cities", new CitiesPanel()); // вкладка городов
        jtp.addTab("Colors", new ColorsPanel()); // вкладка цветов
        jtp.addTab("Flavors", new FlavorsPanel()); // вкладка ароматов
        add(jtp);
    }
}

// создать панели, которые будут введены на панели с вкладками
class CitiesPanel extends JPanel {
    public CitiesPanel() {
        JButton b1 = new JButton("New York"); // Нью-Йорк
        add(b1);
        JButton b2 = new JButton("London"); // Лондон
        add(b2);
        JButton b3 = new JButton("Hong Kong"); // Гонконг
        add(b3);
        JButton b4 = new JButton("Tokyo"); // Токио
        add(b4);
    }
}

class ColorsPanel extends JPanel {
    public ColorsPanel() {
        JCheckBox cb1 = new JCheckBox("Red"); // Красный
        add(cb1);
        JCheckBox cb2 = new JCheckBox("Green"); // Зеленый
        add(cb2);
        JCheckBox cb3 = new JCheckBox("Blue"); // Синий
        add(cb3);
    }
}

class FlavorsPanel extends JPanel {
```

```

public FlavorsPanel() {
    JComboBox<String> jcb = new JComboBox<String>();
    jcb.addItem("Vanilla"); // Ванильный аромат
    jcb.addItem("Chocolate"); // Шоколадный аромат
    jcb.addItem("Strawberry"); // Клубничный аромат
    add(jcb);
}
}

```

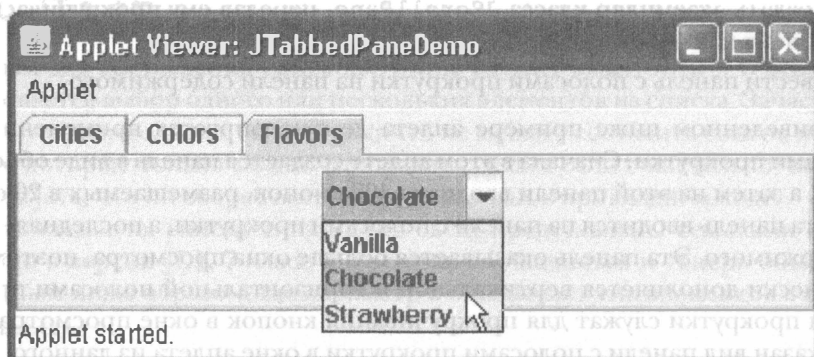
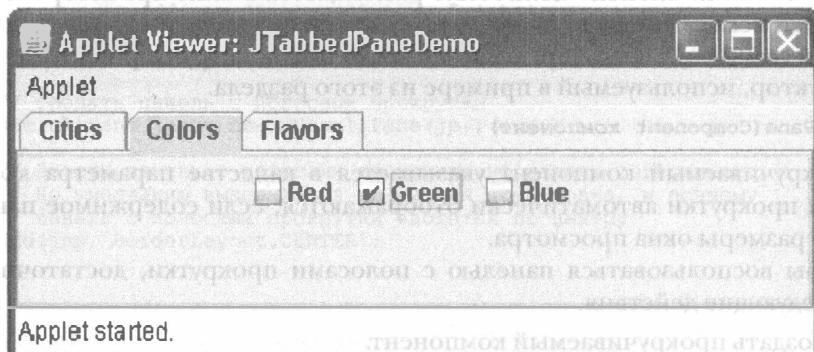


Рис. 32.7. Выбор каждой из трех вкладок на панели в окне апплета JTabbedPaneDemo

Класс JScrollPane

Класс `JScrollPane` представляет легковесный контейнер, автоматически выполняющий прокрутку другого компонента. Прокручиваться может как отдельный компонент (например, таблица), так и группа компонентов, содержащихся в другом легковесном контейнере, например `JPanel`. Но в любом случае прокручиваемый компонент дополняется горизонтальной и/или вертикальной полосой прокрутки, если он больше области просмотра, что позволяет прокручивать компонент в пределах панели. Класс `JScrollPane` автоматизирует процесс прокрутки, избавляя от необходимости управлять отдельными полосами прокрутки.

Просматриваемая область панели с полосами прокрутки называется *окном просмотра*. Это окно, в котором отображается прокручиваемый компонент. Таким образом, в окне просмотра будет показана видимая часть прокручиваемого компонента. Полосы прокрутки служат для прокручивания компонента в окне просмотра. По умолчанию класс `JScrollPane` динамически добавляет или удаляет полосу прокрутки по мере надобности. Так, если компонент оказывается больше по высоте, чем окно просмотра, то оно дополняется вертикальной полосой прокрутки. А если компонент полностью размещается в окне просмотра, то полосы прокрутки исключаются.

В классе `JScrollPane` определяется несколько конструкторов. Ниже приведен конструктор, используемый в примере из этого раздела.

`JScrollPane` (Component компонент)

Прокручиваемый компонент указывается в качестве параметра *компонент*. Полосы прокрутки автоматически отображаются, если содержимое панели превышает размеры окна просмотра.

Чтобы воспользоваться панелью с полосами прокрутки, достаточно выполнить следующие действия.

1. Создать прокручиваемый компонент.
2. Создать экземпляр класса `JScrollPane`, передав ему прокручиваемый компонент как объект.
3. Ввести панель с полосами прокрутки на панели содержимого.

В приведенном ниже примере апплета демонстрируется применение панели с полосами прокрутки. Сначала в этом апплете создается панель в виде объекта типа `JPanel`, а затем на этой панели вводится 400 кнопок, размещаемых в 20 столбцах. Далее эта панель вводится на панели с полосами прокрутки, а последняя — на панели содержимого. Эта панель оказывается больше окна просмотра, поэтому она автоматически дополняется вертикальной и горизонтальной полосами прокрутки. Полосы прокрутки служат для прокручивания кнопок в окне просмотра. На рис. 32.8 показан вид панели с полосами прокрутки в окне апплета из данного примера.

```
// Продемонстрировать применение компонента JScrollPane
import java.awt.*;
import javax.swing.*;
/*
<applet code="JScrollPaneDemo" width=300 height=250>
</applet>
```

```

*/
public class JScrollPaneDemo extends JApplet {
    public void init() {
        try {
            SwingUtilities.invokeAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
            // Нельзя создать по указанной причине
        }
    }

    private void makeGUI() {
        // ввести 400 кнопок на панели
        JPanel jp = new JPanel();
        jp.setLayout(new GridLayout(20, 20));
        int b = 0;
        for(int i = 0; i < 20; i++) {
            for(int j = 0; j < 20; j++) {
                jp.add(new JButton("Button " + b));
                ++b;
            }
        }
        // создать панель с полосами прокрутки
        JScrollPane jsp = new JScrollPane(jp);

        // Ввести панель с полосами прокрутки на панели содержимого.
        // По умолчанию выполняется граничная компоновка, и поэтому
        // панель с полосами прокрутки вводится по центру
        add(jsp, BorderLayout.CENTER);
    }
}

```

Класс JList

Базовым для составления списков в Swing служит класс JList. В этом классе поддерживается выбор одного или нескольких элементов из списка. Зачастую список состоит из символьных строк, но ничто не мешает составить список из любых объектов, которые только можно отобразить. Класс JList настолько часто применяется в Java, что он скорее всего встречался вам в прикладном коде.

Раньше элементы списка типа JList были представлены ссылками на класс Object. Но в версии JDK 7 класс JList стал обобщенным и теперь объявляется приведенным ниже образом, где параметр E обозначает тип элементов в списке.

```
class JList<E>
```

В классе JList предоставляется несколько конструкторов. Ниже приведен один из наиболее употребительных конструкторов данного класса. Этот конструктор создает список типа JList, содержащий элементы в массиве, обозначаемом в качестве параметра *элементы*.

```
JList(E[] элементы)
```

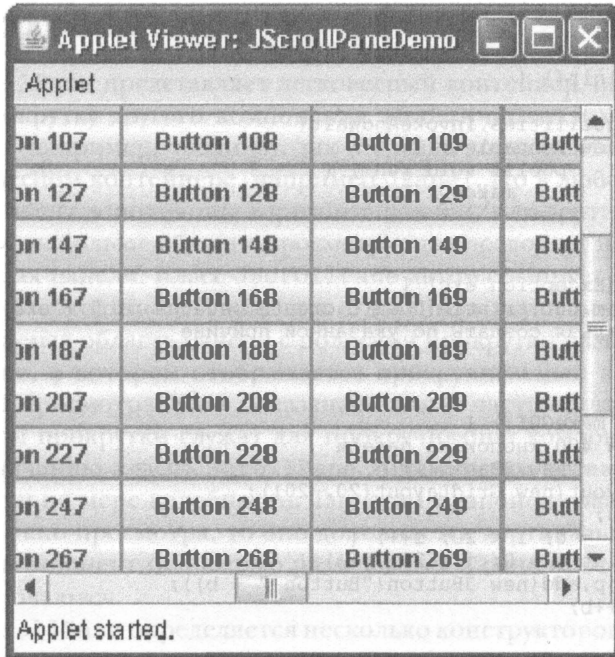



Рис. 32.8. Вид панели с полосами прокрутки в окне апплета JScrollPaneDemo

Класс `JList` основывается на двух моделях. Первая модель определяется в интерфейсе `ListModel` и устанавливает порядок доступа к данным в списке. Вторая модель определяется в интерфейсе `ListSelectionModel`, где объявляются методы, позволяющие выявить выбранный из списка элемент (или элементы).

Несмотря на то что компонент типа `JList` вполне способен действовать самостоятельно, он обычно размещается на панели типа `JScrollPane`. Благодаря этому длинные списки становятся автоматически прокручиваемыми. Этим упрощается не только построение ГПИ, но и изменение количества записей в списке, не требуя изменять размеры компонента типа `JList`.

Компонент типа `JList` генерирует событие типа `ListSelectionEvent`, когда пользователь выбирает элемент или изменяет выбор элемента в списке. Это событие наступает и в том случае, если пользователь отменяет выбор элемента. Оно обрабатывается приемником событий, реализующим интерфейс `ListSelectionListener`, в котором определяется единственный метод `valueChanged()`:

```
void valueChanged(ListSelectionEvent событие_списка)
```

где параметр *событие_списка* обозначает ссылку на событие. И хотя в классе `ListSelectionEvent` предоставляются свои методы для выяснения событий, наступающих при выборе элементов из списка, как правило, для этой цели достаточно обратиться непосредственно к объекту типа `JList`. Класс `ListSelectionEvent` и интерфейс `ListSelectionListener` определены в пакете `javax.swing.event`.

По умолчанию компонент типа `JList` позволяет выбирать несколько элементов из списка, но это поведение можно изменить, вызвав метод `setSelectionMode()`, определяемый в классе `JList`. Ниже приведена его общая форма.

```
void setSelectionMode(int режим)
```

Здесь параметр *режим* обозначает заданный режим выбора. Этот параметр должен принимать значение одной из следующих констант, определенных в интерфейсе `ListSelectionModel`:

```
SINGLE_SELECTION  
SINGLE_INTERVAL_SELECTION  
MULTIPLE_INTERVAL_SELECTION
```

По умолчанию выбирается значение последней константы, позволяющее выбирать несколько интервалов элементов из списка. Если же задан режим выбора в одном интервале (константа `SINGLE_INTERVAL_SELECTION`), то выбрать можно только один ряд элементов из списка. А если задан режим выбора одного элемента (константа `SINGLE_SELECTION`), то выбрать можно только один элемент из списка. Разумеется, один элемент можно выбрать и в двух других режимах, но эти режимы позволяют также выбирать несколько элементов из списка.

Вызвав метод `getSelectedIndex()`, можно получить индекс первого выбранного элемента, который оказывается также индексом единственного выбранного элемента в режиме `SINGLE_SELECTION`. Ниже приведена общая форма метода `getSelectedIndex()`.

```
int getSelectedIndex()
```

Индексация элементов списка начинается с нуля. Так, если выбран первый элемент, метод `getSelectedIndex()` возвращает нулевое значение. А если не выбрано ни одного элемента, то возвращается значение `-1`.

Вместо того чтобы получать индекс выбранного элемента, можно получить значение, связанное с выбранным элементом, вызвав метод `getSelectedValue()`. Ниже приведена общая форма этого метода.

```
E getSelectedValue()
```

Этот метод возвращает ссылку на первое выбранное значение. Если не выбрано ни одного значения, то возвращается пустое значение `null`.

В приведенном ниже примере апплета демонстрируется применение простого компонента типа `JList`, содержащего список городов. Всякий раз, когда из этого списка выбирается город, наступает событие типа `ListSelectionEvent`, обрабатываемое методом `valueChanged()`, определяемым в интерфейсе `ListSelectionListener`. Этот метод получает индекс выбранного элемента и отображает имя выбранного города на месте метки. На рис. 32.9 показано, как город выбирается из списка в окне апплета из данного примера.

```
// Продемонстрировать применение компонента JList  
import javax.swing.*;  
import javax.swing.event.*;  
import java.awt.*;  
import java.awt.event.*;  
  
/*
```

```

<applet code="JListDemo" width=200 height=120>
</applet>
*/

public class JListDemo extends JApplet {
    JList<String> jlst;
    JLabel jlab;
    JScrollPane jscrlp;
    // создать массив из названий городов
    String Cities[] = { "New York", "Chicago", "Houston",
                        "Denver", "Los Angeles", "Seattle",
                        "London", "Paris", "New Delhi",
                        "Hong Kong", "Tokyo", "Sydney" };

    public void init() {
        try {
            SwingUtilities.invokeLater(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
            // Нельзя создать по указанной причине
        }
    }

    private void makeGUI() {
        // изменить поточную компоновку
        setLayout(new FlowLayout());

        // создать список на основе компонента типа JList
        jlst = new JList<String>(Cities);

        // задать режим выбора единственного элемента из списка
        jlst.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

        // ввести список на панели с полосами прокрутки
        jscrlp = new JScrollPane(jlst);

        // задать предпочтительные размеры панели с полосами прокрутки
        jscrlp.setPreferredSize(new Dimension(120, 90));

        // создать метку для отображения выбранного города
        jlab = new JLabel("Choose a City"); // Выбор города

        // ввести приемник событий выбора из списка
        jlst.addListSelectionListener(new ListSelectionListener() {
            public void valueChanged(ListSelectionEvent le) {
                // получить индекс измененного элемента
                int idx = jlst.getSelectedIndex();
                // отобразить сделанный выбор, если элемент
                // был выбран из списка
                if(idx != -1)
                    jlab.setText("Current selection: " + Cities[idx]);
                // Текущий выбор: указанный город
            } else // В противном случае еще раз предложить
                // выбрать город из списка
                jlab.setText("Choose a City"); // Выбор города
        })
    }
}

```

```

    });
    // ввести список и метку на панели содержимого
    add(jscrlp);
    add(jlab);
}
}

```

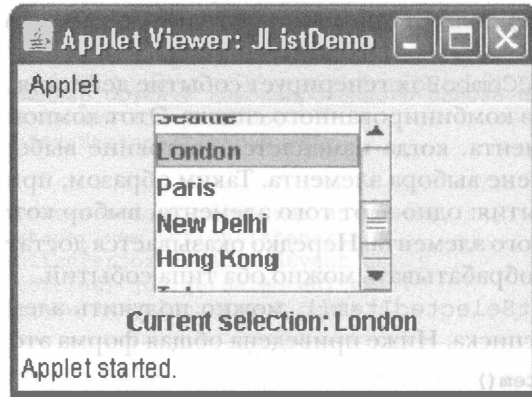


Рис. 32.9. Выбор города из списка в окне апплета JListDemo

Класс JComboBox

С помощью класса `JComboBox` из библиотеки Swing определяется компонент, называемый *комбинированным списком* и сочетающий текстовое поле с раскрывающимся списком. Как правило, комбинированный список отображает одну запись, но он может отображать и раскрывающийся список, позволяющий выбирать другие элементы. Имеется также возможность создать комбинированный список, позволяющий вводить выбираемый элемент в текстовом поле.

Раньше элементы комбинированного списка на основе компонента типа `JComboBox` были представлены ссылками на класс `Object`. Но в версии JDK 7 класс `JComboBox` был сделан обобщенным и теперь объявляется приведенным ниже образом, где параметр `E` обозначает тип элементов комбинированного списка.

```
class JComboBox<E>
```

Ниже приведен конструктор класса `JComboBox`, используемый в примере из этого раздела.

```
JComboBox(E[] элементы)
```

Здесь параметр *элементы* обозначает массив, инициализирующий комбинированный список. В классе `JComboBox` имеются и другие конструкторы.

В классе `JComboBox` применяется модель, определяемая в интерфейсе `ComboBoxModel`. Для создания изменяемых комбинированных списков (т.е. таких списков, элементы которых могут изменяться) применяется модель, определяемая в интерфейсе `MutableComboBoxModel`.

Кроме передачи массива элементов, которые должны быть отображены в раскрывающемся списке, элементы можно динамически вводить в список с помощью метода `addItem()`. Ниже приведена общая форма этого метода.

```
void addItem(Е объект)
```

Здесь параметр *объект* обозначает объект, вводимый в комбинированный список. Метод `addItem()` должен применяться только к изменяемым комбинированным спискам.

Компонент типа `JComboBox` генерирует событие действия, когда пользователь выбирает элемент из комбинированного списка. Этот компонент генерирует также событие от элемента, когда изменяется состояние выбора, что происходит при выборе или отмене выбора элемента. Таким образом, при изменении выбора происходят два события: одно — от того элемента, выбор которого был отменен, другое — от выбранного элемента. Нередко оказывается достаточно принимать события действия, но обрабатывать можно оба типа событий.

Вызвав метод `getSelectedItem()`, можно получить элемент, выбранный из комбинированного списка. Ниже приведена общая форма этого метода.

```
Object getItemSelected()
```

Значение, возвращаемое этим методом, следует привести к типу объекта, хранящегося в списке. В приведенном ниже примере апплета демонстрируется применение комбинированного списка. Этот список содержит элементы "Hourglass" (Песочные часы), "Analog" (Аналоговые часы), "Digital" (Цифровые часы) и "Stopwatch" (Секундомер). Если пользователь выберет часы, метка обновится, отображая значок данной разновидности часов. Обратите внимание, как мало кода требуется, чтобы воспользоваться этим сложным компонентом. На рис. 32.10 показан комбинированный список в окне апплета из данного примера.

```
// Продемонстрировать применение компонента типа JComboBox
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/*
  <applet code="JComboBoxDemo" width=300 height=200>
  </applet>
*/

public class JComboBoxDemo extends JApplet {
    JLabel jlab;
    ImageIcon hourglass, analog, digital, stopwatch;
    JComboBox<String> jcb;

    String timepieces[] = { "Hourglass", "Analog", "Digital", "Stopwatch" };

    public void init() {
        try {
            SwingUtilities.invokeAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        }
    }
}
```

```

    } catch (Exception exc) {
        System.out.println("Can't create because of " + exc);
        // Нельзя создать по указанной причине
    }
}

private void makeGUI() {

    // изменить поточную компоновку
    setLayout(new FlowLayout());

    // получить экземпляр объекта комбинированного списка и
    // ввести его на панели содержимого
    jcb = new JComboBox<String>(timepieces);
    add(jcb);

    // обработать события выбора элементов из списка
    jcb.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            String s = (String) jcb.getSelectedItem();
            jlab.setIcon(new ImageIcon(s + ".png"));
        }
    });

    // создать метку и ввести ее на панели содержимого
    jlab = new JLabel(new ImageIcon("hourglass.png"));
    add(jlab);
}
}

```

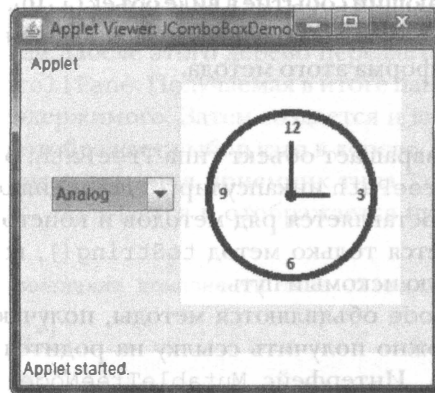


Рис. 32.10. Выбор часов из комбинированного списка в окне апплета JComboBoxDemo

Деревья

Дерево — это компонент для иерархического представления данных. В таком представлении пользователь может разворачивать или свертывать отдельные узлы. В библиотеке Swing деревья реализуются в классе `JTree`. Ниже приведены некоторые из конструкторов этого класса.

```

JTree(Object объект[])
JTree(Vector<?> v)
JTree(TreeNode tn)

```

В первой форме конструктора дерево создается из элементов массива *объект*, во второй форме — из элементов вектора *v*, в третьей — на основании корневого узла *tn*. Несмотря на то что класс `JTree` входит в пакет `javax.swing`, он поддерживает классы и интерфейсы, определенные в пакете `javax.swing.tree`. Дело в том, что количество классов и интерфейсов, требующихся для поддержки компонента типа `JTree`, слишком велико.

Класс `JTree` основывается на двух моделях типа `TreeModel` и `TreeSelectionModel`. Компонент типа `JTree` генерирует различные события, но к деревьям непосредственное отношение имеют лишь три типа событий: `TreeExpansionEvent`, `TreeSelectionEvent` и `TreeModelEvent`. Событие типа `TreeExpansionEvent` наступает при разворачивании или сворачивании узла дерева, событие типа `TreeSelectionEvent` — при выборе пользователем узла дерева или отмене его выбора, а событие типа `TreeModelEvent` — при изменении данных или структуры дерева. Эти события отслеживаются приемниками типа `TreeExpansionListener`, `TreeSelectionListener` и `TreeModelListener` соответственно. Классы событий, происходящих в дереве, а также интерфейсы приемников этих событий определены в пакете `javax.swing.event`.

В примере апплета из этого раздела обрабатывается событие типа `TreeSelectionEvent`. Чтобы принять это событие, следует реализовать интерфейс `TreeSelectionListener`. В этом интерфейсе определяется единственный метод `valueChanged()`, получающий событие в виде объекта типа `TreeSelectionEvent`. Вызвав метод `getPath()`, можно получить путь к выбранному объекту события. Ниже приведена общая форма этого метода.

```

TreePath getPath()

```

Метод `getPath()` возвращает объект типа `TreePath`, описывающий путь к измененному узлу. Класс `TreePath` инкапсулирует сведения о пути к определенному узлу дерева. В нем предоставляется ряд методов и конструкторов. В примере из этого раздела используется только метод `toString()`, возвращающий символьную строку, описывающую искомый путь.

В интерфейсе `TreeNode` объявляются методы, получающие сведения об узле дерева. В частности, можно получить ссылку на родительский узел или список порожденных им узлов. Интерфейс `MutableTreeNode` расширяет интерфейс `TreeNode`. В нем объявляются методы, позволяющие вводить и удалять порожденные узлы или изменять родительский узел дерева.

Класс `DefaultMutableTreeNode` реализует интерфейс `MutableTreeNode`. Он представляет узел дерева. Ниже приведен один из его конструкторов.

```

DefaultMutableTreeNode(Object объект)

```

Здесь параметр *объект* обозначает тот объект, который требуется заключить в данном узле дерева. У нового узла дерева отсутствует родительский или порожденный узел.

Чтобы создать иерархию из трех узлов дерева, достаточно вызвать метод `add()` из класса `DefaultMutableTreeNode`. Ниже приведена общая форма этого метода, где параметр *потомок* обозначает изменяющийся узел дерева, который требуется ввести в качестве порожденного от текущего узла.

```
void add(DefaultMutableTreeNode потомок)
```

Сам компонент типа `JTree` не предоставляет никаких возможностей для прокрутки. Поэтому этот компонент обычно размещается в контейнере типа `JScrollPane`. Таким образом, большое дерево можно прокрутить в окне просмотра меньших размеров.

Ниже перечислены действия, которые требуется выполнить, чтобы воспользоваться деревом.

1. Создать экземпляр класса `JTree`.
2. Создать экземпляр класса `JScrollPane` и определить дерево в качестве прокручиваемого объекта.
3. Ввести дерево на панели с полосами прокрутки.
4. Ввести панель с полосами прокрутки на панели содержимого.

В приведенном ниже примере апплета демонстрируется создание дерева и обработка событий выбора его узлов. В данном апплете создается изменяемый узел дерева в виде экземпляра класса `DefaultMutableTreeNode` с заголовком `Options` (Варианты выбора). Это самый верхний узел в иерархии дерева. Затем создаются дополнительные узлы дерева и вызывается метод `add()` для присоединения этих узлов к дереву. Ссылка на верхний узел дерева передается в качестве параметра конструктору класса `JTree`. После этого дерево передается в качестве параметра конструктору класса `JScrollPane`. Получаемая в итоге панель с полосами прокрутки вводится на панели содержимого. Затем создается и вводится метка на панели содержимого. Эта метка отображает выбор узла в дереве. Для получения событий выбора узлов в дереве регистрируется приемник типа `TreeSelectionListener`. В методе `valueChanged()` получается и отображается путь к текущему месту выбора в дереве.

```
// Продемонстрировать применение компонента типа JTree
import java.awt.*;
import javax.swing.event.*;
import javax.swing.*;
import javax.swing.tree.*;

/*
  <applet code="JTreeDemo" width=400 height=200>
  </applet>
*/

public class JTreeDemo extends JApplet {
    JTree tree;
    JLabel jlab;
    public void init() {
        try {
            SwingUtilities.invokeAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        }
    }
}
```



```

        }
    }
};
} catch (Exception exc) {
    System.out.println("Can't create because of " + exc);
    // Нельзя создать по указанной причине
}
}

private void makeGUI() {
    // создать самый верхний узел дерева
    DefaultMutableTreeNode top =
        new DefaultMutableTreeNode("Options");

    // создать поддерево "А"
    DefaultMutableTreeNode a = new DefaultMutableTreeNode("А");
    top.add(a);
    DefaultMutableTreeNode a1 = new DefaultMutableTreeNode("А1");
    a.add(a1);
    DefaultMutableTreeNode a2 = new DefaultMutableTreeNode("А2");
    a.add(a2);

    // создать поддерево "В"
    DefaultMutableTreeNode b = new DefaultMutableTreeNode("В");
    top.add(b);
    DefaultMutableTreeNode b1 = new DefaultMutableTreeNode("В1");
    b.add(b1);
    DefaultMutableTreeNode b2 = new DefaultMutableTreeNode("В2");
    b.add(b2);
    DefaultMutableTreeNode b3 = new DefaultMutableTreeNode("В3");
    b.add(b3);

    // создать дерево
    tree = new JTree(top);

    // ввести дерево на панели прокрутки
    JScrollPane jsp = new JScrollPane(tree);

    // ввести панель с полосами прокрутки на панели содержимого
    add(jsp);

    // ввести метку на панели содержимого
    jlab = new JLabel();
    add(jlab, BorderLayout.SOUTH);

    // обработать события выбора узлов дерева
    tree.addTreeSelectionListener(new TreeSelectionListener() {
        public void valueChanged(TreeSelectionEvent tse) {
            jlab.setText("Selection is " + tse.getPath());
            // Выбран узел дерева по указанному пути
        }
    });
}
}

```

На рис. 32.11 показано дерево, развернутое в окне апплета из данного примера. Символьная строка, представленная в текстовом поле, обозначает путь от верхнего узла дерева к выбранному узлу.

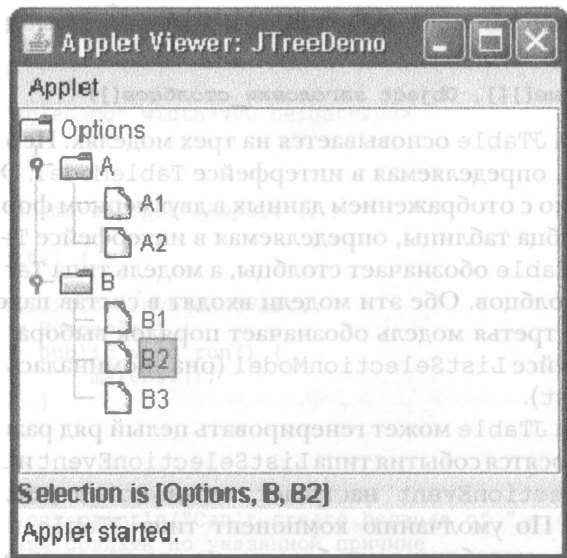


Рис. 32.11. Дерево, развернутое в окне апплета JTreeDemo

Класс JTable

Класс `JTable` представляет компонент, отображающий данные в виде строк и столбцов таблицы. Чтобы изменить размеры столбцов, достаточно перетащить их границы мышью. Кроме того, весь столбец можно перетащить в другое место. В зависимости от конфигурации можно выбрать символьную строку, столбец или ячейку в таблице, а также изменить данные в ячейке. Компонент типа `JTable` довольно сложный, он предоставляет немало вариантов выбора и средств, рассмотреть которые полностью просто невозможно в одном разделе. (Это едва ли не самый сложный компонент Swing.) Но в своей стандартной конфигурации компонент типа `JTable` предоставляет простые в употреблении средства, которых должно быть достаточно для представления данных в табличном виде. Приведенный ниже краткий обзор позволяет составить ясное представление о возможностях этого сложного компонента.

Как и с компонентом типа `JTree`, с компонентом типа `JTable` связаны многие классы и интерфейсы. Все они входят в состав пакета `javax.swing.table`.

В своей основе компонент типа `JTable` очень прост. Он состоит из одного или нескольких столбцов с данными. Вверху каждого столбца находится заголовок. Кроме описания данных в столбце, заголовок предоставляет механизм, с помощью которого пользователь может изменять размеры столбца или его местоположение в таблице. Компонент типа `JTable` не предоставляет никаких возможностей для прокрутки, поэтому он, как правило, размещается в контейнере типа `JScrollPane`.

В классе `JTable` предоставляется несколько конструкторов. Ниже приведен один из них, где параметр *данные* обозначает двухмерный массив данных, пред-

ставляемых в табличном виде, а параметр *заголовки_столбцов* — одномерный массив, содержащий заголовки столбцов.

`JTable(Object данные[],[], Object заголовки_столбцов[])`

Компонент типа `JTable` основывается на трех моделях. Первой из них является модель таблицы, определяемая в интерфейсе `TableModel`. Эта модель определяет все, что связано с отображением данных в двухмерном формате. А второй является модель столбца таблицы, определяемая в интерфейсе `TableColumnModel`. Компонент типа `JTable` обозначает столбцы, а модель типа `TableColumnModel` — характеристики столбцов. Обе эти модели входят в состав пакета `javax.swing.table`. И наконец, третья модель обозначает порядок выбора элементов и определяется в интерфейсе `ListSelectionModel` (она упоминалась выше при рассмотрении класса `JList`).

Компонент типа `JTable` может генерировать целый ряд разных событий. К самым основным относятся события типа `ListSelectionEvent` и `TableModelEvent`. Событие `ListSelectionEvent` наступает, когда пользователь выбирает что-нибудь в таблице. По умолчанию компонент типа `JTable` позволяет выбрать полностью одну строку таблицы или больше, но это поведение можно изменить, чтобы пользователь мог выбрать один или несколько столбцов или одну или несколько отдельных ячеек таблицы. Событие типа `TableModelEvent` наступает, когда данные каким-нибудь образом изменяются в таблице. Обработка таких событий требует больших затрат труда, чем обработка событий в описанных ранее компонентах, а ее рассмотрение выходит за рамки данной книги. Но если компонент типа `JTable` требуется лишь для отображения данных в табличном виде, как в приведенном ниже примере, то никаких событий обрабатывать не придется.

Чтобы создать простой компонент типа `JTable` для отображения данных в табличном виде, достаточно выполнить следующие действия.

1. Создать экземпляр класса `JTable`.
2. Создать экземпляр класса `JScrollPane`, определив таблицу в качестве прокручиваемого объекта.
3. Ввести таблицу на панели с полосами прокрутки.
4. Ввести панель с полосами прокрутки на панели содержимого.

В приведенном ниже примере апплета демонстрируется создание и применение простой таблицы. В данном апплете сначала создается одномерный массив символьных строк `colHeads` для заголовков столбцов, а также двухмерный массив символьных строк `data` для данных в ячейках таблицы. Каждый элемент массива `data` является массивом из трех символьных строк. Эти массивы передаются конструктору класса `JTable`. Таблица вводится на панели с полосами прокрутки, а та — на панели содержимого. В таблице отображаются данные из массива `data`. Стандартная конфигурация таблицы позволяет также редактировать содержимое ячеек. Все вносимые в них изменения отражаются на содержимом базового массива `data`. На рис. 32.12 показана таблица, отображающая данные в окне апплета из данного примера.

```
// Продемонстрировать применение компонента JTable
import java.awt.*;
import javax.swing.*;
/*
  <applet code="JTableDemo" width=400 height=200>
  </applet>
*/

public class JTableDemo extends JApplet {

    public void init() {
        try {
            SwingUtilities.invokeLaterAndWait(
                new Runnable() {
                    public void run() {
                        makeGUI();
                    }
                }
            );
        } catch (Exception exc) {
            System.out.println("Can't create because of " + exc);
            // Нельзя создать по указанной причине
        }
    }

    private void makeGUI() {

        // инициализировать заголовки столбцов
        String[] colHeads = { "Name", "Extension", "ID#" };
        // Имя, добавочный номер телефона, идентификационный номер

        // инициализировать данные
        Object[][] data = {
            { "Gail", "4567", "865" },
            { "Ken", "7566", "555" },
            { "Viviane", "5634", "587" },
            { "Melanie", "7345", "922" },
            { "Anne", "1237", "333" },
            { "John", "5656", "314" },
            { "Matt", "5672", "217" },
            { "Claire", "6741", "444" },
            { "Erwin", "9023", "519" },
            { "Ellen", "1134", "532" },
            { "Jennifer", "5689", "112" },
            { "Ed", "9030", "133" },
            { "Helen", "6751", "145" }
        };

        // создать таблицу
        JTable table = new JTable(data, colHeads);

        // ввести таблицу на панели с полосами прокрутки
        JScrollPane jsp = new JScrollPane(table);

        // ввести панель с полосами прокрутки на панели содержимого
        add(jsp);
    }
}
```

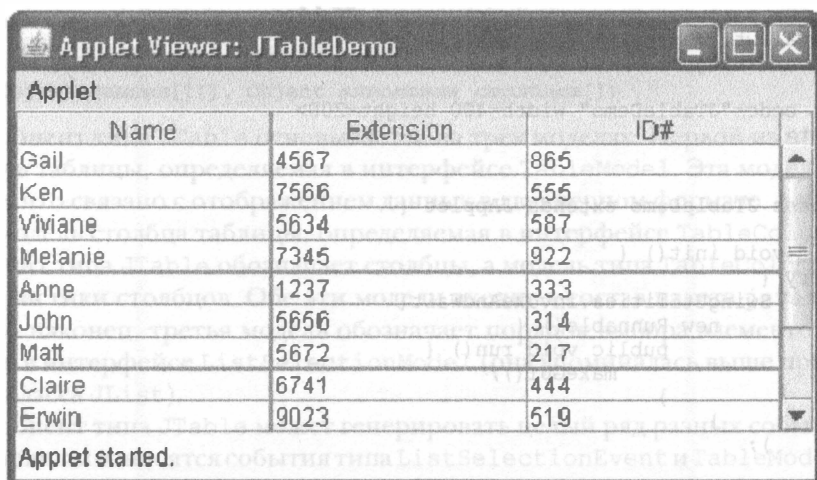


Рис. 32.12. Таблица, отображающая данные в окне апплета JTableDemo

В этой главе представлены меню — один из главных элементов построения ГПИ на основе библиотеки Swing. Меню являются неотъемлемой частью многих приложений, поскольку они представляют пользователю функциональные возможности прикладной программы. В силу своего особого значения меню нашли широкую поддержку в библиотеке Swing. Именно в меню проявляется истинный потенциал библиотеки Swing.

Система меню в Swing поддерживает целый ряд главных элементов меню, включая следующие:

- Строка меню, отображающая главное меню прикладной программы.
- Стандартное меню, которое может содержать выбираемые пункты или другие меню, иначе называемые подменю.
- Всплывающее или контекстное меню, которое обычно активизируется щелчком правой кнопкой мыши.
- Панель инструментов, предоставляющая быстрый доступ к функциональным возможностям прикладной программы (зачастую параллельно с пунктами меню).
- Действие, позволяющее одному объекту управлять разными (двумя или больше) компонентами. Действия обычно употребляются вместе с меню и панелями инструментов.

Меню в Swing поддерживают также оперативные клавиши, позволяющие быстро выбирать пункты, не активизируя само меню, а также мнемонику для выбора пунктов меню нажатием клавиш после отображения самого меню.

Основные положения о меню

Система меню в Swing опирается на ряд связанных с ней классов. В табл. 33.1 перечислены классы, упоминаемые в этой главе и составляющие основу системы меню в Swing. На первый взгляд, меню в Swing могут привести в некоторое замешательство, но на самом деле пользоваться ними очень просто. Эти меню допускают специальную настройку, если таковая требуется, в самых широких пределах, но как правило, классы меню применяются в без изменений, поскольку они поддер-

живают все необходимые варианты построения меню. Например, в меню совсем не трудно ввести изображения и клавиатурные сокращения.

Таблица 33.1. Основные классы меню в Swing

Класс	Описание
JMenuBar	Объект этого класса содержит меню верхнего уровня для приложения
JMenu	Стандартное меню, состоящее из одного или больше пунктов в виде объектов типа JMenuItem
JMenuItem	Объект этого класса представляет пункты, наполняющие меню
JCheckBoxMenuItem	Отмечаемый флажком пункт меню
JRadioButtonMenuItem	Отмечаемый кнопкой-переключателем пункт меню
JSeparator	Визуальный разделитель пунктов меню
JPopupMenu	Меню, которое обычно активизируется щелчком правой кнопкой мыши

Сделаем краткий обзор совместного применения классов, перечисленных в табл. 33.1. Чтобы построить меню верхнего уровня для приложения, сначала следует создать объект класса **JMenuBar**. Проще говоря, этот класс служит контейнером для меню. В экземпляр класса **JMenuBar** обычно вводятся экземпляры класса **JMenu**, причем каждый объект типа **JMenu** определяет отдельное меню. Это означает, что каждый объект типа **JMenu** содержит один или больше выбираемых пунктов. А пункты, отображаемые объектами типа **JMenu**, в свою очередь, являются объектами класса **JMenuItem**. Следовательно, класс **JMenuItem** определяет пункт меню, выбираемый пользователем.

Помимо меню, раскрывающихся из строки меню, можно создавать автономные всплывающие меню. С этой целью следует сначала создать объект типа **JPopupMenu**, а затем ввести в него объекты типа **JMenuItem** в виде пунктов меню. Как правило, всплывающее (или контекстное) меню активизируется щелчком правой кнопкой мыши, когда курсор находится на том компоненте, для которого определено всплывающее меню.

В меню можно вводить не только стандартные пункты, но и пункты, отмечаемые флажками или кнопками-переключателями. В частности, отмечаемый флажком пункт меню создается средствами класса **JCheckBoxMenuItem**, а отмечаемый кнопкой-переключателем пункт меню — средствами класса **JRadioButtonMenuItem**. Оба эти класса расширяют класс **JMenuItem**. Их можно применять в стандартных и всплывающих меню.

Класс **JToolBar** позволяет создать панель инструментов в виде автономного компонента, связанного с меню. Такой компонент нередко служит для быстрого доступа к функциональным возможностям, имеющимся в меню приложения. Например, панель инструментов может предоставлять быстрый доступ к командам форматирования, которые поддерживаются в текстовом редакторе. А служебный класс **JSeparator** позволяет создать линию, разделяющую пункты меню.

В отношении меню Swing следует иметь в виду, что каждый пункт меню представлен объектом класса, расширяющего класс **AbstractButton**. Напомним, что

класс `AbstractButton` служит суперклассом для всех компонентов кнопок в библиотеке Swing, в том числе и компонента типа `JButton`. Следовательно, все пункты меню, по существу, являются кнопками. Очевидно, что пункты не похожи внешне на кнопки, когда выбираются из меню, но они действуют, главным образом, как кнопки. Так, если выбрать пункт меню, то событие действия будет сгенерировано таким же образом, как и при нажатии экранной кнопки.

Следует также иметь в виду, что класс `JMenuItem` служит суперклассом для класса `JMenu`. Это дает возможность создавать подменю, т.е. одни меню, вложенные в другие. Чтобы построить подменю, следует сначала создать меню в виде объекта типа `JMenu` и наполнить его пунктами, а затем ввести его в другой объект типа `JMenu`. Этот процесс демонстрируется в следующем разделе.

Как упоминалось выше, при выборе пункта меню генерируется событие действия. Символьная строка с командой действия, связанная с этим событием действия, по умолчанию обозначает наименование выбранного пункта меню. Следовательно, проанализировав команду действия, можно выяснить, какой именно пункт меню был выбран. Безусловно, для обработки событий действия от каждого пункта меню можно воспользоваться отдельными анонимными классами или лямбда-выражениями. В этом случае выбранный пункт меню уже известен, и поэтому нет нужды анализировать команду действия, чтобы выяснить, какой именно пункт меню был выбран.

Меню могут генерировать и другие типы событий. Например, всякий раз, когда активизируется или выбирается меню или же отменяется его выбор, генерируется событие типа `MenuEvent`, которое может отслеживаться приемником событий из интерфейса `MenuListener`. К числу других событий, связанных с меню, относятся события типа `MenuKeyEvent`, `MenuDragMouseEvent` и `PopupMenuEvent`. Но как правило, отслеживать требуется только события действия, и поэтому в этой главе рассматривается обработка только этих событий.

Краткий обзор классов `JMenuBar`, `JMenu` и `JMenuItem`

Прежде чем строить меню, нужно иметь хотя бы какое-то представление о трех основных классах меню: `JMenuBar`, `JMenu` и `JMenuItem`. Эти классы составляют тот минимум средств, которые требуются для построения главного меню приложения. Кроме того, классы `JMenu` и `JMenuItem` служат для построения всплывающих меню. Таким образом, эти классы образуют основание системы меню в Swing.

Класс `JMenuBar`

Как упоминалось выше, класс `JMenuBar`, по существу, служит контейнером для меню. Как и все остальные классы компонентов, он наследует от класса `JComponent`, а тот — от классов `Container` и `Component`. У этого класса имеется единственный конструктор по умолчанию. Следовательно, строка меню первоначально будет пустой, и поэтому ее придется наполнить меню, прежде чем ею воспользоваться. У каждого приложения имеется одна и только одна строка меню.

В классе `JMenuBar` определяется несколько методов, но на практике применяется только метод `add()`. Этот метод вводит меню в виде объекта типа `JMenu` в строку меню. Ниже приведена его общая форма.

`JMenu add(JMenu меню)`

Здесь параметр *меню* обозначает экземпляр класса `JMenu`, вводимый в строку меню. Меню располагаются в строке меню слева направо в том порядке, в каком они вводятся. Если же меню требуется ввести в конкретном месте, то с этой целью следует воспользоваться приведенным ниже вариантом метода `add()`, наследуемого из класса `Container`.

`Component add(Component меню, int индекс)`

Здесь параметр *меню* обозначает конкретное меню, вводимое по указанному индексу. Индексирование начинается с нуля, причем нулевой индекс обозначает крайнее слева меню в строке меню.

Иногда из строки меню требуется удалить меню, которое больше не нужно. Для этого достаточно вызвать метод `remove()`, наследуемый из класса `Container`. У этого метода имеются следующие общие формы:

**`void remove(Component меню)`
`void remove(int индекс)`**

Здесь параметр *меню* обозначает ссылку на удаляемое меню, а параметр *индекс* — указанный индекс удаляемого меню. Индексация меню начинается с нуля.

Иногда полезным оказывается и метод `getMenuCount()`. Ниже приведена его общая форма. Этот метод возвращает количество элементов, содержащихся в строке меню.

`int getMenuCount()`

В классе `JMenuBar` определяются и другие методы, которым можно найти специальное применение. Например, вызвав метод `getSubElements()`, можно получить массив ссылок на меню, находящихся в строке меню, а вызвав метод `isSelected()`, — определить, выбрано ли конкретное меню.

Как только строка меню будет создана и наполнена отдельными меню, ее можно ввести в контейнер типа `JFrame`, вызвав метод `setJMenuBar()` для экземпляра класса `JFrame`. (Строки меню *не* вводятся на панели содержимого.) Ниже приведена общая форма метода `setJMenuBar()`.

`void setJMenuBar(JMenuBar строка_меню)`

Здесь *строка_меню* обозначает ссылку на строку меню. Строка меню отображается на позиции, определяемой стилем оформления ГПИ. Как правило, оно располагается в вдоль верхнего края окна приложения.

Класс `JMenu`

Класс `JMenu` инкапсулирует меню, наполняемое пунктами в виде объектов типа `JMenuItem`. Как упоминалось ранее, этот класс наследует от класса `JMenuItem`. Это означает, что одно меню типа `JMenu` можно выбирать из другого. Следовательно, одно

меню может служить подменю для другого. В классе `JMenu` определяется целый ряд конструкторов. Здесь и далее в этой главе применяется следующий конструктор:

```
JMenu(String имя)
```

Этот конструктор создает меню с заголовком, обозначаемым параметром *имя*. Разумеется, присваивать наименование меню совсем не обязательно. Для создания безымянного меню можно воспользоваться следующим конструктором по умолчанию:

```
JMenu()
```

В классе `JMenu` поддерживаются и другие конструкторы. Но в любом случае создаваемое с их помощью меню будет пустым до тех пор, пока в него не будут введены отдельные пункты.

В классе `JMenu` определяется немало методов. Здесь вкратце описываются лишь наиболее употребительные из них. Для ввода пункта в меню служит метод `add()`, у которого имеется несколько общих форм. Ниже приведены две его общие формы.

```
JMenuItem add(JMenuItem пункт)  
JMenuItem add(Component пункт, int индекс)
```

Здесь параметр *пункт* обозначает пункт, вводимый в меню. В первой форме пункт вводится в конце меню, а во второй форме — по указанному *индексу*. Как и следовало ожидать, индексирование пунктов меню начинается с нуля. В обеих рассматриваемых здесь формах метод `add()` возвращает ссылку на вводимый пункт меню. Любопытно, что для ввода пунктов в меню можно также воспользоваться методом `insert()`.

В меню можно ввести разделитель (объект типа `JSeparator`), вызвав метод `addSeparator()`, как показано ниже. Разделитель вводится в конце меню.

```
void addSeparator()
```

А для ввода разделителя в нужном месте меню можно вызвать метод `insertSeparator()`. Ниже приведена его общая форма, где параметр *индекс* обозначает отсчитываемый от нуля индекс, по которому вводится разделитель.

```
void insertSeparator(int индекс)
```

Вызвав метод `remove()`, можно удалить пункт из меню. Ниже приведены две его общие формы, где параметр *меню* обозначает ссылку на удаляемый пункт меню, а параметр *индекс* — указанный индекс удаляемого пункта меню.

```
void remove(JMenuItem меню)  
void remove(int индекс)
```

Вызвав метод `getMenuComponentCount()`, можно получить количество пунктов в меню:

```
int getMenuComponentCount()
```

А если вызвать метод `getMenuComponents()`, то можно получить массив из пунктов меню, как показано ниже. Этот метод возвращает массив, содержащий компоненты меню.

```
Component[] getMenuComponents()
```

Класс JMenuItem

Класс JMenuItem инкапсулирует пункт меню. Выбор этого элемента ГПИ может быть связан с некоторым действием прикладной программы, например, с сохранением данных или закрытием файла, или же с отображением подменю. Как упоминалось ранее, класс JMenuItem является производным от класса AbstractButton, и поэтому каждый пункт меню можно рассматривать как особого рода кнопку. Следовательно, когда выбирается пункт меню, событие действия генерируется таким же образом, как и при нажатии экранной кнопки типа JButton. В классе JMenuItem определено немало конструкторов. Ниже приведены применяемые здесь и далее конструкторы данного класса.

```
JMenuItem(String имя)
JMenuItem(Icon изображение)
JMenuItem(String имя, Icon изображение)
JMenuItem(String имя, int мнемоника)
JMenuItem(Action действие)
```

Первый конструктор создает пункт меню, обозначаемый параметром *имя*. Второй конструктор создает пункт меню, отображающий указанное *изображение*. Третий конструктор создает пункт меню, обозначаемый параметром *имя* и отображающий указанное *изображение*. Четвертый конструктор создает пункт меню, обозначаемый параметром *имя* и использующий указанную клавиатурную *мнемонику*. Эта *мнемоника* позволяет выбрать пункт меню нажатием указанной клавиши. И последний конструктор создает пункт меню, используя сведения, обозначаемые параметром *действие*. В данном классе поддерживается также конструктор по умолчанию.

Благодаря тому что класс JMenuItem наследует от класса AbstractButton, функциональные возможности последнего доступны для пунктов меню. В частности, для меню нередко оказывается полезным метод `setEnabled()`, позволяющий включать или отключать отдельные пункты меню. Ниже приведена общая форма этого метода.

```
void setEnabled(boolean включить)
```

Если параметр *включить* принимает логическое значение `true`, то пункт меню включается. А если этот параметр принимает логическое значение `false`, то пункт меню отключается и не может быть выбран.

Создание главного меню

По традиции наиболее употребительным считается *главное меню*. Оно доступно из строки меню и определяет все (или почти все) функциональные возможности приложения. К счастью, библиотека Swing значительно упрощает создание главного меню и управление им. Ниже будет показано, как создать элементарное главное меню и ввести в него отдельные варианты выбора.

Процесс создания главного меню состоит из нескольких стадий. Сначала создается объект типа `JMenuBar`, который будет содержать отдельные меню. Затем соз-

дается каждое меню, располагаемое в строке меню. В общем, построение меню начинается с создания объекта типа `JMenu`, в который затем вводятся пункты меню в виде объектов типа `JMenuItem`. Как только отдельные меню будут созданы, они вводятся в строку меню. А саму строку меню следует ввести во фрейм, вызвав метод `setJMenuBar()`. И наконец, каждый пункт меню должен быть дополнен приемником действий, обрабатывающим событие действия, наступающее при выборе отдельного пункта из меню.

Процесс создания меню и управления ими станет понятнее, если продемонстрировать его на конкретном примере. Ниже приведена программа, в которой создается простая строка меню, состоящая из трех меню. Первым из них является меню **File** (Файл), состоящее из выбираемых пунктов **Open** (Открыть), **Close** (Заккрыть), **Save** (Сохранить) и **Exit** (Выход). Второе меню называется **Options** (Параметры) и состоит из двух подменю: **Colors** (Цвета) и **Priority** (Приоритет). Третье меню называется **Help** (Справка) и состоит из единственного пункта **About** (О программе). Когда выбирается пункт меню, его наименование отображается в метке на панели содержимого. Вид главного меню в окне программы из данного примера приведен на рис. 33.1.

// Продемонстрировать простое главное меню

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class MenuDemo implements ActionListener {

    JLabel jlab;

    MenuDemo() {
        // создать новый контейнер типа JFrame
        JFrame jfrm = new JFrame("Menu Demo"); // Демонстрация меню

        // указать диспетчер поточной компоновки типа FlowLayout
        jfrm.setLayout(new FlowLayout());

        // задать исходные размеры фрейма
        jfrm.setSize(220, 200);

        // завершить прикладную программу, когда пользователь
        // закроет ее окно
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // создать метку для отображения результатов выбора из меню
        jlab = new JLabel();

        // создать строку меню
        JMenuBar jmb = new JMenuBar();

        // создать меню File
        JMenu jmFile = new JMenu("File"); // Файл
        JMenuItem jmiOpen = new JMenuItem("Open"); // Открыть
        JMenuItem jmiClose = new JMenuItem("Close"); // Заккрыть
        JMenuItem jmiSave = new JMenuItem("Save"); // Сохранить
        JMenuItem jmiExit = new JMenuItem("Exit"); // Выход
        jmFile.add(jmiOpen);
        jmFile.add(jmiClose);
```

```

jmFile.add(jmiSave);
jmFile.addSeparator();
jmFile.add(jmiExit);
jmb.add(jmFile);

// создать меню Options
JMenu jmOptions = new JMenu("Options"); // Параметры

// создать подменю Colors
JMenu jmColors = new JMenu("Colors"); // Цвета
JMenuItem jmiRed = new JMenuItem("Red"); // Красный
JMenuItem jmiGreen = new JMenuItem("Green"); // Зеленый
JMenuItem jmiBlue = new JMenuItem("Blue"); // Синий
jmColors.add(jmiRed);
jmColors.add(jmiGreen);
jmColors.add(jmiBlue);
jmOptions.add(jmColors);

// создать подменю Priority
JMenu jmPriority = new JMenu("Priority"); // Приоритет
JMenuItem jmiHigh = new JMenuItem("High"); // Высокий
JMenuItem jmiLow = new JMenuItem("Low"); // Низкий
jmPriority.add(jmiHigh);
jmPriority.add(jmiLow);
jmOptions.add(jmPriority);

// создать пункт меню Reset
JMenuItem jmiReset = new JMenuItem("Reset"); // Сбросить
jmOptions.addSeparator();
jmOptions.add(jmiReset);

// И наконец, ввести все выбираемые меню в строку меню
jmb.add(jmOptions);

// создать меню Help
JMenu jmHelp = new JMenu("Help"); // Справка
JMenuItem jmiAbout = new JMenuItem("About"); // О программе
jmHelp.add(jmiAbout);
jmb.add(jmHelp);

// ввести приемники действий от пунктов меню
jmiOpen.addActionListener(this);
jmiClose.addActionListener(this);
jmiSave.addActionListener(this);
jmiExit.addActionListener(this);
jmiRed.addActionListener(this);
jmiGreen.addActionListener(this);
jmiBlue.addActionListener(this);
jmiHigh.addActionListener(this);
jmiLow.addActionListener(this);
jmiReset.addActionListener(this);
jmiAbout.addActionListener(this);

// ввести метку на панели содержимого
jfrm.add(jlab);

// ввести строку меню во фрейм
jfrm.setJMenuBar(jmb);

// отобразить фрейм
jfrm.setVisible(true);

```

```

}

// обработать события действия от пунктов меню
public void actionPerformed(ActionEvent ae) {
    // получить команду действия из выбранного меню
    String comStr = ae.getActionCommand();

    // выйти из программы, если пользователь выберет пункт меню Exit
    if(comStr.equals("Exit")) System.exit(0);

    // отобразить в противном случае результат выбора из меню
    jlab.setText(comStr + " Selected"); // Выбрано указанное
}

public static void main(String args[]) {
    // создать фрейм в потоке диспетчеризации событий
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new MenuDemo();
        }
    });
}
}

```

Рассмотрим подробнее, каким образом в данной программе создаются меню, начиная с конструктора класса MenuDemo. Сначала в этом конструкторе создается фрейм типа JFrame, устанавливается диспетчер компоновки, задаются размеры фрейма и стандартная операция закрытия прикладной программы. (Все эти операции подробно описаны в главе 31.) Затем создается метка типа JLabel, предназначенная для отображения результатов выбора из меню. Далее создается строка меню, а ссылка на нее присваивается переменной jmb, как показано ниже.

```

// создать строку меню
JMenuBar jmb = new JMenuBar();

```

Затем создается меню File и его пункты, а ссылка на него присваивается переменной jmFile:

```

// создать меню File
JMenu jmFile = new JMenu("File"); // Файл
JMenuItem jmiOpen = new JMenuItem("Open"); // Открыть
JMenuItem jmiClose = new JMenuItem("Close"); // Закрыть
JMenuItem jmiSave = new JMenuItem("Save"); // Сохранить
JMenuItem jmiExit = new JMenuItem("Exit"); // Выход

```

Имена отдельных пунктов Open, Close, Save и Exit меню File будут отображаться на месте метки при их выборе. Далее эти пункты вводятся в меню File следующим образом:

```

jmFile.add(jmiOpen);
jmFile.add(jmiClose);
jmFile.add(jmiSave);
jmFile.addSeparator();
jmFile.add(jmiExit);

```

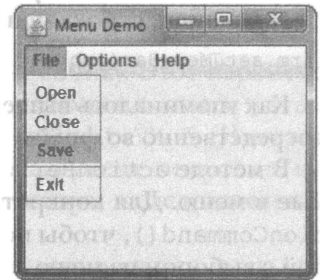


Рис. 33.1. Вид главного меню в окне программы MenuDemo

И наконец, меню **File** вводится в строку меню, как показано ниже.

```
jmb.add(jmFile);
```

По завершении приведенной выше последовательности кода строка меню будет содержать один элемент: меню **File**, тогда как само меню **File** — пункты **Open**, **Close**, **Save** и **Exit**. (Обратите внимание на разделитель, введенный перед пунктом меню **Exit**, — он визуально отделяет пункт **Exit** от остальных пунктов меню **File**.)

Аналогичным образом создается и меню **Options**, но оно состоит из двух подменю, **Colors** и **Priority**, и одного пункта **Reset**. Сначала подменю создаются по отдельности, а затем вводятся в строку меню. И последним в данное меню вводится пункт **Reset**. После этого меню **Options** вводится в строку меню. Подобным же образом создается и меню **Help**.

Следует заметить, что класс `MenuDemo` реализует интерфейс `ActionListener`, а события действия, генерируемые при выборе меню, обрабатываются методом `actionPerformed()`, определяемым в классе `MenuDemo`. Следовательно, для пунктов меню в данной программе вводятся приемники действий по ссылке `this`. В то же время приемники действий не вводятся для пунктов меню **Colors** и **Priority**, поскольку они не выбираются, а просто активизируют подменю.

И наконец, строка меню вводится во фрейм следующим образом:

```
jfrm.setJMenuBar(jmb);
```

Как упоминалось выше, строки меню вводятся не на панели содержимого, а непосредственно во фрейм типа `JFrame`.

В методе `actionPerformed()` обрабатываются события действия, генерируемые в меню. Для конкретного события в этом методе вызывается метод `getActionCommand()`, чтобы получить символьную строку с командой действия, связанной с выбором из меню. Ссылка на эту символьную строку сохраняется в переменной `comStr`. Затем эта строка проверяется на наличие команды действия `"Exit"`:

```
if(comStr.equals("Exit")) System.exit(0);
```

Если символьная строка содержит команду действия `"Exit"`, то программа завершается вызовом метода `System.exit()`. Этот метод немедленно завершает программу и передает свой аргумент в качестве кода состояния вызывающему процессу, которым обычно является операционная система или браузер. Условно нулевой код состояния означает нормальное завершение программы, а любой другой код состояния — ненормальное ее завершение. Результат выбора всех остальных пунктов меню, кроме **Exit**, отображается на месте заданной метки.

Можете поэкспериментировать с программой `MenuDemo`, попробовав ввести еще одно меню или дополнительные пункты в уже имеющиеся меню. Ваша главная задача — уяснить основные принципы создания меню, поскольку данная программа будет постепенно усложняться по ходу изложения материала этой главы.

Ввод мнемоники и оперативных клавиш в меню

Меню, созданное в предыдущем примере, вполне работоспособно, но его можно усовершенствовать. Меню реальных прикладных программ обычно поддержи-

вают клавиатурные сокращения, предоставляя опытным пользователям возможность быстро выбирать пункты меню. Клавиатурные сокращения принимают две формы: мнемонику и оперативные клавиши. Применительно к меню мнемоника определяет клавишу, нажатием которой выбирается отдельный пункт активного меню. Следовательно, *мнемоника* позволяет выбирать с клавиатуры пункты того меню, которое уже отображается, а *оперативная клавиша* — сделать то же самое, не активизируя предварительно меню.

Мнемонику можно указать как для объектов типа JMenuItem (пунктов меню), так и для объектов типа JMenu (самих меню). Мнемоника для объекта типа JMenuItem задается двумя способами. Во-первых, ее можно указать при создании данного объекта с помощью следующего конструктора:

```
JMenuItem(String имя, int мнемоника)
```

В данном случае наименование пункта меню передается в качестве параметра *имя*, а мнемоника — в качестве параметра *мнемоника*. И во-вторых, мнемонику для пунктов меню (объектов типа JMenuItem) или самого меню (объекта типа JMenu) можно задать, вызвав метод setMnemonic(). Этот метод наследуется обоими классам, JMenuItem и JMenu из класса AbstractButton. Ниже приведена его общая форма.

```
void setMnemonic(int мнемоника)
```

Здесь параметр *мнемоника* обозначает задаваемую мнемонику. Этот параметр должен принимать значение одной из констант, определяемых в классе java.awt.event.KeyEvent, в том числе KeyEvent.VK_F или KeyEvent.VK_Z. (Имеется и другой вариант метода setMnemonic(), принимающий аргумент типа char, но он считается устаревшим.) Мнемоника не зависит от регистра, а следовательно, мнемоники VK_A и VK_a равнозначны.

По умолчанию подчеркивается первая буква в наименовании пункта меню, совпадающая с заданной мнемоникой. Если же требуется подчеркнуть другую букву, следует указать ее индекс в качестве аргумента метода setDisplayedMnemonicIndex(), наследуемого классами JMenuItem и JMenu из класса AbstractButton. Ниже приведена его общая форма, где параметр *индекс* обозначает указанный индекс подчеркиваемой буквы.

```
void setDisplayedMnemonicIndex(int индекс)
```

Оперативная клавиша может быть связана с объектом типа JMenuItem. Она задается с помощью метода setAccelerator(), как показано ниже.

```
void setAccelerator(KeyStroke сочетание_клавиш)
```

Здесь параметр *сочетание_клавиш* обозначает комбинацию клавиш, нажимаемых для выбора пункта меню. Класс KeyStroke содержит ряд фабричных методов для построения разнотипных комбинаций клавиш быстрого выбора пунктов меню. Ниже приведены три общие формы одного из таких методов.

```
static KeyStroke getKeyStroke(char символ)  
static KeyStroke getKeyStroke(Character символ, int модификатор)  
static KeyStroke getKeyStroke(int символ, int модификатор)
```

Здесь параметр *символ* обозначает конкретный символ оперативной клавиши. В первой форме данного метода этот символ указывается в виде значения типа

Ввод изображений и всплывающих подсказок в пункты меню

Изображения можно вводить в пункты меню или употреблять их вместо текста. Самый простой способ ввести изображение в пункт меню — указать его при создании этого пункта меню с помощью одного из следующих конструкторов:

```
JMenuItem(Icon изображение)
JMenuItem(String имя, Icon изображение)
```

Первый конструктор создает пункт меню, отображающий указанное *изображение*; второй конструктор — пункт меню, обозначаемый параметром *имя* и отображающий указанное *изображение*. Например, в приведенном ниже фрагменте кода создается пункт меню **About** вместе с указанным изображением.

```
ImageIcon icon = new ImageIcon("AboutIcon.gif");
JMenuItem jmiAbout = new JMenuItem("About", icon);
```

Если ввести этот фрагмент кода в программу Menu-Demo, то рядом с текстом наименования пункта меню **About** появится значок, указанный в переменной *icon* (рис. 33.3). Этот значок можно ввести и после создания пункта меню, вызвав метод `setIcon()`, наследуемый из класса `AbstractButton`. А вызвав метод `setHorizontalTextPosition()`, можно установить выравнивание изображения по горизонтали относительно текста наименования пункта меню.

Вызвав метод `setDisabledIcon()`, можно также указать неактивный значок, который отображается вместе с недоступным пунктом меню. Как правило, такой значок отображается светло-серым цветом, когда пункт меню недоступен. Если неактивный значок указан, то он отображается при условии, что пункт меню недоступен.

Всплывающая подсказка — это небольшое сообщение, кратко описывающее пункт меню. Такая подсказка всплывает автоматически, если навести курсор на пункт меню и оставить его на мгновение в таком положении. Чтобы ввести всплывающую подсказку в пункт меню, достаточно вызвать метод `setToolTipText()` для этого пункта меню, указав текст, который требуется вывести во всплывающей подсказке. Ниже приведена общая форма этого метода.

```
void setToolTipText(String сообщение)
```

В данном случае параметр *сообщение* обозначает символьную строку, которая отображается, когда появляется всплывающая подсказка. Например, в следующей строке кода создается всплывающая подсказка для пункта меню **About**:

```
jmiAbout.setToolTipText("Info about the MenuDemo program.");
```

Любопытно, что метод `setToolTipText()` наследуется классом `JMenuItem` из класса `JComponent`. Это означает, что всплывающую подсказку можно ввести и в другие типы компонентов, в том числе и экранные кнопки. Попробуйте сделать это в качестве упражнения.

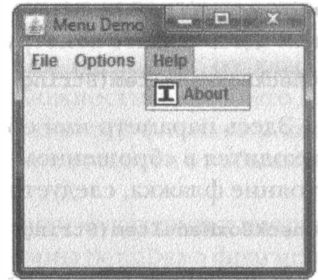


Рис. 33.3. Вид пункта меню **About** в окне программы MenuDemo после ввода значка

Классы `JRadioButtonMenuItem` и `JCheckBoxMenuItem`

В приведенных выше примерах демонстрировались наиболее употребительные типы пунктов меню, но в Swing определяются также два других типа пункта меню: с флажками и кнопками-переключателями. Эти отмечаемые пункты меню упрощают построение ГПИ, делая доступными из меню такие функциональные возможности, для предоставления которых в противном случае потребовались бы дополнительные автономные компоненты. Кроме того, наличие флажков и кнопок-переключателей в меню иногда кажется вполне естественным для выбора конкретного ряда средств. Но независимо от конкретных причин для применения флажков и кнопок-переключателей в меню библиотека Swing позволяет сделать это довольно просто, как поясняется ниже.

Чтобы ввести флажок в меню, следует создать объект класса `JCheckBoxMenuItem`. В этом классе определяется несколько конструкторов. Ниже приведен применяемый здесь и далее конструктор данного класса.

```
JCheckBoxMenuItem(String имя)
```

Здесь параметр *имя* обозначает наименование пункта меню. Исходно флажок находится в сброшенном состоянии. Если же требуется явно указать исходное состояние флажка, следует воспользоваться конструктором

```
JCheckBoxMenuItem(String имя, boolean состояние)
```

Если параметр *состояние* принимает логическое значение `true`, то флажок исходно установлен, а иначе — сброшен. В классе `JCheckBoxMenuItem` предоставляются также конструкторы, позволяющие снабдить значком пункт меню с флажком. Ниже приведен один из таких конструкторов.

```
JCheckBoxMenuItem(String имя, Icon значок)
```

В данном случае параметр *имя* обозначает наименование пункта меню, а параметр *значок* — связанное с ним изображение. Исходно флажок в данном пункте меню сброшен. В классе `JCheckBoxMenuItem` предоставляются и другие конструкторы.

Флажки в меню действуют аналогично автономным флажкам. В частности, они генерируют события действия и события от элементов при изменении их состояния. Флажки особенно полезны в тех меню, где требуется отобразить состояние выбранных или невыбранных пунктов.

Чтобы ввести кнопку-переключатель в пункт меню, следует создать объект класса `JRadioButtonMenuItem`. Этот класс наследует от класса `JMenuItem` и предоставляет богатый набор конструкторов. Ниже приведены конструкторы, применяемые здесь и далее в этой главе.

```
JRadioButtonMenuItem(String имя)
```

```
JRadioButtonMenuItem(String имя, boolean состояние)
```

Первый конструктор создает пункт меню с невыбранной кнопкой-переключателем и наименованием, определяемым параметром *имя*. А второй конструктор

позволяет дополнительно указать исходное *состояние* кнопки-переключателя. Если параметр *состояние* принимает логическое значение true, то кнопка-переключатель исходно выбрана, а иначе — не выбрана. Другие конструкторы данного класса позволяют дополнительно указать изображение для пункта меню с кнопкой-переключателем. Ниже приведен пример одного из таких конструкторов.

JRadioButtonMenuItem(String *имя*, Icon *значок*, boolean *состояние*)

Этот конструктор создает пункт меню с кнопкой-переключателем, определяемый параметром *имя* и отображающий указанный *значок*. Если параметр *состояние* принимает логическое значение true, то кнопка-переключатель исходно выбрана, а иначе — не выбрана. В классе JRadioButtonMenuItem поддерживается и ряд других конструкторов.

Пункт меню типа JRadioButtonMenuItem действует аналогично автономной кнопке-переключателю, генерируя событие от элемента и событие действия. Как и автономные кнопки-переключатели, в меню кнопки-переключатели должны быть объединены в группу, чтобы они действовали в режиме взаимоисключающего выбора.

Классы JCheckBoxMenuItem и JRadioButtonMenuItem наследуют от класса JRadioButtonMenuItem, и поэтому функциональные возможности последнего доступны каждому из них. Помимо дополнительных возможностей флажков и кнопок-переключателей, отмечаемые пункты меню ничем не отличаются от обычных пунктов меню.

Чтобы опробовать на практике пункты меню, отмечаемые флажками и кнопками-переключателями, удалите сначала из примера программы MenuDemo фрагмент кода, в котором создается меню Options. Затем подставьте приведенный ниже фрагмент кода, в котором создаются подменю Colors и Priority с флажками и кнопками-переключателями соответственно. После этой замены меню Options будет выглядеть так, как показано на рис. 33.4.

```
// создать меню Options
JMenu jmOptions = new JMenu("Options");

// создать подменю Colors
JMenu jmColors = new JMenu("Colors");

// использовать флажки, чтобы пользователь мог выбрать
// сразу несколько цветов
JCheckBoxMenuItem jmiRed = new JCheckBoxMenuItem("Red");
JCheckBoxMenuItem jmiGreen = new JCheckBoxMenuItem("Green");
JCheckBoxMenuItem jmiBlue = new JCheckBoxMenuItem("Blue");

jmColors.add(jmiRed);
jmColors.add(jmiGreen);
jmColors.add(jmiBlue);
jmOptions.add(jmColors);

// создать подменю Priority
JMenu jmPriority = new JMenu("Priority");

// Использовать кнопки-переключатели для установки приоритета.
// Благодаря этому в меню не только отображается установленный
// приоритет, но и гарантируется установка одного и только
```

```
// одного приоритета. Исходно выбирается кнопка-переключатель
// в пункте меню High
JRadioButtonMenuItem jmiHigh =
    new JRadioButtonMenuItem("High", true);
JRadioButtonMenuItem jmiLow =
    new JRadioButtonMenuItem("Low");

jmPriority.add(jmiHigh);
jmPriority.add(jmiLow);
jmOptions.add(jmPriority);

// создать группу кнопок-переключателей в пунктах подменю Priority
ButtonGroup bg = new ButtonGroup();
bg.add(jmiHigh);
bg.add(jmiLow);

// создать пункт меню Reset
JMenuItem jmiReset = new JMenuItem("Reset");
jmOptions.addSeparator();
jmOptions.add(jmiReset);

// и наконец, ввести меню Options в строку меню
jmb.add(jmOptions);
```

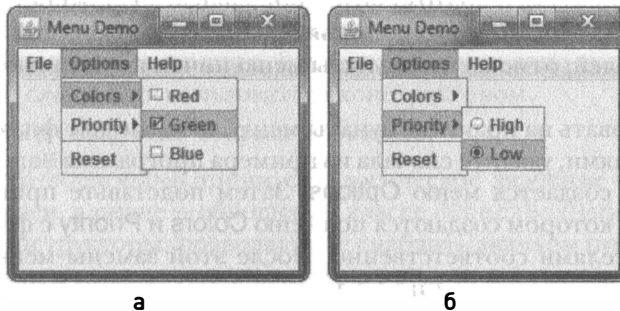


Рис. 33.4. Результат установки флажка (а) и кнопки-переключателя (б) в пунктах меню Options

Создание всплывающего меню

Всплывающее меню служит весьма распространенной альтернативой или дополнением строки меню. Как правило, всплывающее меню активизируется щелчком правой кнопкой мыши на компоненте. Всплывающие меню поддерживаются в классе `JPopupMenu` из библиотеки Swing. У этого класса имеются два конструктора, но здесь и далее в этой главе применяется только следующий конструктор по умолчанию:

```
JPopupMenu ()
```

Этот конструктор создает стандартное всплывающее меню. А другой конструктор позволяет указать заголовок всплывающего меню. Отображение этого заголовка зависит от выбранного стиля оформления ГПИ.

В общем, всплывающие меню создаются таким же образом, как и обычные меню. Сначала создается объект типа `JPopupMenu`, а затем он наполняется пунктами всплывающего меню. Обработка результатов выбора пунктов всплывающего меню выполняется тем же самым способом: приемом событий действия. А отличаются всплывающие меню от обычных главным образом процессом их активизации.

Процесс активизации всплывающего меню разделяется на три стадии.

- Регистрация приемника событий от мыши.
- Отслеживание в приемнике событий момента запуска всплывающего меню.
- Отображение всплывающего меню при получении события запуска всплывающего меню, для чего вызывается метод `show()`.

Рассмотрим каждую из этих стадий подробнее. Всплывающее меню обычно активизируется щелчком правой кнопкой мыши, когда курсор находится на том компоненте, для которого определено всплывающее меню. Следовательно, *запуск всплывающего меню* происходит после щелчка правой кнопкой мыши на компоненте, снабженном всплывающим меню. Для приема события запуска всплывающего меню сначала реализуется интерфейс `MouseListener`, а затем регистрируется приемник подобных событий. С этой целью вызывается метод `addMouseListener()`. Как пояснялось в главе 24, в интерфейсе `MouseListener` определяются приведенные ниже методы.

```
void mouseClicked(MouseEvent событие_от_мышь)
void mouseEntered(MouseEvent событие_от_мышь)
void mouseExited(MouseEvent событие_от_мышь)
void mousePressed(MouseEvent событие_от_мышь)
void mouseReleased(MouseEvent событие_от_мышь)
```

Из всех перечисленных выше методов для создания всплывающих меню особое значение имеют два метода: `mousePressed()` и `mouseReleased()`. В зависимости от выбранного стиля оформления ГПИ любой из этих методов приема событий от мыши может запустить всплывающее меню. Именно поэтому зачастую проще воспользоваться классом `MouseAdapter`, чтобы реализовать интерфейс `MouseListener` и просто переопределить его методы `mousePressed()` и `mouseReleased()`.

В классе `MouseEvent` определяется целый ряд методов, но только четыре из них обычно требуются для активизации всплывающего меню. Эти методы перечислены ниже.

```
int getX()
int getY()
boolean isPopupTrigger()
Component getComponent()
```

Текущие координаты `X`, `Y` положения курсора относительно источника события определяются с помощью методов `getX()` и `getY()` соответственно. Полученные с их помощью координаты служат для указания левого верхнего угла всплывающего меню при его отображении. Метод `isPopupTrigger()` возвращает логическое значение `true`, если событие от мыши обозначает запуск всплывающего меню, а иначе — логическое значение `false`. Чтобы получить ссылку на компонент, сгенерировавший событие от мыши, следует вызвать метод `getComponent()`.

Чтобы отобразить всплывающее меню, следует вызвать метод `show()`, определяемый в классе `JPopupMenu` следующим образом:

```
void show(Component вызывающий, int X, int Y)
```

где параметр *вызывающий* обозначает компонент, относительно которого отображается всплывающее меню; *X* и *Y* — координаты *X, Y* местоположения левого верхнего угла всплывающего меню относительно *вызывающего* компонента. Чтобы получить вызывающий компонент, проще всего вызвать метод `getComponent()` для объекта события, передаваемого обработчику событий от мыши.

Все сказанное выше относительно всплывающих меню можно применить на практике, введя всплывающее меню **Edit** (Правка) в пример программы `MenuDemo`, представленный в начале этой главы. Это меню будет состоять из трех пунктов: **Cut** (Вырезать), **Copy** (Скопировать) и **Paste** (Вставить). Сначала введите в программу `MenuDemo` приведенную ниже строку кода, где переменная `jpu` объявляется для хранения ссылки на всплывающее меню.

```
JPopupMenu jpu;
```

Затем введите следующий фрагмент кода в конструктор класса `MenuDemo`:

```
// создать всплывающее меню Edit
jpu = new JPopupMenu();

// создать пункты всплывающего меню
JMenuItem jmiCut = new JMenuItem("Cut");
JMenuItem jmiCopy = new JMenuItem("Copy");
JMenuItem jmiPaste = new JMenuItem("Paste");

// ввести пункты во всплывающее меню
jpu.add(jmiCut);
jpu.add(jmiCopy);
jpu.add(jmiPaste);

// ввести приемник событий запуска всплывающего меню
jfrm.addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent me) {
        if(me.isPopupTrigger())
            jpu.show(me.getComponent(), me.getX(), me.getY());
    }
    public void mouseReleased(MouseEvent me) {
        if(me.isPopupTrigger())
            jpu.show(me.getComponent(), me.getX(), me.getY());
    }
});
```

Сначала в приведенном выше фрагменте кода создается экземпляр класса `JPopupMenu`, сохраняемый в переменной `jpu`. Затем в нем создаются обычным образом три пункта меню, **Cut**, **Copy** и **Paste**, которыми наполняется объект, хранящийся в переменной `jpu`. На этом создание всплывающего меню **Edit** завершается. Всплывающие меню не вводятся ни в строку меню, ни в какой-нибудь другой объект.

Далее создается анонимный внутренний класс для ввода приемника событий от мыши типа `MouseListener`. Этот класс опирается на класс `MouseAdapter`, а следовательно, в приемнике событий нужно переопределить только те методы, которые имеют отношение ко всплывающему меню, а именно `mousePressed()` и `mouseReleased()`. В классе адаптера предоставляются стандартные реализации

других методов из интерфейса `MouseListener`. Следует также заметить, что приемник событий от мыши вводится во фрейм, хранящийся в переменной `jfrm`. Это означает, что если щелкнуть правой кнопкой мыши в любом месте панели содержимого, то произойдет запуск всплывающего меню.

В методах `mousePressed()` и `mouseReleased()` вызывается метод `isPopupTrigger()`, чтобы выяснить, связано ли наступившее событие с запуском всплывающего меню. Если это действительно так, то вызывается метод `show()`, чтобы отобразить всплывающее меню. С целью получить вызывающий компонент вызывается метод `getComponent()` для события от мыши. В данном случае вызывающим компонентом будет панель содержимого. А для получения координат левого верхнего угла всплывающего меню вызываются методы `getX()` и `getY()`. В итоге левый верхний угол всплывающего меню оказывается прямо под курсором.

И наконец, в данную программу нужно ввести приведенные ниже приемники действий. Они обрабатывают события действия, наступающие в тот момент, когда пользователь выбирает пункт из всплывающего меню.

```
jmiCut.addActionListener(this);
jmiCopy.addActionListener(this);
jmiPaste.addActionListener(this);
```

После внесения описанных выше дополнений в программу `MenuDemo` всплывающее меню можно активизировать щелчком правой кнопкой мыши в любом месте панели содержимого. Полученный результат показан на рис. 33.5.

В отношении рассматриваемого здесь примера всплывающего меню следует также заметить, что компонентом, вызывающим всплывающее меню, в данном случае всегда остается фрейм `jfrm`, и поэтому его можно передать явным образом, не вызывая метод `getComponent()`. С этой целью фрейм `jfrm` нужно присвоить переменной экземпляра класса `MenuDemo`, а не локальной переменной, чтобы сделать его доступным во внутреннем классе. И тогда для отображения всплывающего меню метод `show()` можно вызвать следующим образом:

```
jpu.show(jfrm, me.getX(), me.getY());
```

И хотя такой прием вполне пригоден в данном примере, преимущество вызова метода `getComponent()` заключается в том, что всплывающее меню будет автоматически появляться относительно вызывающего компонента. Следовательно, один и тот же код может быть использован для отображения любого всплывающего меню относительно вызывающего его объекта.

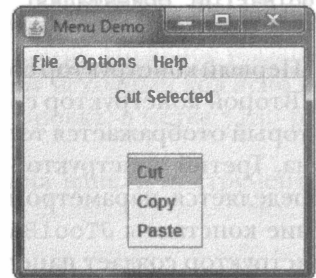


Рис. 33.5. Меню, всплывающее в окне программы `MenuDemo`

Создание панели инструментов

Панель инструментов является компонентом, служащим как альтернативой, так и дополнением меню. Такая панель состоит из ряда кнопок (или других ком-

понентов), предоставляющих пользователю возможность немедленного доступа к различным параметрам и режимам работы программы. Например, панель инструментов может содержать кнопки для выбора параметров шрифтового оформления текста, в том числе полужирного или наклонного начертания, выделения или подчеркивания текста. Эти параметры можно выбрать, не раскрывая меню. Как правило, кнопки на панели инструментов обозначены значками, а не текстовыми надписями, хотя допускается и то и другое. Кроме того, кнопки на панели инструментов нередко снабжаются всплывающим подсказками. Панель инструментов можно размещать вдоль любой стороны рабочего окна, перетаскивая ее в пределах окна и даже за его пределы. В последнем случае она становится свободно плавающей.

В библиотеке Swing панели инструментов являются экземплярами класса `JToolBar`. Конструкторы этого класса позволяют создать панель инструментов как с заголовком, так и без него, а также указать расположение панели инструментов по горизонтали или по вертикали. Ниже перечислены конструкторы класса `JToolBar`.

```
JToolBar()  
JToolBar(String заголовок)  
JToolBar(int ориентация)  
JToolBar(String заголовок, int ориентация)
```

Первый конструктор создает горизонтальную панель инструментов без заголовка. Второй конструктор создает горизонтальную панель с указанным *заголовком*, который отображается только в том случае, если панель перемещается за пределы окна. Третий конструктор создает панель инструментов, расположение которой определяется параметром *ориентация*. Этот параметр должен принимать значение константы `JToolBar.VERTICAL` или `JToolBar.HORIZONTAL`. И четвертый конструктор создает панель инструментов с указанным *заголовком* и расположением, определяемым параметром *ориентация*.

Как правило, панель инструментов применяется в окне с граничной компоновкой по двум причинам. Во-первых, это позволяет первоначально расположить панель инструментов вдоль одной из границ окна. (Зачастую она располагается вдоль верхней границы окна.) И во-вторых, это дает возможность перетаскивать панель инструментов на любую сторону окна.

Панель инструментов можно перетаскивать не только в любое место окна, но и за его пределы. В последнем случае панель инструментов оказывается *отстыкованной*. Если при создании панели инструментов указать заголовок, то он появится, когда панель будет отстыкована.

Кнопки (или другие компоненты) вводятся на панели инструментов таким же образом, как и в строке меню. Для этого достаточно вызвать метод `add()`. Компоненты отображаются на панели инструментов в том порядке, в каком они вводятся.

Как только панель инструментов будет создана, ее *не следует* вводить в строку меню, если таковая имеется. Вместо этого ее следует ввести в контейнер окна. Как упоминалось выше, панель инструментов обычно вводится вдоль верхней границы (северного расположения в граничной компоновке) с горизонтальной ориентацией. А затрагиваемый компонент вводится по центру граничной компоновки. При таком расположении панель инструментов окажется там, где и предполагает

ся ее обнаружить при запуске прикладной программы. Но это не мешает перетасовать панель инструментов в любое другое место окна или вообще за его пределы.

В качестве примера ниже показано, как ввести панель в упоминавшуюся ранее программу MenuDemo. Эта панель состоит из трех кнопок выбора режимов отладки: установки точки прерывания, очистки точки прерывания и возобновления выполнения программы. Все эти стадии процесса отладки следует ввести на панели инструментов.

Прежде всего удалите из программы MenuDemo приведенную ниже строку кода. Благодаря этому во фрейме типа JFrame автоматически используется граничная компоновка.

```
jfrm.setLayout(new FlowLayout());
```

Затем внесите приведенные ниже изменения в той строке кода, где во фрейм вводится метка jlab, поскольку теперь применяется диспетчер граничной компоновки типа BorderLayout.

```
jfrm.add(jlab, BorderLayout.CENTER);
```

В данной строке кода метка jlab явным образом вводится по центру граничной компоновки. (Формально указывать явным образом центральное расположение вводимого компонента совсем не обязательно, поскольку в граничной компоновке компоненты по умолчанию располагаются по центру. Но явное указание центрального расположения компонента дает читающему исходный код ясно понять, что в данном случае применяется граничная компоновка, а метка jlab размещается по центру.)

Далее введите следующий фрагмент кода для создания панели инструментов **Debug** (Отладка):

```
// создать панель инструментов Debug
JToolBar jtb = new JToolBar("Debug");

// загрузить изображения значков кнопок
ImageIcon set = new ImageIcon("setBP.gif");
ImageIcon clear = new ImageIcon("clearBP.gif");
ImageIcon resume = new ImageIcon("resume.gif");

// создать кнопки для панели инструментов
JButton jbbtnSet = new JButton(set);
jbbtnSet.setActionCommand("Set Breakpoint");
jbbtnSet.setToolTipText("Set Breakpoint");
// Установить точку прерывания

JButton jbbtnClear = new JButton(clear);
jbbtnClear.setActionCommand("Clear Breakpoint");
jbbtnClear.setToolTipText("Clear Breakpoint");
// Очистить точку прерывания

JButton jbbtnResume = new JButton(resume);
jbbtnResume.setActionCommand("Resume");
jbbtnResume.setToolTipText("Resume");
// Возобновить выполнение программы

// ввести кнопки на панели инструментов
jtb.add(jbbtnSet);
jtb.add(jbbtnClear);
```

```
jtb.add(jbtnResume);
```

```
// ввести панель инструментов в северном расположении
// на панели содержимого
jfrm.add(jtb, BorderLayout.NORTH);
```

Рассмотрим подробнее приведенный выше фрагмент кода. Сначала в нем создается панель инструментов в виде объекта типа `JToolBar`, и ей присваивается заголовок "Debug". Затем создается ряд объектов типа `ImageIcon` для хранения изображений значков кнопок на панели инструментов. Далее создаются три кнопки для панели инструментов, причем у каждой из них имеется свой значок вместо текстовой надписи. Кроме того, для каждой кнопки явным образом задается команда действия и всплывающая подсказка. Команды действия задаются потому, что кнопкам не присваиваются имена при их создании. Всплывающие подсказки особенно удобны для компонентов, представленных только значками на панели инструментов, поскольку не всегда удастся оформить такие значки, которые интуитивно понятны всем пользователям. После этого кнопки вводятся на панель инструментов, а сама панель — на северной стороне граничной компоновки фрейма.

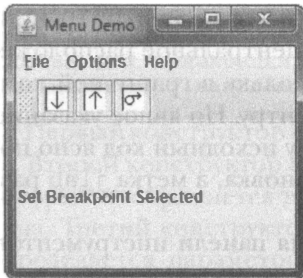


Рис. 33.6. Вид панели инструментов **Debug** в окне программы **MenuDemo**

И наконец, введите приемники действий для панели инструментов, как показано ниже.

```
// ввести приемники действий для панели инструментов
jbtnSet.addActionListener(this);
jbtnClear.addActionListener(this);
jbtnResume.addActionListener(this);
```

Всякий раз, когда пользователь щелкает на кнопке на панели инструментов, наступает событие действия, которое обрабатывается таким же образом, как и другие связанные с меню события. Вид панели инструментов отладки в окне программы **MenuDemo** показан на рис. 33.6.

Действия

Нередко панель инструментов и меню содержат одинаковые элементы. Например, режимы отладки, доступные на панели инструментов **Debug** из предыдущего примера, могут быть также доступны для выбора из меню. В подобных случаях выбор какого-нибудь варианта (например, установки точки прерывания при отладке) из меню или панели инструментов вызывает одно и то же действие. Кроме того, для обозначения кнопки на панели инструментов и пункта меню используется (чаще всего) один и тот же значок. Более того, если кнопка становится недоступной на панели инструментов, то недоступным оказывается и соответствующий пункт меню. В подобных случаях обычно накапливается довольно большой объем дублирующегося взаимосвязанного кода, что нельзя назвать оптимальным решением. Правда, библиотека **Swing** предлагает выход из этого затруднительного положения в виде *действий*.

Действие является экземпляром интерфейса `Action`, расширяющего интерфейс `ActionListener` и предоставляющего средства для сочетания сведений о состоянии с обработкой событий методом `actionPerformed()`. Благодаря такому

сочетанию одним действием можно управлять двумя или более компонентами. Например, действие позволяет централизовать управление кнопкой на панели инструментов и пунктом в меню. Вместо дублирования кода в прикладной программе нужно лишь создать действие, управляющее обоими компонентами.

В связи с тем что интерфейс `Action` расширяет интерфейс `ActionListener`, действие должно предоставить реализацию метода `actionPerformed()`. Этот метод будет обрабатывать события действия, генерируемые объектами, связанными с отдельным действием.

Помимо наследуемого метода `actionPerformed()`, в интерфейсе `Action` определяется ряд собственных методов. Особый интерес среди них вызывает метод `putValue()`, задающий значения различных свойств, связанных с действием, как показано ниже.

```
void putValue(String ключ, Object значение)
```

Этот метод присваивает указанное значение требуемому свойству, обозначаемому параметром *ключ*. В интерфейсе `Action` предоставляется также метод `getValue()`, получающий указанное свойство, хотя в представленном далее примере он не применяется. Ниже приведена общая форма этого метода. Он возвращает ссылку на свойство, обозначаемое параметром *ключ*.

```
Object getValue(String ключ)
```

Значения ключей, используемые в методах `putValue()` и `getValue()`, приведены в табл. 33.2.

Таблица 33.2. Значения ключей, обозначающих свойства, связанные с действиями

Значение ключа	Описание
static final String ACCELERATOR_KEY	Представляет свойство оперативной клавиши. Оперативные клавиши указываются в виде объектов типа KeyStroke
static final String ACTION_COMMAND_KEY	Представляет свойство команды действия. Команда действия указывается в виде символьной строки
static final String DISPLAYED_MNEMONIC_INDEX_KEY	Представляет индекс символа, отображаемого в мнемоническом виде. Это значение типа Integer
static final String LARGE_ICON_KEY	Представляет крупный значок, связанный с действием. Значок указывается в виде объекта типа Icon
static final String LONG_DESCRIPTION	Представляет длинное описание действия. Это описание указывается в виде символьной строки
static final String MNEMONIC_KEY	Представляет мнемоническое свойство. Мнемоника указывается в виде константы типа KeyEvent
static final String NAME	Представляет имя действия, которое становится также именем элемента (кнопки или пункта меню), с которым связано это действие. Это имя указывается в виде символьной строки

Значение ключа	Описание
static final String SELECTED_KEY	Представляет состояние выбора. Если оно установлено, то элемент выбран. Состояние представлено логическим значением
static final String SHORT_ DESCRIPTION	Представляет текст всплывающего сообщения, связанный с действием. Этот текст указывается в виде символьной строки
static final String SMALL_ ICON	Представляет значок, связанный с действием. Этот значок указывается в виде объекта тип Icon

Например, чтобы задать мнемонику для буквы X, достаточно сделать следующий вызов метода `putValue()`:

```
actionOb.putValue(MNEMONIC_KEY, new Integer(KeyEvent.VK_X));
```

Единственным свойством типа `Action`, недоступным через методы `putValue()` и `getValue()`, является разрешенное или запрещенное состояние. Для доступа к нему следует воспользоваться методами `setEnabled()` и `isEnabled()`. Ниже приведены общие формы этих методов.

```
void setEnabled(boolean разрешение)
boolean isEnabled()
```

Если параметр *разрешение* метода `setEnabled()` принимает логическое значение `true`, то действие разрешено, а иначе оно запрещено. Если же действие разрешено, то метод `isEnabled()` возвращает логическое значение `true`, а иначе — логическое значение `false`.

Полностью реализовать интерфейс `Action` можно и самостоятельно, но обычно в этом нет никакой нужды. Вместо этого в библиотеке `Swing` предоставляются частичные его реализации, называемые `AbstractAction` и допускающие расширение. Расширяя класс `AbstractAction`, придется реализовать только один метод `actionPerformed()`, а остальные методы из интерфейса `Action` предоставляются автоматически. У класса `AbstractAction` имеются три конструктора. Ниже приведен конструктор, применяемый здесь и далее в этой главе. Этот конструктор создает абстрактное действие в виде объекта класса `AbstractAction`, получающее заданное имя и указанное изображение значка.

```
AbstractAction(String имя, Icon изображение)
```

Как только действие будет создано, его можно ввести на панели инструментов типа `JToolBar` и воспользоваться им для создания пункта меню типа `JMenuItem`. Чтобы ввести действие на панели инструментов типа `JToolBar`, следует воспользоваться следующим вариантом метода `add()`:

```
void add(Action объект_действия)
```

где параметр *объект_действия* обозначает действие, вводимое на панели инструментов. Свойства, определяемые параметром *объект_действия*, используются для создания кнопки на панели инструментов. А для создания пункта меню из конкретного действия служит приведенный ниже конструктор класса `JMenuItem`, где

объект_действия обозначает действие, используемое для создания пункта меню в соответствии с его свойствами.

JMenuItem(Action *объект_действия*)

На заметку! Помимо компонентов типа **JToolBar** и **JMenuItem**, действия поддерживаются и в других компонентах **Swing**, в том числе **JPopupMenu**, **JButton**, **JRadioButton**, **JCheckBox**, **JRadioButtonMenuItem** и **JCheckBoxMenuItem**.

Продemonстрируем преимущества действий на примере управления созданной ранее панелью инструментов **Debug** в рассматриваемом здесь примере программы **MenuDemo**. Для этого меню **Options** будет дополнено подменю **Debug** со следующими пунктами, аналогичными кнопкам выбора режимов отладки на панели инструментов **Debug**: **Set Breakpoint** (Установить точку прерывания), **Clear Breakpoint** (Очистить точку прерывания) и **Resume** (Возобновить выполнение). Одни и те же действия будут поддерживать как пункты в подменю **Debug**, так и кнопки на одноименной панели инструментов. Следовательно, вместо того чтобы писать дублирующий код управления подменю и панелью инструментов **Debug** по отдельности, достаточно организовать действия, которые будут управлять обоими этими элементами ГПИ.

Итак, создайте сначала внутренний класс **DebugAction**, расширяющий класс **AbstractAction** и приведенный ниже.

```
// Класс действий для подменю и панели инструментов Debug
class DebugAction extends AbstractAction {
    public DebugAction(String name, Icon image, int mnem,
                       int accel, String tTip) {
        super(name, image);
        putValue(ACCELERATOR_KEY,
                 KeyStroke.getKeyStroke(accel,
                                         InputEvent.CTRL_DOWN_MASK));
        putValue(MNEMONIC_KEY, new Integer(mnem));
        putValue(SHORT_DESCRIPTION, tTip);
    }

    // обработать события как на панели инструментов, так
    // и в подменю Debug
    public void actionPerformed(ActionEvent ae) {
        String comStr = ae.getActionCommand();

        jlab.setText(comStr + " Selected"); // Выбрано указанное

        // изменить разрешенное состояние вариантов выбора
        // режимов установки и очистки точек прерывания
        if(comStr.equals("Set Breakpoint")) {
            clearAct.setEnabled(true);
            setAct.setEnabled(false);
        } else if(comStr.equals("Clear Breakpoint")) {
            clearAct.setEnabled(false);
            setAct.setEnabled(true);
        }
    }
}
```

Приведенный выше класс `DebugAction` расширяет класс `AbstractAction`. В итоге получается класс действия, предназначенный для определения свойств, связанных с подменю и панелью инструментов **Debug**. Его конструктор принимает пять параметров, позволяющих указать следующие элементы:

- Имя.
- Значок.
- Мнемоника.
- Оперативная клавиша.
- Всплывающая подсказка.

Два первых параметра передаются конструктору класса `AbstractAction` по ссылке `super`, а остальные три задаются через вызов метода `putValue()`.

Метод `actionPerformed()` из класса `DebugAction` обрабатывает события, связанные с выполняемым действием. Это означает, что если для создания кнопки на панели инструментов и пункта меню используется экземпляр класса `DebugAction`, то события, генерируемые любым из этих компонентов, обрабатываются методом `actionPerformed()` из класса `DebugAction`. Следует заметить, что этот обработчик событий отображает результаты выбора в метке `jlab`. Если выбирается вариант (кнопка или пункт меню) **Set Breakpoint**, то вариант **Clear Breakpoint** становится доступным, а вариант **Set Breakpoint** — недоступным. Если же выбирается вариант **Clear Breakpoint**, то вариант **Set Breakpoint** становится доступным, а вариант **Clear Breakpoint** — недоступным. Это наглядно показывает, каким образом действие позволяет активизировать или деактивизировать компонент. Если действие запрещено, то запрещены и все варианты его использования. В данном случае становятся недоступным кнопка на панели инструментов и пункт меню **Set Breakpoint**, если запрещено действие установки точки прерывания.

Затем введите приведенные ниже переменные экземпляра класса `DebugAction` в программу `MenuDemo`.

```
DebugAction setAct;
DebugAction clearAct;
DebugAction resumeAct;
```

Далее создайте три значка типа `ImageIcons`, представляющих режимы отладки, как показано ниже.

```
// загрузить изображения для обозначения действий при отладке
ImageIcon setIcon = new ImageIcon("setBP.gif");
ImageIcon clearIcon = new ImageIcon("clearBP.gif");
ImageIcon resumeIcon = new ImageIcon("resume.gif");
```

А теперь создайте действия, управляющие вариантами выбора режимов отладки в подменю и на панели инструментов **Debug**, как показано ниже.

```
// создать действия
setAct =
    new DebugAction("Set Breakpoint",
                    setIcon,
                    KeyEvent.VK_S,
                    KeyEvent.VK_B,
```

```

        "Set a break point.");
        // Установить точку прерывания

clearAct =
    new DebugAction("Clear Breakpoint",
        clearIcon,
        KeyEvent.VK_C,
        KeyEvent.VK_L,
        "Clear a break point.");
        // Очистить точку прерывания

resumeAct =
    new DebugAction("Resume",
        resumeIcon,
        KeyEvent.VK_R,
        KeyEvent.VK_R,
        "Resume execution after breakpoint.");
        // Возобновить выполнение после точки прерывания

// сделать первоначально недоступным вариант выбора Clear Breakpoint
clearAct.setEnabled(false);

```

Следует заметить, что для быстрого выбора варианта **Set Breakpoint** назначена оперативная клавиша , а для быстрого выбора варианта **Clear Breakpoint** — оперативная клавиша <L>. Выбор именно этих клавиш, а не клавиш <S> и <C> объясняется тем, что последние уже назначены для быстрого выбора пунктов **Save** и **Close** соответственно из меню **File**. Но в то же время их можно использовать в качестве мнемоники, поскольку каждая мнемоника действует только в пределах своего меню. Следует также заметить, что действие, представляющее режим очистки точки прерывания (**Clear Breakpoint**), первоначально запрещено. Оно будет разрешено только после установки точки прерывания.

Далее воспользуйтесь действиями для создания и ввода кнопок на панели инструментов, как показано ниже.

```

// создать кнопки для панели инструментов,
// используя соответствующие действия
JButton jbtnSet = new JButton(setAct);
JButton jbtnClear = new JButton(clearAct);
JButton jbtnResume = new JButton(resumeAct);

// создать панель инструментов Debug
JToolBar jtb = new JToolBar("Breakpoints");

// ввести кнопки на панели инструментов
jtb.add(jbtnSet);
jtb.add(jbtnClear);
jtb.add(jbtnResume);

// ввести панель инструментов в северном расположении
// панели содержимого
jfrm.add(jtb, BorderLayout.NORTH);

```

И наконец, создайте подменю **Debug**, как показано ниже.

```

// создать подменю Debug, входящее в меню Options,
// используя действия для создания пунктов этого подменю
JMenu jmDebug = new JMenu("Debug");
JMenuItem jmiSetBP = new JMenuItem(setAct);
JMenuItem jmiClearBP = new JMenuItem(clearAct);

```



```

JMenuItem jmiResume = new JMenuItem(resumeAct);
jmDebug.add(jmiSetBP);
jmDebug.add(jmiClearBP);
jmDebug.add(jmiResume);
jmOptions.add(jmDebug);

```

После внесения описанных выше изменений и дополнений в программу MenuDemo созданные действия будут одновременно управлять подменю и панелью инструментов Debug. Таким образом, любое изменение в свойстве действия (например, его запрет) будет оказывать воздействие на все варианты его использования. Вид, который теперь принимает подменю и панель инструментов Debug в окне данной программы, показан на рис. 33.7.

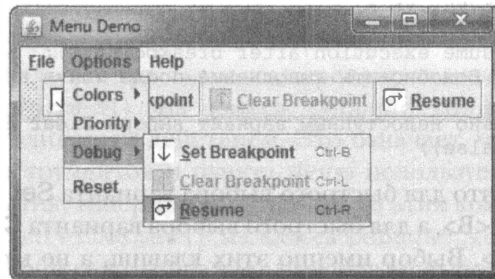


Рис. 33.7. Вид подменю и панели инструментов Debug, управляемых действиями, в окне программы MenuDemo

Составление окончательного варианта программы MenuDemo

По ходу изложения материала этой главы в исходный вариант прикладной программы MenuDemo было внесено немало изменений и дополнений. Прежде чем завершить эту главу, целесообразно составить окончательный вариант данной программы с учетом всех изменений и дополнений. Это позволит не только избежать недоразумений относительно согласованности отдельных частей программы, но и получить рабочую программу для демонстрации меню и дальнейшего экспериментирования с ними.

В приведенный ниже окончательный вариант прикладной программы MenuDemo вошли все изменения и дополнения, описанные ранее в этой главе. Ради большей ясности эта программа была реорганизована таким образом, чтобы использовать отдельные методы для создания разных меню и панели инструментов. Следует также иметь в виду, что некоторые переменные, связанные с меню, в том числе jmb, jmFile и jtb, были превращены в переменные экземпляра, чтобы сделать их доступными в любой части класса.

```
// Окончательный вариант программы complete MenuDemo
```

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

```

```
class MenuDemo implements ActionListener {

    JLabel jlab;

    JMenuBar jmb;

    JToolBar jtb;

    JPopupMenu jpu;

    DebugAction setAct;
    DebugAction clearAct;
    DebugAction resumeAct;

    MenuDemo() {
        // создать новый контейнер типа JFrame
        JFrame jfrm = new JFrame("Complete Menu Demo");
        // Полная демонстрация меню

        // использовать граничную компоновку, выбираемую по умолчанию

        // задать исходные размеры фрейма
        jfrm.setSize(360, 200);

        // завершить прикладную программу, когда пользователь
        // закроет ее окно
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // создать метку для отображения результатов выбора из меню
        jlab = new JLabel();

        // создать строку меню
        jmb = new JMenuBar();

        // создать меню File
        makeFileMenu();

        // создать действия отладки
        makeActions();

        // создать панель инструментов
        makeToolBar();

        // создать меню Options
        makeOptionsMenu();

        // создать меню Help
        makeHelpMenu();

        // создать меню Edit
        makeEditPUMenu();

        // ввести приемник событий запуска всплывающего меню
        jfrm.addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent me) {
                if(me.isPopupTrigger())
                    jpu.show(me.getComponent(), me.getX(), me.getY());
            }
            public void mouseReleased(MouseEvent me) {
                if(me.isPopupTrigger())
```

```

        jpu.show(me.getComponent(), me.getX(), me.getY());
    }
});

// ввести метку в центре панели содержимого
jfrm.add(jlab, SwingConstants.CENTER);

// ввести панель инструментов в северном положении
// панели содержимого
jfrm.add(jtb, BorderLayout.NORTH);

// ввести строку меню во фрейм
jfrm.setJMenuBar(jmb);

// отобразить фрейм
jfrm.setVisible(true);
}

// Обработать события действия от пунктов меню.
// Здесь НЕ обрабатываются события, генерируемые
// при выборе режимов отладки в подменю или на панели
// инструментов Debug
public void actionPerformed(ActionEvent ae) {
    // получить команду действия из выбранного меню
    String comStr = ae.getActionCommand();

    // выйти из программы, если пользователь выберет пункт меню Exit
    if(comStr.equals("Exit")) System.exit(0);

    // отобразить в противном случае результат выбора из меню
    jlab.setText(comStr + " Selected");
}

// Класс действий для подменю и панели инструментов Debug
class DebugAction extends AbstractAction {
    public DebugAction(String name, Icon image, int mnem,
                       int accel, String tTip) {
        super(name, image);
        putValue(ACCELERATOR_KEY,
                 KeyStroke.getKeyStroke(
                     accel, InputEvent.CTRL_DOWN_MASK));
        putValue(MNEMONIC_KEY, new Integer(mnem));
        putValue(SHORT_DESCRIPTION, tTip);
    }

    // обработать события как на панели инструментов, так
    // и в подменю Debug
    public void actionPerformed(ActionEvent ae) {
        String comStr = ae.getActionCommand();

        jlab.setText(comStr + " Selected"); // Выбрать указанное

        // изменить разрешенное состояние вариантов выбора
        // режимов установки и очистки точек прерывания
        if(comStr.equals("Set Breakpoint")) {
            clearAct.setEnabled(true);
            setAct.setEnabled(false);
        } else if(comStr.equals("Clear Breakpoint")) {
            clearAct.setEnabled(false);
            setAct.setEnabled(true);
        }
    }
}

```

```
}  
}  
  
// создать меню File с мнемоникой и оперативными клавишами  
void makeFileMenu() {  
    JMenu jmFile = new JMenu("File");  
    jmFile.setMnemonic(KeyEvent.VK_F);  
  
    JMenuItem jmiOpen = new JMenuItem("Open", KeyEvent.VK_O);  
    jmOpen.setAccelerator(  
        KeyStroke.getKeyStroke(  
            KeyEvent.VK_O, InputEvent.CTRL_DOWN_MASK));  
  
    JMenuItem jmiClose = new JMenuItem("Close", KeyEvent.VK_C);  
    jmClose.setAccelerator(  
        KeyStroke.getKeyStroke(  
            KeyEvent.VK_C, InputEvent.CTRL_DOWN_MASK));  
  
    JMenuItem jmiSave = new JMenuItem("Save", KeyEvent.VK_S);  
    jmSave.setAccelerator(  
        KeyStroke.getKeyStroke(  
            KeyEvent.VK_S, InputEvent.CTRL_DOWN_MASK));  
    JMenuItem jmiExit = new JMenuItem("Exit", KeyEvent.VK_E);  
    jmExit.setAccelerator(  
        KeyStroke.getKeyStroke(KeyEvent.VK_E,  
            InputEvent.CTRL_DOWN_MASK));  
  
    jmFile.add(jmiOpen);  
    jmFile.add(jmiClose);  
    jmFile.add(jmiSave);  
    jmFile.addSeparator();  
    jmFile.add(jmiExit);  
    jmb.add(jmFile);  
  
    // ввести приемники действий для пунктов меню File  
    jmiOpen.addActionListener(this);  
    jmiClose.addActionListener(this);  
    jmiSave.addActionListener(this);  
    jmiExit.addActionListener(this);  
}  
  
// создать меню Options  
void makeOptionsMenu() {  
    JMenu jmOptions = new JMenu("Options");  
  
    // создать подменю Colors  
    JMenu jmColors = new JMenu("Colors");  
  
    // использовать флажки, чтобы пользователь мог выбрать  
    // сразу несколько цветов  
    JCheckBoxMenuItem jmiRed = new JCheckBoxMenuItem("Red");  
    JCheckBoxMenuItem jmiGreen = new JCheckBoxMenuItem("Green");  
    JCheckBoxMenuItem jmiBlue = new JCheckBoxMenuItem("Blue");  
  
    // ввести пункты в подменю Colors  
    jmColors.add(jmiRed);  
    jmColors.add(jmiGreen);  
    jmColors.add(jmiBlue);  
    jmOptions.add(jmColors);  
  
    // создать подменю Priority
```

```

JMenu jmPriority = new JMenu("Priority");

// Использовать кнопки-переключатели для установки приоритета.
// Благодаря этому в меню не только отображается установленный
// приоритет, но и гарантируется установка одного и только
// одного приоритета. Исходно выбирается кнопка-переключатель
// в пункте меню High
JRadioButtonMenuItem jmiHigh =
    new JRadioButtonMenuItem("High", true);
JRadioButtonMenuItem jmiLow =
    new JRadioButtonMenuItem("Low");

// ввести пункты в подменю Priority
jmPriority.add(jmiHigh);
jmPriority.add(jmiLow);
jmOptions.add(jmPriority);

// создать группу кнопок-переключателей
// в пунктах подменю Priority
ButtonGroup bg = new ButtonGroup();
bg.add(jmiHigh);
bg.add(jmiLow);

// создать подменю Debug, входящее в меню Options,
// используя действия для создания пунктов этого подменю
JMenu jmDebug = new JMenu("Debug");
JMenuItem jmiSetBP = new JMenuItem(setAct);
JMenuItem jmiClearBP = new JMenuItem(clearAct);
JMenuItem jmiResume = new JMenuItem(resumeAct);

// ввести пункты в подменю Debug
jmDebug.add(jmiSetBP);
jmDebug.add(jmiClearBP);
jmDebug.add(jmiResume);
jmOptions.add(jmDebug);

// создать пункт меню Reset
JMenuItem jmiReset = new JMenuItem("Reset");
jmOptions.addSeparator();
jmOptions.add(jmiReset);

// И наконец, ввести все выбираемые меню в строку меню
jmb.add(jmOptions);

// ввести приемники действий для пунктов меню Options,
// кроме тех, что поддерживаются в подменю Debug
jmiRed.addActionListener(this);
jmiGreen.addActionListener(this);
jmiBlue.addActionListener(this);
jmiHigh.addActionListener(this);
jmiLow.addActionListener(this);
jmiReset.addActionListener(this);
}

// создать меню Help
void makeHelpMenu() {
    JMenu jmHelp = new JMenu("Help");

    // ввести значок для пункта меню About
    ImageIcon icon = new ImageIcon("AboutIcon.gif");

```

```
JMenuItem jmiAbout = new JMenuItem("About", icon);
jmiAbout.setToolTipText("Info about the MenuDemo program.");
jmHelp.add(jmiAbout);
jmb.add(jmHelp);

// ввести приемник действий для пункта меню About
jmiAbout.addActionListener(this);
}

// создать действия для управления подменю и
// панелью инструментов Debug
void makeActions() {
    // загрузить изображения для обозначения действий
    ImageIcon setIcon = new ImageIcon("setBP.gif");
    ImageIcon clearIcon = new ImageIcon("clearBP.gif");
    ImageIcon resumeIcon = new ImageIcon("resume.gif");

    // создать действия
    setAct =
        new DebugAction("Set Breakpoint",
            setIcon,
            KeyEvent.VK_S,
            KeyEvent.VK_B,
            "Set a break point.");

    clearAct =
        new DebugAction("Clear Breakpoint",
            clearIcon,
            KeyEvent.VK_C,
            KeyEvent.VK_L,
            "Clear a break point.");

    resumeAct =
        new DebugAction("Resume",
            resumeIcon,
            KeyEvent.VK_R,
            KeyEvent.VK_R,
            "Resume execution after breakpoint.");

    // Initially disable the Clear Breakpoint option.
    clearAct.setEnabled(false);
}

// создать панель инструментов Debug
void makeToolBar() {
    // создать кнопки для панели инструментов,
    // используя соответствующие действия
    JButton jbtnSet = new JButton(setAct);
    JButton jbtnClear = new JButton(clearAct);
    JButton jbtnResume = new JButton(resumeAct);

    // создать панель инструментов Debug
    jtb = new JToolBar("Breakpoints");

    // ввести кнопки на панели инструментов
    jtb.add(jbtnSet);
    jtb.add(jbtnClear);
    jtb.add(jbtnResume);
}

// создать всплывающее меню Edit
```

```

void makeEditPopupMenu() {
    jpu = new JPopupMenu();

    // создать пункты всплывающего меню Edit
    JMenuItem jmiCut = new JMenuItem("Cut");
    JMenuItem jmiCopy = new JMenuItem("Copy");
    JMenuItem jmiPaste = new JMenuItem("Paste");

    // ввести пункты во всплывающее меню Edit
    jpu.add(jmiCut);
    jpu.add(jmiCopy);
    jpu.add(jmiPaste);

    // ввести приемники действий для всплывающего меню Edit
    jmiCut.addActionListener(this);
    jmiCopy.addActionListener(this);
    jmiPaste.addActionListener(this);
}

public static void main(String args[]) {
    // создать фрейм в потоке диспетчеризации событий
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            new MenuDemo();
        }
    });
}
}

```

Дальнейшее изучение библиотеки Swing

В библиотеке Swing определяется довольно большой набор инструментальных средств для построения ГПИ. Она предоставляет немало других средств, которые вам придется освоить самостоятельно. В частности, библиотека Swing предоставляет классы `JOptionPane` и `JDialog`, упрощающие процесс построения диалоговых окон, а также дополнительные элементы управления, помимо представленных в главе 31. Рекомендуется изучить два компонента: `JSpinner` (для создания счетчика) и `JFormattedTextField` (для форматирования текста). Можете также поэкспериментировать с определением собственной модели для различных компонентов. Откровенно говоря, самый лучший способ изучения возможностей библиотеки Swing — экспериментировать с ней.

ЧАСТЬ

IV

Введение в программирование ГПИ средствами JavaFX

ГЛАВА 34

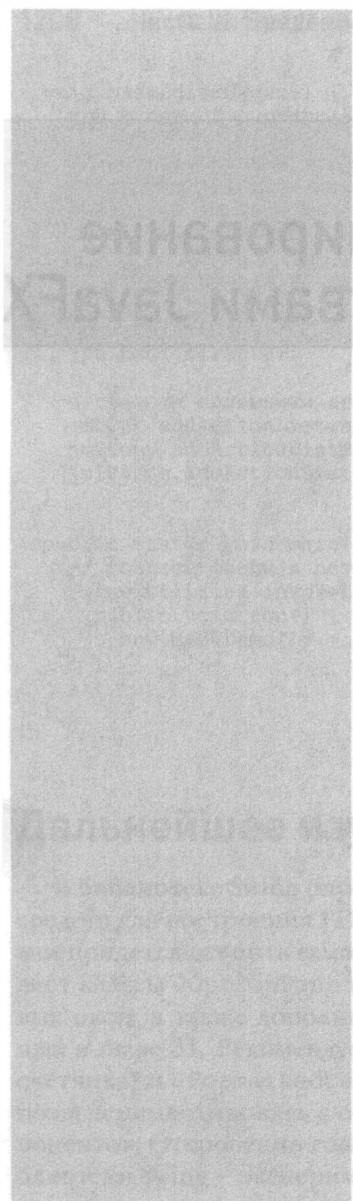
Введение в JavaFX

ГЛАВА 35

Элементы управления
JavaFX

ГЛАВА 36

Введение в меню JavaFX



Введение в JavaFX

Как и все удачно разработанные языки программирования, Java продолжает развиваться и совершенствоваться. И это относится к его библиотекам. Характерным примером такого развития служат библиотеки (или так называемые каркасы) для построения ГПИ. Как пояснялось ранее в данной книге, первоначально для построения ГПИ была разработана библиотека AWT. Но в силу ряда присущих ей ограничений вскоре была разработана библиотека Swing, предлагавшая более совершенный подход к построению ГПИ. Эта библиотека оказалась настолько удачной, что до сих пор остается основной в Java для построения ГПИ. И скорее всего, она таковой останется еще долго, несмотря на быстро меняющуюся ситуацию в области программирования! Тем не менее библиотека Swing была создана в то время, когда в разработке программного обеспечения господствующее положение занимали корпоративные приложения. А ныне особое значение приобрели потребительские приложения, особенно для мобильных устройств. И от таких приложений зачастую требуется визуальная привлекательность. Более того, наметилась общая тенденция к наделению приложений, независимо от их типа, захватывающими визуальными эффектами. Поэтому с учетом этой тенденции потребовался новый подход к построению ГПИ, что и привело к разработке библиотеки JavaFX. Она стала новым поколением платформы и библиотеки для построения ГПИ клиентских приложений в Java.

JavaFX служит эффективным, рациональным, удобным каркасом для построения современных визуально привлекательных ГПИ, а следовательно, это довольно крупная система, которую, как и Swing, просто невозможно описать полностью в рамках данной книги. Поэтому в этой и двух последующих главах представлены лишь основные средства и способы построения ГПИ на основе JavaFX. Овладев этими основами, вы сможете без особого труда самостоятельно изучить остальные особенности JavaFX.

В связи с появлением JavaFX возникает следующий вполне резонный вопрос: служит ли JavaFX в качестве замены Swing? Определенно служит. Но, принимая во внимание немалый объем унаследованного кода, а также огромную армию специалистов, умеющих разрабатывать приложения на основе библиотеки Swing, она еще нескоро выйдет из употребления. И это особенно касается корпоративных приложений. Тем не менее JavaFX была ясно определена как перспективная платформа. Ожидается, что через несколько лет JavaFX полностью вытеснит Swing для разработки новых проектов. Поэтому ни один программирующий на Java не должен игнорировать JavaFX.

Прежде чем продолжить дальше, следует вкратце упомянуть, что разработка JavaFX происходила в две основные стадии. Первоначально разработка JavaFX велась на основе языка сценариев *JavaFX Script*, но этот язык был упразднен. Поэтому, начиная с версии 2.0 программирование JavaFX велось уже непосредственно на Java, в результате чего появился исчерпывающий прикладной программный интерфейс API. В JavaFX поддерживается также язык FXML, на котором можно (хотя и не обязательно) размечать пользовательский интерфейс. Начиная с обновления 4 версии JDK 7, JavaFX входит в комплект Java. А последняя версия JavaFX 8 входит в комплект JDK 8. (Номер 8 для версии выбран для того, чтобы соответствовать версии JDK, поэтому номера версии 3–7 были пропущены.) Таким образом, на момент написания данной книги самой последней была версия JavaFX 8, которая здесь и рассматривается. Более того, под обозначением *JavaFX* здесь и далее подразумевается версия JavaFX 8.

На заметку! В этой и двух последующих главах предполагается, что вы владеете основами обработки событий, представленными в главе 24, а также основами Swing, описанными в части III данной книги.

Основные понятия JavaFX

В общем, каркас JavaFX обладает всеми полезными средствами Swing, включая легковесность компонентов и поддержку архитектуры MVC. Большая часть того, что вам уже известно о построении ГПИ средствами Swing, пригодится и в JavaFX. Тем не менее у JavaFX и Swing имеются существенные отличия.

Первые отличия, с точки зрения программирования, состоят в организации самого каркаса и взаимосвязи главных его компонентов. Проще говоря, JavaFX предлагает более рациональный, простой в употреблении и усовершенствованный подход к построению ГПИ, а также значительно упрощает воспроизведение объектов благодаря автоматической перерисовке. Решать эту задачу вручную больше не нужно. Но это совсем не означает, что библиотека Swing разработана неудачно. Просто искусство программирования заметно продвинулось вперед, и плоды этого прогресса были пожаты в JavaFX. В целом JavaFX способствует выбору более динамичного визуальнo подхода к построению ГПИ.

Пакеты JavaFX

Компоненты JavaFX содержатся в отдельных пакетах, имена которых начинаются с префикса `javafx`. На момент написания данной книги насчитывалось более 30 пакетов, составляющих прикладной программный интерфейс JavaFX API. К их числу относятся следующие пакеты: `javafx.application`, `javafx.stage`, `javafx.scene` и `javafx.scene.layout`. В примерах этой главы употребляются лишь некоторые пакеты, поэтому вам придется ознакомиться с функциональными возможностями остальных пакетов JavaFX самостоятельно, а они довольно обширны.

Классы подмостков и сцены

Центральным понятием, внедренным в JavaFX, являются *подмостки*. Как и в настоящей театральной постановке, подмостки содержат *сцену*. Проще говоря, подмостки определяют пространство, а сцена — то, что находится в этом пространстве. Иными словами, подмостки служат контейнером для сцен, а сцена — контейнером для элементов, которые ее составляют. Таким образом, все JavaFX-приложения состоят, по крайней мере, из одних подмостков и одной сцены. Эти элементы построения ГПИ инкапсулированы в классах Stage и Scene, входящих в состав JavaFX API. Чтобы создать JavaFX-приложение, нужно ввести хотя бы один объект типа Scene в контейнер типа Stage. Рассмотрим оба эти класса более подробно.

Класс Stage служит контейнером верхнего уровня. Все JavaFX-приложения автоматически получают доступ к одному контейнеру типа Stage, называемому *главными подмостками*. Главные подмостки предоставляются исполняющей системой при запуске JavaFX-приложения на выполнение. Для многих приложений требуются лишь одни главные подмостки, хотя можно создать и другие подмостки.

Как упоминалось выше, класс Scene служит контейнером для всех элементов, составляющих сцену. Это могут быть элементы управления, в том числе экранные кнопки и флажки, а также текст и графика. Чтобы создать сцену, следует ввести все эти элементы в экземпляр класса Scene.

Узлы и графы сцены

Отдельные элементы сцены называются *узлами*. Например, узлом считается элемент управления экранной кнопкой. Но сами узлы могут состоять из групп других узлов. Более того, у каждого узла может быть потомок, или порожденный узел, и тогда он называется *родительским узлом*, или *узлом ветвления*. А узлы без потомков являются концевыми и называются *листьями*. Совокупность всех узлов в сцене называется *графом*, образующим *дерево*.

В графе сцены имеется специальный тип узла, называемый *корневым*. Это самый верхний и единственный узел графа, не имеющий родителя. Следовательно, все узлы, кроме корневого, являются родительскими, причем все они прямо или косвенно происходят от корневого узла.

Базовым для всех узлов служит класс Node. Имеется ряд других классов, которые прямо или косвенно происходят от класса Node. К их числу относятся классы Parent, Group, Region и Control.

Компоновки

В JavaFX предоставляется ряд панелей компоновки для управления процессом размещения элементов в сцене. Например, класс FlowPane предоставляет поточную компоновку, а класс GridPane — сеточную компоновку в виде рядов и столбцов таблицы. Имеются и другие разновидности компоновки, в том числе компоновка типа BorderPane, аналогичная граничной компоновке типа BorderLayout из библиотеки AWT. Панели компоновки входят в состав пакета `javafx.scene.layout`.

Класс приложения и методы его жизненного цикла

JavaFX-приложение должно быть подклассом, производным от класса `Application`, входящего в состав пакета `javafx.application`. Следовательно, класс приложения должен расширять класс `Application`. В классе `Application` определяются три метода жизненного цикла, которые могут быть переопределены в классе приложения. Эти методы называются `init()`, `start()`, `stop()` и приведены ниже в том порядке, в каком они вызываются.

```
void init()
abstract void start(Stage главные_подмостки)
void stop()
```

Метод `init()` вызывается в тот момент, когда приложение начинает выполняться. Он служит для выполнения различных инициализаций. Но, как поясняется далее, с его помощью *нельзя* создать подмостки или построить сцену. Если же инициализация не требуется, то и переопределять метод `init()` не нужно, поскольку по умолчанию предоставляется его пустой вариант.

Метод `start()` вызывается после метода `init()`. Именно с него и начинается приложение, поскольку он *позволяет* построить и установить сцену. Как видите, в качестве параметра *главные_подмостки* этому методу передается ссылка на объект типа `Stage`. Это главные подмостки, предоставляемые исполняющей системой. (Имеется возможность создать и другие подмостки, но для простых приложений этого не требуется.) Следует заметить, что метод `start()` является абстрактным, а следовательно, он должен быть переопределен в приложении.

Когда приложение завершается, вызывается метод `stop()`. Именно в нем должны быть произведены все операции очистки или закрытия. Если же такие операции не требуются, то по умолчанию предоставляется пустой вариант данного метода.

Запуск JavaFX-приложения

Для того чтобы запустить автономное JavaFX-приложение на выполнение, следует вызвать метод `launch()`, определяемый в классе `Application`. У этого метода имеются две общие формы. Ниже приведена его общая форма, применяемая здесь и далее в этой главе.

```
public static void launch(String ... аргументы)
```

Здесь параметр *аргументы* обозначает пустой, возможно, список символьных строк, обозначающих, как правило, аргументы командной строки. Вызов метода `launch()` приводит к построению приложения и последующему вызову методов `init()` и `start()`. Возврат из метода `launch()` не происходит до тех пор, пока приложение не завершится. В рассматриваемой здесь форме данного метода начинает выполняться подкласс, производный от класса `Application`, а из него вызывается метод `launch()`. Во второй форме данного метода можно указать для запуска другой, а не охватывающий его класс.

Прежде чем продолжить дальше, следует особо подчеркнуть, что в JavaFX-приложения, упакованные с помощью утилиты `javafxpackager` (или ее эквива-

лента в ИСР), совсем не обязательно включать вызов метода `launch()`. Тем не менее его включение упрощает процесс тестирования и отладки приложения и позволяет эксплуатировать программу, не прибегая к созданию архивного JAR-файла. Именно поэтому метод `launch()` включен в примеры всех JavaFX-приложений, представленные в части IV данной книги.

Скелет JavaFX-приложения

Все JavaFX-приложения создаются по одному и тому же образцу — типичному скелету. Поэтому, прежде чем перейти к рассмотрению каких-нибудь других средств JavaFX, стоит продемонстрировать скелет JavaFX-приложения. Помимо общей формы такого приложения, скелет демонстрирует порядок его запуска и вызова методов его жизненного цикла. Когда вызывается каждый метод жизненного цикла, на консоль выводится извещающее об этом сообщение. Ниже приведен весь скелет JavaFX-приложения.

```
// Скелет JavaFX-приложения

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;

public class JavaFXSkel extends Application {

    public static void main(String[] args) {

        System.out.println("Запуск JavaFX-приложения.");

        // запустить JavaFX-приложение, вызвав метод launch()
        launch(args);
    }

    // переопределить метод init()
    public void init() {
        System.out.println("В теле метода init().");
    }

    // переопределить метод start()
    public void start(Stage myStage) {

        System.out.println("В теле метода start().");

        // присвоить заголовок подмосткам
        myStage.setTitle("JavaFX Skeleton.");
        // Скелет JavaFX-приложения

        // Создать корневой узел. В данном случае используется
        // панель поточной компоновки, хотя возможны и другие
        // варианты компоновки
        FlowPane rootNode = new FlowPane();

        // создать сцену
        Scene myScene = new Scene(rootNode, 300, 200);
```

```

// установить сцену на подмостках
myStage.setScene(myScene);

// показать подмостки и сцену на них
myStage.show();
}

// переопределить метод stop()
public void stop() {
    System.out.println("В теле метода stop().");
}
}

```

Несмотря на всю краткость такого скелета JavaFX-приложения, его можно скомпилировать и запустить на выполнение. В итоге на экране появляется окно, приведенное на рис. 34.1.

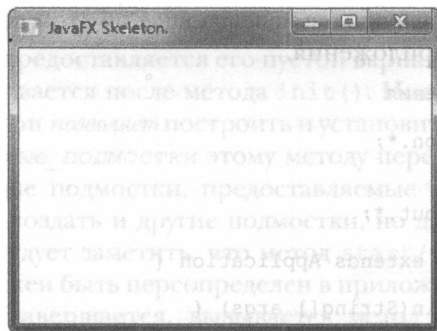


Рис. 34.1. Окно скелета JavaFX-приложения

Кроме того, при выполнении данного скелета JavaFX-приложения на консоль выводятся следующие сообщения:

```

Запуск JavaFX-приложения.
В теле метода init().
В теле метода start().
В теле метода stop().

```

При закрытии окна данного скелета JavaFX-приложения на консоль выводится следующее сообщение:

```

В теле метода stop().

```

Разумеется, в реальном JavaFX-приложении методы его жизненного цикла обычно не направляют никаких сообщений в стандартный поток вывода `System.out`. Но в данном скелете это делается ради того, чтобы показать, когда именно вызывается каждый метод жизненного цикла JavaFX-приложения. Более того, методы `init()` и `stop()` придется, как пояснялось ранее, переопределить только в том случае, если приложение должно выполнить специальные действия для запуска и закрытия. В противном случае можно воспользоваться реализациями этих методов, предоставляемыми по умолчанию в классе `Application`.

Рассмотрим данный скелет JavaFX-приложения более подробно. Сначала в нем импортируются четыре пакета. Первым из них является пакет `javafx.application`,

содержащий класс `Application`; вторым — пакет `javafx.scene`, содержащий класс `Scene`; третьим — пакет `javafx.stage`, содержащий класс `Stage`; четвертым — пакет `javafx.scene.layout`, предоставляющий ряд панелей компоновки. В данном случае используется панель поточной компоновки типа `FlowPane`.

Далее создается класс приложения `JavaFXSkel`, расширяющий класс `Application`. Как пояснялось ранее, от класса `Application` происходят классы всех JavaFX-приложений. Класс `JavaFXSkel` содержит два метода. Первый из них называется `main()` и служит для запуска приложения через метод `launch()`. Обратите внимание на то, что параметр `args` передается не только методу `main()`, но и методу `launch()`. И хотя это типичный подход, методу `launch()` можно передать и другие параметры или вообще не передавать их. И, как пояснялось ранее, метод `launch()` требуется только для автономных приложений. Если же он не нужен, то ненужным оказывается и метод `main()`. Но по упоминавшимся ранее причинам оба метода, `main()` и `launch()`, включаются в примеры JavaFX-приложений, представленные в части IV данной книги.

Когда приложение начинает выполняться, первым из исполняющей системы JavaFX вызывается метод `init()`. Ради большей наглядности примера этот метод просто направляет сообщение в стандартный поток вывода `System.out`, но, как правило, он служит для инициализации некоторых свойств приложения. Разумеется, если инициализация не требуется, то и переопределять метод `init()` не нужно, поскольку по умолчанию предоставляется его пустая реализация. Следует особо подчеркнуть, что с помощью метода `init()` нельзя создать подмости или сцену. Эти элементы ГПИ следует создавать и отображать с помощью метода `start()`.

По завершении метода `init()` начинает выполняться метод `start()`. Именно в нем и создается первоначальная сцена и устанавливаются главные подмости. Проанализируем этот метод построчно. Прежде всего следует заметить, что у метода `start()` имеется параметр типа `Stage`. Когда метод `start()` вызывается, этот параметр получает ссылку на главные подмости приложения, где и устанавливается сцена для приложения.

После вывода на консоль сообщения о том, что метод `start()` начал выполняться, вызывается метод `setTitle()`, чтобы задать заголовок сцены, как показано ниже.

```
myStage.setTitle("JavaFX Skeleton.");
```

Эта операция характерна для автономных приложений, хотя и не является обязательной. Указанный заголовок становится заголовком главного окна приложения.

Далее создается корневой узел сцены. Это единственный узел графа сцены, у которого отсутствует родитель. В данном случае в качестве корневого узла служит панель поточной компоновки типа `FlowPane`, как показано ниже. Хотя в корневом узле могут быть использованы и другие классы.

```
FlowPane rootNode = new FlowPane();
```

Как упоминалось ранее, класс `FlowPane` предоставляет диспетчер поточной компоновки, где элементы располагаются построчно с автоматическим переходом на новую строку, если требуется. Следовательно, этот класс действует аналогично классу `FlowLayout` из библиотек AWT и Swing. В данном случае применяется поточная компоновка по горизонтали, хотя можно указать и поточную компоновку по вертикали. Имеется возможность указать и другие свойства компоновки, в том

числе промежутки между элементами по горизонтали и по вертикали, а также выравнивание, хотя в данном скелетном приложении этого не требуется. Далее в этой главе будет показано, как устанавливаются свойства компоновки.

В следующей строке кода корневой узел используется для построения сцены в виде объекта типа Scene:

```
Scene myScene = new Scene(rootNode, 300, 200);
```

В классе Scene предоставляется несколько форм конструктора. В той форме конструктора, которая применяется здесь и далее в этой главе, создается сцена с указанным корневым узлом и заданной шириной и высотой. Ниже приведена данная форма варианта конструктора класса Scene.

Scene(Parent *корневой_узел*, double *ширина*, double *высота*)

Как видите, параметр *корневой_узел* относится к типу Parent. Это класс, производный от класса Node и инкапсулирующий узлы, у которых могут быть потомки. А параметры *ширина* и *высота* относятся к типу double. Это дает возможность передавать данному конструктору дробные числовые значения ширины и высоты сцены. В данном скелете JavaFX-приложения указан корневой узел rootNode и задана ширина 300 и высота 200 сцены.

В приведенной ниже строке кода устанавливается сцена myScene на подмостках myStage. Для этого вызывается метод setScene(), определяемый в классе Stage и устанавливающий сцену, указываемую в качестве его аргумента.

```
myStage.setScene(myScene);
```

В тех случаях, когда сцена больше не используется, обе предыдущие операции можно объединить в одну, как показано ниже. Благодаря своей компактности эта форма используется в большинстве представленных далее примеров.

```
myStage.setScene(new Scene(rootNode, 300, 200));
```

И в последней, приведенной ниже строке кода из метода start() отображаются подмостки и сцена. По существу, метод show() отображает окно, созданное на подмостках и сцене.

```
myStage.show();
```

Когда приложение закрывается, его окно удаляется с экрана, а из исполняющей системы JavaFX вызывается метод stop(). В данном случае метод stop() просто выводит на консоль сообщение о том, что он начал выполняться. Но, как правило, этот метод ничего не выводит на экран. Более того, если в приложении не требуется выполнять никаких закрывающих действий, то и переопределять метод stop() не нужно, поскольку по умолчанию предоставляется его пустая реализация.

Компиляция и выполнение JavaFX-приложения

К числу самых главных преимуществ JavaFX относится возможность выполнять одну и ту же прикладную программу в различных исполняющих средах. В частности, ее можно выполнить как автономное настольное приложение, в окне браузе-

ра или же как приложение типа Web Start. Но иногда прикладной программе могут потребоваться различные вспомогательные файлы, в том числе HTML-файл или файл формата JNLP (Java Network Launch Protocol — сетевой протокол запуска приложений на Java).

В общем, JavaFX-приложение компилируется аналогично любой другой программе, написанной на Java. Но поскольку ему требуется дополнительная поддержка для различных исполняющих сред, то его проще всего скомпилировать в интегрированной среде разработки (ИСР), полностью поддерживающей программирование средствами JavaFX, например, в NetBeans. Для этого достаточно следовать инструкциям в применяемой ИСР.

С другой стороны, если требуется скомпилировать и проверить JavaFX-приложение, используя инструментальные средства в режиме командной строки, то и это сделать совсем не трудно. Для этого достаточно скомпилировать и выполнить приложение обычным образом по командам `javac` и `java` соответственно. Следует, однако, иметь в виду, что компилятор, работающий в режиме командной строки, не создает ни архивный JAR-файл приложения, ни файлы формата HTML или JNLP, которые могут потребоваться, если приложение необходимо выполнить не как автономное, а как-то иначе. Для создания этих типов файлов придется воспользоваться такой утилитой, как `javafxpackager`.

Поток исполнения приложения

Как упоминалось ранее, метод `init()` нельзя использовать для построения подмостков или сцены. Эти элементы ГПИ нельзя создать и в конструкторе класса приложения, потому что подмостки или сцена должны строиться в *потоке исполнения приложения*. Но конструктор класса приложения и метод `init()` вызываются в главном потоке исполнения, иначе называемом *запускающим потоком исполнения*. Следовательно, их нельзя использовать для построения подмостков или сцены. Вместо этого нужно воспользоваться методом `start()`, чтобы построить первоначальный ГПИ, как продемонстрировано в рассмотренном выше скелете JavaFX-приложения, поскольку метод `start()` вызывается в потоке исполнения приложения.

Более того, любые изменения в ГПИ, отображаемом в данный момент, должны производиться из потока исполнения приложения. Правда, в JavaFX события посылаются прикладной программе в потоке исполнения приложения. Поэтому для взаимодействия с ГПИ могут быть использованы обработчики событий. Метод `stop()` также вызывается в потоке исполнения приложения.

Метка — простейший элемент управления в JavaFX

Главным компонентом большинства ГПИ является элемент управления, поскольку он дает приложению возможность взаимодействовать с пользователем. Как и следовало ожидать, в JavaFX предоставляется богатый арсенал элементов

управления. Простейшим элементом управления является метка, поскольку она только отображает сообщение, например текстовое. Несмотря на всю свою простоту, метка служит удобным примером для демонстрации приемов, которыми нужно овладеть, чтобы приступить к построению графа сцены.

В JavaFX метка представлена экземпляром класса `Label`, входящего в состав пакета `javafx.scene.control`. Класс `Label` наследует, среди прочих, от классов `Labeled` и `Control`. В классе `Labeled` определяется ряд средств, общих для всех отмечаемых меткой элементов, т.е. тех, что могут содержать текст, а в классе `Control` — средства, связанные со всеми элементами управления.

В классе `Label` определяются три конструктора. Ниже приведен конструктор, применяемый здесь и далее в этой главе, где параметр *строка* обозначает символьную строку, отображаемую на месте метки.

Label(String строка)

Как только метка будет создана, она должна быть введена в содержимое сцены, т.е. в граф сцены. Это же относится и к любому другому элементу управления. С этой целью сначала вызывается метод `getChildren()` для корневого узла в графе сцены. Этот метод возвращает список порожденных узлов в форме `ObservableList<Node>`. Класс `ObservableList` входит в состав пакета `javafx.collections` и наследует от класса `java.util.List`, а следовательно, поддерживает все средства, доступные для составления списка и определенные в каркасе коллекций `Collections Framework`. В возвращаемый список порожденных узлов можно ввести метку, вызвав метод `add()` и передав ему ссылку на метку.

Все сказанное выше демонстрируется на практике в следующем примере, где создается JavaFX-приложение, отображающее метку:

// Продемонстрировать применение элемента управления меткой в JavaFX

```
import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;

public class JavaFXLabelDemo extends Application {

    public static void main(String[] args) {

        // Start the JavaFX application by calling launch().
        launch(args);
    }

    // переопределить метод start()
    public void start(Stage myStage) {

        // присвоить заголовок подмосткам
        myStage.setTitle("Demonstrate a JavaFX label.");
        // Продемонстрировать метку в JavaFX

        // использовать панель типа FlowPane в качестве корневого узла
        FlowPane rootNode = new FlowPane();

        // создать сцену
        Scene myScene = new Scene(rootNode, 300, 200);
```

```
// установить сцену на подмостках
myStage.setScene(myScene);

// создать метку
Label myLabel = new Label("This is a JavaFX label");
// Это метка в JavaFX

// ввести метку в граф сцены
rootNode.getChildren().add(myLabel);

// показать подмостки и сцену на них
myStage.show();
}
```

На рис. 34.2 показано окно данного JavaFX-приложения с отображаемой меткой.

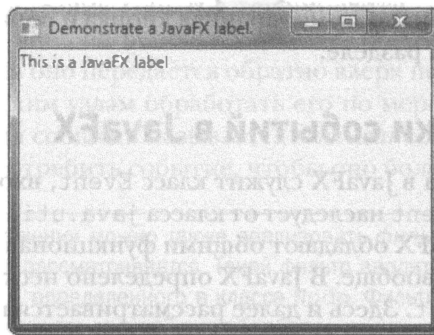


Рис. 34.2. Окно JavaFX-приложения с отображаемой меткой

В данном JavaFX-приложении особое внимание обращает на себя следующая строка кода:

```
rootNode.getChildren().add(myLabel);
```

В этой строке кода метка вводится в список порожденных узлов, родителем которых является корневой узел `rootNode`. И хотя эту строку кода можно, если потребуется, разделить на отдельные части, зачастую она записывается указанным выше способом.

Прежде чем продолжить дальше, следует заметить, что в классе `ObservableList` предоставляется метод `addAll()`, с помощью которого можно ввести в граф сцены сразу два и больше порожденных узлов, как будет показано в приведенном далее примере. Чтобы удалить элемент управления из графа сцены, достаточно вызвать метод `remove()` для объекта типа `ObservableList`. Так, в приведенной ниже строке кода метка `myLabel` удаляется со сцены.

```
rootNode.getChildren().remove(myLabel);
```

Применение кнопок и событий

Несмотря на то что в предыдущем примере JavaFX-приложения демонстрировалось применение простого элемента управления и построение графа сцены,

в нем не было показано, каким образом обрабатываются события. Как вам должно быть уже известно, большинство элементов управления ГПИ генерируют события, обрабатываемые в прикладных программах. Например, кнопки, флажки и списки — все эти элементы управления генерируют события, когда ими манипулирует пользователь. Во многих отношениях обработка событий в JavaFX организуется таким же образом, как и в Swing или AWT, но только она более рационализирована. Следовательно, если вы приобрели практические навыки обработки событий в ГПИ на основе библиотек Swing и AWT, то вам не составит особого труда овладеть системой обработки событий, предоставляемой в JavaFX.

К числу наиболее употребительных элементов управления относится экранная кнопка, поэтому события от кнопок обрабатываются чаще всего. И это обстоятельство делает кнопку отличным примером для демонстрации основ обработки событий в JavaFX. Именно поэтому экранная кнопка и обработка событий рассматриваются вместе в этом разделе.

Основы обработки событий в JavaFX

Базовым для событий в JavaFX служит класс `Event`, входящий в состав пакета `javafx.event`. Класс `Event` наследует от класса `java.util.EventObject`, а следовательно, события в JavaFX обладают общими функциональными возможностями для всех событий в Java вообще. В JavaFX определено несколько подклассов, производных от класса `Event`. Здесь и далее рассматривается подкласс `ActionEvent`, предназначенный для обработки событий, генерируемых экранной кнопкой.

В общем, для обработки событий в JavaFX применяется модель делегирования событий. Чтобы обработать событие, нужно сначала зарегистрировать его обработчик, действующий как приемник данного события. Когда наступает событие, вызывается его приемник, который должен отреагировать на событие и выполнить возврат. В этом отношении управление событиями в JavaFX осуществляется таким же образом, как, например, в Swing.

Для обработки событий следует реализовать интерфейс `EventHandler`, который также входит в состав пакета `javafx.event`. Это обобщенный интерфейс, объявляемый следующим образом:

```
interface EventHandler<T extends Event>
```

где параметр `T` обозначает тип события, обрабатываемого соответствующим обработчиком. В данном интерфейсе определяется единственный метод `handle()`, принимающий в качестве параметра объект события. Ниже приведена общая форма этого метода.

```
void handle(T объект_события)
```

Здесь параметр `объект_события` обозначает наступившее событие. Как правило, обработчики событий реализуются в анонимных внутренних классах или лямбда-выражениях, но для этой цели подходят и автономные классы, если они более пригодны для приложения. (Например, в том случае, если один обработчик должен обрабатывать события из нескольких источников.)

Иногда полезно знать источник события, хотя в представленных далее примерах этого не требуется. Это особенно важно, если один обработчик должен обрабатывать события из разных источников. Чтобы получить источник события, достаточно вызвать метод `getSource()`, наследуемый из класса `java.util.EventObject`. Ниже приведена общая форма этого метода.

```
Object getSource()
```

Остальные методы из класса `Event` позволяют получить тип события, выяснить, было ли событие употреблено, употребить или сгенерировать событие, а также получить адресат события. Как только событие будет употреблено, его передача родительскому обработчику прекращается.

И последнее замечание: в JavaFX обработка событий выполняется по *цепочке диспетчеризации событий*. Когда событие генерируется, оно передается сначала корневому узлу, а затем вниз по цепочке адресату события. После обработки события в узле его адресата оно передается обратно вверх по цепочке, предоставляя возможность родительским узлам обработать его по мере надобности. Такой механизм распространения событий называется *всплыванием событий*. Он позволяет любому узлу цепочки употребить событие, чтобы оно больше не обрабатывалось.

На заметку! В JavaFX-приложении можно также реализовать *фильтр событий* для управления ими, хотя здесь он не рассматривается. Такой фильтр вводится в узел с помощью метода `addEventFilter()`, определенного в классе `Node`. Фильтр может употребить событие, предотвратив тем самым его дальнейшую обработку.

Элемент управления экранной кнопкой

В JavaFX элемент управления экранной кнопкой предоставляется в классе `Button`, входящем в состав пакета `javafx.scene.control`. Класс `Button` наследует довольно обширный перечень базовых классов, включая `ButtonBase`, `Labeled`, `Region`, `Control`, `Parent` и `Node`. Как следует из документации на класс `Button`, большая часть его функциональных возможностей наследуется из базовых классов. Кроме того, в этом классе поддерживается большое разнообразие массивов. Но здесь рассматривается его форма, используемая по умолчанию. Экранные кнопки могут содержать текстовую надпись, кнопку или и то и другое. В примерах из этой главы применяются кнопки с текстовой надписью, а пример кнопки с графикой приводится в следующей главе.

В классе `Button` определяются три конструктора. Ниже приведен конструктор, применяемый здесь и далее в этой главе, где параметр *строка* обозначает текстовую надпись на кнопке.

```
Button(String строка)
```

После щелчка на экранной кнопке генерируется событие типа `ActionEvent`. Класс `ActionEvent` входит в состав пакета `javafx.event`. Вызвав метод `setOnAction()`, можно зарегистрировать приемник данного события. Ниже приведена общая форма этого метода.

```
final void setOnAction(EventHandler<ActionEvent> обработчик)
```

Здесь параметр *обработчик* обозначает регистрируемый обработчик событий указанного типа. Как упоминалось ранее, для реализации обработчика событий нередко применяется анонимный внутренний класс или лямбда-выражение. Метод `setOnAction()` устанавливает свойство `onAction`, сохраняющее ссылку на обработчик событий. Как и при обработке всех остальных событий в Java, такой обработчик событий должен как можно быстрее реагировать на событие и выполнять возврат. Если же обработчик событий делает это слишком медленно, то тем самым он заметно тормозит работу приложения. Поэтому для длительных операций следует предусмотреть отдельный поток исполнения.

Демонстрация обработки событий на примере экранных кнопок

В приведенном ниже примере JavaFX-приложения обработка событий демонстрируется на примере экранных кнопок. В этом приложении используются две такие кнопки и одна метка. Всякий раз, когда нажимается экранная кнопка, на месте метки выводится сообщение, извещающее, какая именно кнопка была нажата.

```
// Продемонстрировать применение экранных кнопок
// и обработку событий в JavaFX

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class JavaFXEventDemo extends Application {

    Label response;

    public static void main(String[] args) {

        // запустить JavaFX-приложение, вызвав метод launch()
        launch(args);
    }

    // переопределить метод start()
    public void start(Stage myStage) {

        // присвоить заголовок подмонокам
        myStage.setTitle("Demonstrate JavaFX Buttons and Events.");
        // Продемонстрировать кнопки и события в JavaFX

        // Использовать панель поточной компоновки FlowPane
        // в качестве корневого узла. Установить промежутки
        // между элементами управления по горизонтали и
        // по вертикали равными 10
        FlowPane rootNode = new FlowPane(10, 10);

        // выровнять элементы управления по центру сцены
        rootNode.setAlignment(Pos.CENTER);
```

```

// создать сцену
Scene myScene = new Scene(rootNode, 300, 100);

// установить сцену на подмостках
myStage.setScene(myScene);

// создать метку
response = new Label("Push a Button"); // Нажать кнопку

// создать две экранные кнопки
Button btnAlpha = new Button("Alpha");
Button btnBeta = new Button("Beta");

// обработать события действия от кнопки Alpha
btnAlpha.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        response.setText("Alpha was pressed.");
        // Нажата кнопка Alpha
    }
});

// обработать события действия от кнопки Beta
btnBeta.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        response.setText("Beta was pressed.");
        // Нажата кнопка Beta
    }
});

// ввести метку и кнопки в граф сцены
rootNode.getChildren().addAll(btnAlpha, btnBeta, response);

// показать подмостки и сцену на них
myStage.show();
}
}

```

На рис. 34.3 приведен примерный результат выполнения JavaFX-приложения, демонстрирующего обработку событий от экранных кнопок.

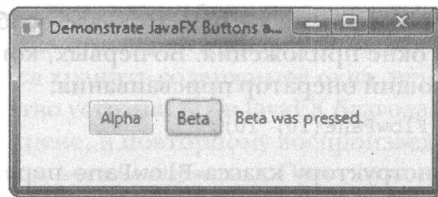


Рис. 34.3. Результат нажатия одной из экранных кнопок в окне приложения JavaFXEventDemo

Рассмотрим данное JavaFX-приложение по некоторым из его главных частей. Прежде всего обратите внимание на приведенные ниже строки кода, в которых создаются экранные кнопки.

```

Button btnAlpha = new Button("Alpha");
Button btnBeta = new Button("Beta");

```


В этих строках кода создаются две экранные кнопки с текстовыми надписями. Первая из них содержит надпись **Alpha**, вторая – надпись **Beta**.

Далее для каждой экранной кнопки задается обработчик событий действия. Ниже показано, как это делается для кнопки **Alpha**.

```
// обработать события действия от кнопки Alpha
btnAlpha.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        response.setText("Alpha was pressed.");
        // Нажата кнопка Alpha
    }
});
```

Как пояснялось ранее, экранные кнопки реагируют на события типа `ActionEvent`. Чтобы зарегистрировать обработчик этих событий от конкретной кнопки, вызывается метод `setOnAction()`, в котором используется анонимный внутренний класс для реализации интерфейса `EventHandler`. (Напомним, что в интерфейсе `EventHandler` определяется единственный метод `handle()`.) В методе `handle()` задается текст, выводимый на месте метки `response`, чтобы отразить факт нажатия экранной кнопки **Alpha**. С этой целью для данной метки вызывается метод `setText()`. Аналогичным образом обрабатываются события от кнопки **Beta**.

Обратите внимание на то, что метка `response` объявляется как поле в классе приложения `JavaFXEventDemo`, а не как локальная переменная. Это делается для того, чтобы она была доступна для обработчиков событий от экранных кнопок, реализуемых в виде анонимных внутренних классов.

После установки обработчиков событий метка `response` и экранные кнопки `btnAlpha` и `btnBeta` вводятся в граф сцены. С этой целью вызывается метод `addAll()`:

```
rootNode.getChildren().addAll(btnAlpha, btnBeta, response);
```

Метод `addAll()` вводит список узлов в вызывающий корневой узел. Безусловно, эти узлы можно было бы ввести и по отдельности, вызвав метод `add()`, но в данном случае удобнее воспользоваться методом `addAll()`.

В данном примере интересно также отметить две особенности отображения элементов управления в окне приложения. Во-первых, когда создается корневой узел, применяется следующий оператор присваивания:

```
FlowPane rootNode = new FlowPane(10, 10);
```

В этом операторе конструктору класса `FlowPane` передаются два параметра, обозначающих промежутки между элементами управления, оставляемые в сцене по горизонтали и по вертикали. Если эти промежутки не указаны, элементы управления (в данном случае две экранные кнопки) располагаются вплотную друг к другу. Следовательно, пользоваться ими будет неудобно, а сам пользовательский интерфейс приобретет неприглядный вид. Во избежание этого и указывается промежуток между элементами управления.

И во-вторых, элементы управления выравниваются по центру на панели поточной компоновки типа `FlowPane`, как показано ниже.

```
rootNode.setAlignment(Pos.CENTER);
```

Чтобы выровнять элементы управления по центру, вызывается метод `setAlignment()` для панели типа `FlowPane`, которая в данном случае служит корневым узлом. Значение константы `Pos.CENTER` обозначает центровку как по вертикали, так и по горизонтали, хотя можно указать и другие стили выравнивания. Класс `Pos` представляет собой перечисление, в котором определяются константы выравнивания. Этот класс входит в состав пакета `javafx.geometry`.

Прежде чем продолжить дальше, следует заметить, что в рассматриваемом здесь примере для обработки событий используются анонимные внутренние классы. Но поскольку в интерфейсе `EventHandler` определяется единственный абстрактный метод `handle()`, то вместо него методу `setOnAction()` можно передать лямбда-выражение. В данном случае параметр типа передает методу `setOnAction()` целевой контекст лямбда-выражения. В качестве примера ниже приведен обработчик событий от кнопки `Alpha`, переписанный в виде лямбда-выражения.

```
btnAlpha.setOnAction( (ae) -> response.setText("Alpha was pressed.") );
```

Как видите, лямбда-выражение выглядит более компактно, чем анонимный внутренний класс. Но в последующих примерах все же применяются анонимные внутренние классы, поскольку лямбда-выражения внедрены лишь недавно в Java, тогда как анонимные внутренние классы уже давно нашли широкое применение и хорошо известны практически всем программирующим на Java. Это позволяет также компилировать примеры программ, используя версию JDK 7, где лямбда-выражения не поддерживаются. Хотя вы можете самостоятельно поэкспериментировать с лямбда-выражениями, чтобы приобрести практические навыки их применения в своем коде.

Рисование непосредственно на холсте

Как упоминалось ранее, задачи воспроизведения решаются в JavaFX автоматически, а не вручную. В этом состоит одно из главных усовершенствований JavaFX по сравнению с Swing. Как вам должно быть уже известно, в библиотеке Swing или AWT для перерисовки окна приходится вызывать метод `repaint()`. Более того, в приложении приходится хранить содержимое окна, перерисовывая его по мере надобности. Это неудобство устраняется в JavaFX благодаря слежению за тем, что требуется отображать в сцене, и повторному воспроизведению сцены по мере надобности в так называемом *режиме удержания*. При таком подходе отпадает потребность вызывать метод `repaint()`, поскольку повторное воспроизведение выполняется автоматически.

Такой подход оказывается особенно полезным при воспроизведении таких графических объектов, как линии, окружности и прямоугольники. Методы, применяемые в JavaFX для воспроизведения графики, находятся в классе `GraphicsContext`, входящем в состав пакета `java.scene.canvas`. Эти методы могут быть использованы для рисования непосредственно на поверхности холста, инкапсулированного в классе `Canvas`, входящем в состав того же самого пакета. Когда на холсте рисуется какой-нибудь графический объект, например линия, он автоматически

воспроизводится средствами JavaFX всякий раз, когда требуется воспроизвести его повторно.

Прежде чем рисовать на холсте, следует выполнить два действия. Во-первых, создать холст в виде объекта типа `Canvas`. Во-вторых, получить графический контекст, ссылающийся на этот холст, в виде объекта типа `GraphicsContext`. Полученным в итоге графическим контекстом типа `GraphicsContext` можно затем воспользоваться для рисования выводимых графических данных на холсте.

Класс `Canvas` является производным от класса `Node`, а следовательно, им можно пользоваться как узлом в графе сцены. В классе `Canvas` определяются два конструктора. Одним из них является конструктор по умолчанию, а другой приведен ниже, где параметры *ширина* и *высота* обозначают соответствующие размеры холста.

```
Canvas(double ширина, double высота)
```

Чтобы получить графический контекст типа `GraphicsContext`, ссылающийся на холст, следует вызвать метод `getGraphicsContext2D()`. Ниже приведена общая форма этого метода. Он возвращает графический контекст для холста.

```
GraphicsContext getGraphicsContext2D()
```

В классе `GraphicsContext` определяется немало методов для рисования форм, воспроизведения текста и изображений, а также для поддержки визуальных эффектов и графических преобразований. Если вы связываете свое будущее с программированием сложной графики, вам определенно следует внимательно изучить возможности этого класса. В целях демонстрации далее будут использованы лишь некоторые методы из этого класса, хотя и они дают ясное представление об его огромном потенциале. Эти методы вкратце описываются далее.

Вызвав метод `strokeLine()`, можно нарисовать линию. Ниже приведена общая форма этого метода.

```
void strokeLine(double начало_X, double начало_Y,  
                 double конец_X, double конец_Y)
```

Этот метод рисует линию от точки с указанными координатами *начало_X*, *начало_Y* к точке с указанными координатами *конец_X*, *конец_Y*, используя текущую обводку, которая может быть выполнена сплошным цветом или оформлена в более сложном стиле.

Чтобы нарисовать прямоугольник, необходимо вызвать метод `strokeRect()` или `fillRect()`. Ниже приведены общие формы этих методов.

```
void strokeRect(double верх_X, double верх_Y,  
                 double ширина, double высота)  
void fillRect(double верх_X, double верх_Y,  
               double ширина, double высота)
```

Параметры координат *верх_X*, *верх_Y* обозначают левый верхний угол рисуемого прямоугольника, параметры *ширина* и *высота* — его ширину и высоту соответственно. Метод `strokeRect()` рисует контур прямоугольника, используя текущую обводку, а метод `fillRect()` заполняет площадь прямоугольника текущей заливкой: сплошным цветом или более сложным рисунком.

Чтобы нарисовать эллипс, следует вызвать метод `strokeOval()` или `fillOval()`. Ниже приведены общие формы этих методов.

```
void strokeOval(double верх_X, double верх_Y,
               double ширина, double высота)
void fillOval(double верх_X, double верх_Y,
              double ширина, double высота)
```

Параметры координат *верх_X*, *верх_Y* обозначают левый верхний угол рисуемого прямоугольника, а параметры *ширина* и *высота* — его ширину и высоту соответственно. Метод `strokeOval()` рисует контур эллипса, используя текущую обводку, а метод `fillOval()` заполняет площадь эллипса текущей заливкой: сплошным цветом или более сложным рисунком. Чтобы нарисовать окружность, методу `strokeOval()` следует передать одинаковые значения параметров *ширина* и *высота*.

Для воспроизведения текста на холсте служат методы `strokeText()` и `fillText()`. Здесь и далее в этой главе используется метод `fillText()`. Ниже приведена его общая форма. Этот метод воспроизводит указанную строку, начиная с точки с указанными координатами *верх_X*, *верх_Y* и заполняя текст текущей заливкой.

```
void fillText(String строка, double верх_X, double верх_Y)
```

Вызвав метод `setFont()`, можно задать тип и размер шрифта, которым должен воспроизводиться текст, а вызвав метод `getFont()` — получить шрифт, используемый на холсте. По умолчанию используется системный шрифт. Построив объект типа `Font`, можно также создать новый шрифт. Класс `Font` входит в состав пакета `javafx.scene.text`. Например, используя приведенный ниже конструктор данного класса, можно создать выбираемый по умолчанию шрифт указанного размера, где параметр *размер_шрифта* обозначает требуемый размер нового шрифта.

```
Font(double размер_шрифта)
```

Используя приведенные ниже методы из класса `Canvas`, можно указать заливку и обводку соответственно.

```
void setFill(Paint новая_заливка)
void setStroke(Paint новая_обводка)
```

Как видите, параметр обоих этих методов относится к типу `Paint`. Класс `Paint` является абстрактным и входит в состав пакета `javafx.scene.paint`. В его подклассах определяются конкретные разновидности заливки и обводки. Здесь и далее в этой главе применяется подкласс `Color`, который просто описывает сплошной цвет. В классе `Color` определяется ряд статических полей для обозначения широкого спектра цветов, например `Color.BLUE`, `Color.RED`, `Color.GREEN` и т.д.

В приведенном ниже примере JavaFX-приложения демонстрируется рисование на холсте описанными выше методами. Сначала в данном приложении воспроизводятся три графических объекта на холсте. Затем цвет этих объектов изменяется всякий раз, когда нажимается кнопка `Change Color` (Изменить цвет). Запустив это приложение на выполнение, вы обнаружите, что формы, цвет которых не меняется, не оказывают никакого влияния на изменение цвета остальных графических объектов. Более того, если попытаться скрыть окно данного приложения, а затем раскрыть его, содержимое холста будет автоматически перерисовано без участия

самого приложения. На рис. 34.4 приведен пример рисования графики на холсте в окне JavaFX-приложения из данного примера.

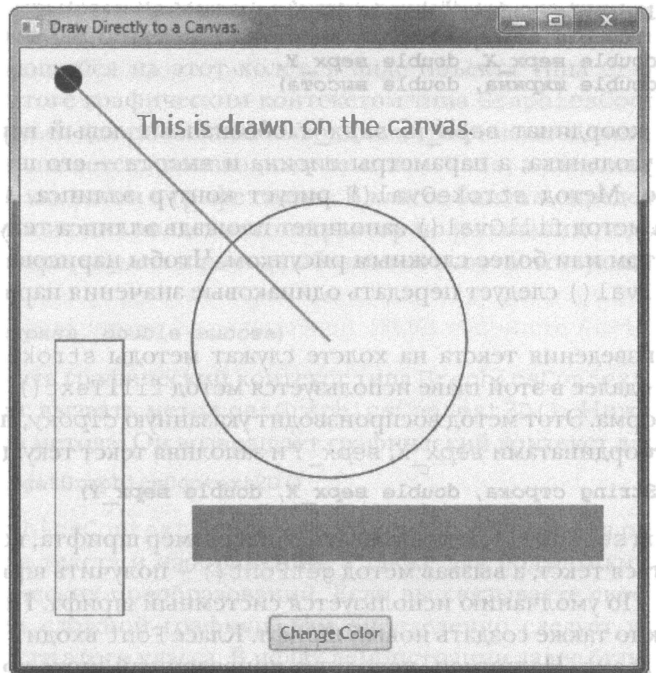


Рис. 34.4. Пример рисования графики на холсте в окне приложения DirectDrawDemo

// Продемонстрировать рисование на холсте

```
import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;
import javafx.scene.shape.*;
import javafx.scene.canvas.*;
import javafx.scene.paint.*;
import javafx.scene.text.*;

public class DirectDrawDemo extends Application {

    GraphicsContext gc;
    Color[] colors = { Color.RED, Color.BLUE, Color.GREEN, Color.BLACK };
    int colorIdx = 0;

    public static void main(String[] args) {

        // запустить JavaFX-приложение, вызвав метод launch()
        launch(args);
    }
}
```

```
// переопределить метод start()
public void start(Stage myStage) {

    // присвоить заголовок подмосткам
    myStage.setTitle("Draw Directly to a Canvas.");
    // Рисование прямо на холсте

    // Использовать панель поточной компоновки FlowPane
    // в качестве корневого узла
    FlowPane rootNode = new FlowPane();

    // расположить узлы по центру сцены
    rootNode.setAlignment(Pos.CENTER);

    // создать сцену
    Scene myScene = new Scene(rootNode, 450, 450);

    // установить сцену на подмостках
    myStage.setScene(myScene);

    // создать холст
    Canvas myCanvas = new Canvas(400, 400);

    // получить графический контекст для холста
    gc = myCanvas.getGraphicsContext2D();

    // создать экранную кнопку
    Button btnChangeColor = new Button("Change Color");

    // обработать события действия от кнопки Change Color
    btnChangeColor.setOnAction(new EventHandler<ActionEvent>() {
        public void handle(ActionEvent ae) {

            // задать цвет обводки и заливки
            gc.setStroke(colors[colorIdx]);
            gc.setFill(colors[colorIdx]);

            // Перерисовать линию, текст и заполненный прямоугольник
            // новым цветом. При этом цвет остальных узлов графа сцены
            // останется без изменения
            gc.strokeLine(0, 0, 200, 200);
            gc.fillText("This is drawn on the canvas.", 60, 50);
            // Это рисуется на холсте
            gc.fillRect(100, 320, 300, 40);

            // изменить цвет
            colorIdx++;
            if(colorIdx == colors.length) colorIdx= 0;
        }
    });

    // нарисовать на холсте графические объекты
    // первоначально выводимые на экран
    gc.strokeLine(0, 0, 200, 200);
    gc.strokeOval(100, 100, 200, 200);
    gc.strokeRect(0, 200, 50, 200);
    gc.fillOval(0, 0, 20, 20);
    gc.fillRect(100, 320, 300, 40);

    // задать шрифт размером 20 и воспроизвести текст
```

```
gc.setFont(new Font(20));
gc.fillText("This is drawn on the canvas.", 60, 50);

// ввести холст и кнопку в граф сцены
rootNode.getChildren().addAll(myCanvas, btnChangeColor);

// показать подмости и сцену на них
myStage.show();
}
}
```

Следует еще раз подчеркнуть, что в классе `GraphicsContext` поддерживается намного больше операций, чем те, что были продемонстрированы в приведенном выше примере JavaFX-приложения. В частности, к графическим объектам можно применять различные эффекты и выполнять над ними преобразования, в том числе вращение, масштабирование и искажение. Несмотря на обилие средств, доступных в этом классе, овладеть ими не так уж и трудно. Следует также иметь в виду, что холст прозрачен. Так, если расположить два холста один за другим, то видно будет содержимое обоих холстов. Иногда такое свойство прозрачности холстов может быть полезным.

На заметку! В состав пакета `javafx.scene.shape` входит несколько классов, которые можно также применять для рисования различных графических форм, в том числе окружностей, дуг и линий. Эти формы представлены узлами, а следовательно, они могут быть введены непосредственно в граф сцены. Рекомендуется изучить их самостоятельно.

Элементы управления JavaFX

В предыдущей главе были рассмотрены основные понятия, имеющие отношение к построению ГПИ средствами JavaFX. По ходу их описания были представлены два элемента управления: метка и кнопка. А в этой главе будет продолжено рассмотрение элементов управления JavaFX. В начале главы поясняется, каким образом метка и кнопка снабжаются изображениями. Затем дается краткий обзор некоторых других элементов управления JavaFX, в том числе флажков, списков и деревьев. Следует, однако, иметь в виду, что JavaFX — насыщенный функциональными возможностями и довольно мощный каркас. Поэтому основное назначение этой главы — представить наиболее употребительные элементы управления JavaFX и описать некоторые общие приемы их применения. Овладев основами, вы сможете без особого труда самостоятельно изучить остальные элементы управления JavaFX.

Ниже перечислены классы элементов управления JavaFX, обсуждаемые в этой главе. Классы этих и других элементов управления JavaFX входят в состав пакета `javafx.scene.control`.

Button	ListView	TextField
CheckBox	RadioButton	ToggleButton
Label	ScrollPane	TreeView

В этой главе обсуждаются также классы `Image` и `ImageView`, поддерживающие изображения в элементах управления; класс `Tooltip`, предназначенный для ввода всплывающих подсказок в элементы управления; а также различные визуальные эффекты и преобразования.

Классы `Image` и `ImageView`

Изображениями можно снабдить целый ряд элементов управления JavaFX. Так, помимо текстовой надписи, метку или кнопку можно снабдить изображением. Более того, изображения можно встраивать непосредственно в сцену как автономные графические объекты. Основу поддержки изображений в JavaFX составляют два класса: `Image` и `ImageView`. В частности, класс `Image` инкапсулирует само изображение, а класс `ImageView` управляет его воспроизведением. Оба эти класса входят в состав пакета `javafx.scene.image`.

Класс `Image` загружает изображение из потока ввода типа `InputStream`, веб-ресурса по указанному URL или по пути к файлу изображения. В классе `Image` определено несколько конструкторов. Ниже приведен конструктор, применяемый в примерах из этой главы.

`Image(String url)`

Здесь параметр `url` обозначает URL или путь к файлу с изображением. Если этот параметр не состоит из сформированного надлежащим образом URL, то предполагается, что он обозначает путь к файлу изображения. В противном случае изображение загружается по указанному URL. В представленных далее примерах изображения загружаются из файлов, находящихся в локальной файловой системе. Другие конструкторы класса `Image` позволяют указать ряд дополнительных параметров, в том числе ширину и высоту изображения. Следует также заметить, что класс `Image` является производным от класса `Node`. Следовательно, его объект может быть введен как узел в граф сцены.

Итак, получив изображение в виде объекта типа `Image`, можно воспроизвести его средствами класса `ImageView`. Класс `ImageView` является производным от класса `Node`, а следовательно, его объект может быть введен как узел в граф сцены. В классе `ImageView` определяются три конструктора. Ниже приведен конструктор, применяемый в примерах из этой главы. Этот конструктор создает объект типа `ImageView`, представляющий указанное изображение.

`ImageView(Image изображение)`

Все сказанное выше демонстрируется на примере JavaFX-приложения, загружающего изображение песочных часов и воспроизводящего его средствами класса `ImageView`. Изображение песочных часов хранится в файле `hourglass.png`, который, как предполагается, находится в локальном каталоге.

```
// Загрузить и воспроизвести изображение

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.geometry.*;
import javafx.scene.image.*;

public class ImageDemo extends Application {

    public static void main(String[] args) {

        // запустить JavaFX-приложение, вызвав метод launch()
        launch(args);
    }

    // переопределить метод start()
    public void start(Stage myStage) {

        // присвоить заголовок подмосткам
        myStage.setTitle("Display an Image");

        // Использовать панель поточной компоновки FlowPane
        // в качестве корневого узла
```

```
FlowPane rootNode = new FlowPane();

// выполнить выравнивание по центру
rootNode.setAlignment(Pos.CENTER);

// создать сцену
Scene myScene = new Scene(rootNode, 300, 200);

// установить сцену на подмостках
myStage.setScene(myScene);

// создать объект изображения
Image hourglass = new Image("hourglass.png");

// создать представление этого изображения
ImageView hourglassIV = new ImageView(hourglass);

// ввести изображение в граф сцены
rootNode.getChildren().add(hourglassIV);

// показать подмостки и сцену на них
myStage.show();
}
```

На рис. 35.1 показано изображение песочных часов, отображаемое в окне JavaFX-приложения из данного примера.

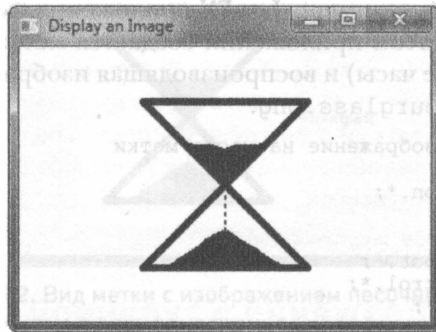


Рис. 35.1. Изображение песочных часов, отображаемое в окне приложения ImageDemo

Обратите в данном примере особое внимание на следующий фрагмент кода, в котором сначала загружается изображение, а затем создается его представление в виде объекта типа `ImageView`:

```
// создать объект изображения
Image hourglass = new Image("hourglass.png");

// создать представление этого изображения
ImageView hourglassIV = new ImageView(hourglass);
```

Как пояснялось ранее, само изображение нельзя ввести в граф сцены. Для этого его нужно сначала встроить в объект типа `ImageView`.

В тех случаях, когда изображение не используется в каких-нибудь других целях, при создании его представления в виде объекта типа `ImageView` можно указать

URL или имя файла. Это избавляет от необходимости создавать объект типа `Image` явным образом. Вместо этого экземпляр класса `Image`, содержащий изображение, получается автоматически и встраивается в объект типа `ImageView`. Для этой цели служит приведенный ниже конструктор класса `ImageView`, где параметр `url` обозначает URL или путь к файлу, содержащему изображение.

```
ImageView(String url)
```

Ввод изображения в метку

Как пояснялось в предыдущей главе, класс `Label` инкапсулирует метку. Он позволяет отображать на месте метки текстовое сообщение, графику или и то и другое. В приведенных ранее примерах на месте метки отображался только текст, но не составляет особого труда воспроизвести там же изображение. Для этой цели служит следующий конструктор класса `Label`:

```
Label(String строка, Node изображение)
```

где параметр *строка* обозначает отображаемое текстовое сообщение, а параметр *изображение* — воспроизводимое изображение. Это очень удобно для выбора типа изображения, вводимого в метку, но в целях демонстрации изображение будет далее представлено типом `ImageView`.

В приведенном ниже примере JavaFX-приложения демонстрируется метка, содержащая графику. В этом приложении создается метка, отображающая текст "Hourglass" (Песочные часы) и воспроизводящая изображение песочных часов, загружаемое из файла `hourglass.png`.

```
// Продемонстрировать изображение на месте метки
```

```
import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.geometry.*;
import javafx.scene.image.*;

public class LabelImageDemo extends Application {

    public static void main(String[] args) {

        // запустить JavaFX-приложение, вызвав метод launch()
        launch(args);
    }

    // переопределить метод start()
    public void start(Stage myStage) {

        // присвоить заголовок подмосткам
        myStage.setTitle("Use an Image in a Label");
        // Использовать изображение в метке

        // Использовать панель поточной компоновки FlowPane
        // в качестве корневого узла
        FlowPane rootNode = new FlowPane();

        // выполнить выравнивание по центру
```

```

rootNode.setAlignment(Pos.CENTER);

// создать сцену
Scene myScene = new Scene(rootNode, 300, 200);

// установить сцену на подмостках
myStage.setScene(myScene);

// создать представление указанного изображения
ImageView hourglassIV = new ImageView("hourglass.png");

// создать метку, содержащую изображение и текст
Label hourglassLabel = new Label("Hourglass", hourglassIV);

// ввести метку в граф сцены
rootNode.getChildren().add(hourglassLabel);

// показать подмостки и сцену на них
myStage.show();
}
}

```

На рис. 35.2 показано окно JavaFX-приложения из данного примера, в котором на месте метки отображается текст и воспроизводится изображение песочных часов.

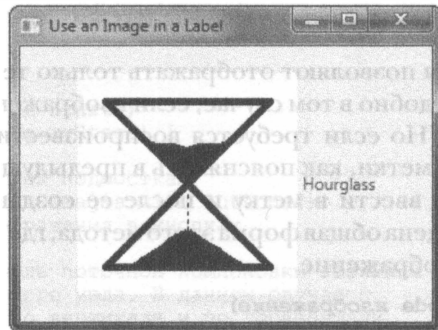


Рис. 35.2. Вид метки с изображением песочных часов и текстом в окне приложения `LabelImageDemo`

Как видите, на месте метки отображается не только текст, но и воспроизводится изображение. Обратите внимание на то, что текст воспроизводится справа от изображения. Это делается по умолчанию. Но относительное расположение изображения и текста можно изменить, вызвав метод `setContentDisplay()` для метки. Ниже приведена общая форма этого метода.

```
final void setContentDisplay(ContentDisplay позиция)
```

Значение, передаваемое в качестве параметра *позиция*, определяет порядок воспроизведения текста и изображения. Это должно быть значение одной из приведенных ниже констант, определяемых в перечислении типа `ContentDisplay`.

BOTTOM	RIGHT
CENTER	TEXT_ONLY
GRAPHIC_ONLY	TOP
LEFT	

Все перечисленные выше константы, кроме `TEXT_ONLY` и `GRAPHIC_ONLY`, обозначают местоположение изображения. Так, если ввести следующую строку кода в JavaFX-приложение из предыдущего примера:

```
hourglassLabel.setContentDisplay(ContentDisplay.TOP);
```

то изображение песочных часов появится над текстом, как показано на рис. 35.3.

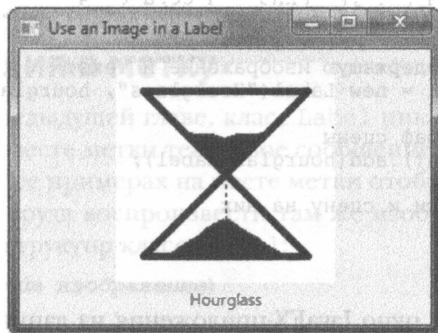


Рис. 35.3. Вид метки с изображением песочных часов над текстом

Две другие константы позволяют отображать только текст или только изображение. Это может быть удобно в том случае, если изображение применяется в приложении лишь иногда. (Но если требуется воспроизвести только изображение, это можно сделать и без метки, как пояснялось в предыдущем разделе.)

Изображение можно ввести в метку и после ее создания, вызвав метод `setGraphic()`. Ниже приведена общая форма этого метода, где параметр *изображение* обозначает вводимое изображение.

```
final void setGraphic(Node изображение)
```

Применение изображения в экранной кнопке

Класс `Button` служит в JavaFX для создания экранных кнопок. Этот класс был представлен в предыдущей главе, где приводился пример применения экранной кнопки с текстовой надписью. И хотя такие кнопки широко распространены, этим их применение в ГПИ не ограничивается, поскольку в них можно вводить не только текст, но и изображение, а если требуется, то лишь одно изображение. Процедура ввода изображения в экранную кнопку практически ничем не отличается от его ввода в метку. С этой целью сначала получается объект типа `ImageView`, представляющий изображение, а затем оно вводится в кнопку. Это можно сделать, например, с помощью следующего конструктора:

```
Button(String строка, Node изображение)
```

где параметр *строка* обозначает текст надписи на кнопке, а параметр *изображение* — воспроизводимое в ней изображение. Вызвав метод `setContentDisplay()`, можно указать относительное расположение текста надписи и изображения способом, описанным выше для метки.

В приведенном ниже примере в окне JavaFX-приложения отображаются две экранные кнопки, содержащие изображения песочных и аналоговых часов соответственно. При нажатии кнопки сообщается о выбранной разновидности часов. Обратите внимание на то, что текстовая надпись на каждой из кнопок отображается под соответствующим изображением часов.

```
// Применить изображение в кнопке

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;

import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;
import javafx.scene.image.*;

public class ButtonImageDemo extends Application {

    Label response;

    public static void main(String[] args) {

        // запустить JavaFX-приложение, вызвав метод launch()
        launch(args);
    }

    // переопределить метод start()
    public void start(Stage myStage) {

        // присвоить заголовок подмосткам
        myStage.setTitle("Use Images with Buttons");
        // Использовать изображения в кнопках

        // Использовать панель поточной компоновки FlowPane
        // в качестве корневого узла. В данном случае с
        // промежутками 10 по вертикали и по горизонтали
        FlowPane rootNode = new FlowPane(10, 10);

        // выровнять элементы управления по центру сцены
        rootNode.setAlignment(Pos.CENTER);

        // создать сцену
        Scene myScene = new Scene(rootNode, 250, 450);

        // установить сцену на подмостках
        myStage.setScene(myScene);

        // создать метку.
        response = new Label("Push a Button"); // Нажать кнопку

        // создать две экранные кнопки с текстовыми надписями
        // и соответствующими изображениями часов
        Button btnHourglass = new Button("Hourglass",
            new ImageView("hourglass.png"));
        Button btnAnalogClock = new Button("Analog Clock",
            new ImageView("analog.png"));

        // расположить текст под изображением
        btnHourglass.setContentDisplay(ContentDisplay.TOP);
        btnAnalogClock.setContentDisplay(ContentDisplay.TOP);
    }
}
```

```

// обработать события действия от экранной кнопки
// с изображением песочных часов
btnHourglass.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        response.setText("Hourglass Pressed");
        // Нажата кнопка с изображением песочных часов
    }
});

// обработать события действия от экранной кнопки
// с изображением аналоговых часов
btnAnalogClock.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        response.setText("Analog Clock Pressed");
        // Нажата кнопка с изображением аналоговых часов
    }
});

// ввести метку и кнопки в граф сцены
rootNode.getChildren().addAll(
    btnHourglass, btnAnalogClock, response);

// показать подмости и сцену на них
myStage.show();
}
}

```

На рис. 35.4 показан вид экранных кнопок с текстовыми надписями и соответствующими изображениями песочных часов в окне JavaFX-приложения из данного примера.

Если же требуется, чтобы кнопка содержала только изображение, вместо текстовой надписи ее конструктору следует передать пустую строку, а затем вызвать метод `setContentDisplay()`, передав ему в качестве параметра константу `ContentDisplay.GRAPHIC_ONLY`. Так, если внести подобные изменения в приложение из предыдущего примера, обе экранные кнопки будут выглядеть в его окне так, как показано на рис. 35.5.

Класс `ToggleButton`

Полезной разновидностью экранной кнопки является так называемый *переключатель*. Он похож на экранную кнопку, но действует иначе, поскольку может находиться только в одном из двух состояний: нажатом и отпущенном. Это означает, что если нажать переключатель, он останется в нажатом состоянии, а не перейдет после нажатия обратно в отпущенное состояние, как это обычно делает экранная кнопка. Если же нажать переключатель еще раз, он будет опущен. Следовательно, всякий раз, когда переключатель нажимается, он переходит в одно из двух своих состояний. В JavaFX переключатель инкапсулирован в классе `ToggleButton`. Как и класс `Button`, класс `ToggleButton` является производным от класса `ButtonBase`. Он реализует интерфейс `Toggle`, в котором определяются функциональные возможности, общие для всех типов кнопок с двумя состояниями.

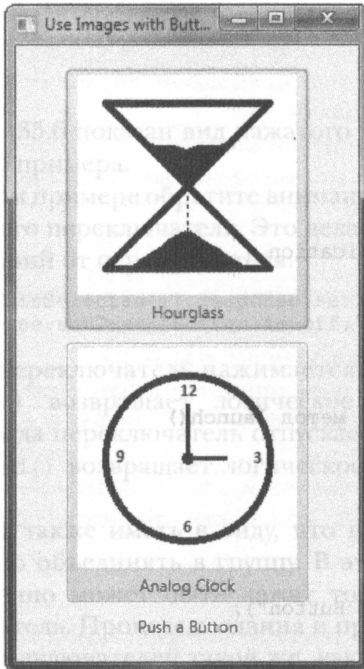


Рис. 35.4. Вид экранных кнопок с текстовыми надписями и соответствующими изображениями песочных часов в окне приложения `ButtonImageDemo`

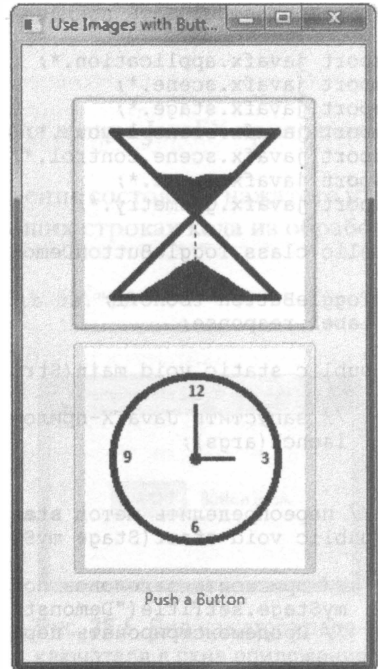


Рис. 35.5. Вид экранных кнопок только с изображениями часов в окне JavaFX-приложения

В классе `ToggleButton` определяются три конструктора. Здесь и далее применяется следующий конструктор:

`ToggleButton(String строка)`

где *строка* обозначает текстовую надпись на переключателе. Другой конструктор данного класса позволяет также ввести изображение в переключатель. Как и остальные разновидности кнопок, переключатель типа `ToggleButton` генерирует событие действия, когда он нажимается.

Итак, в классе `ToggleButton` определяется элемент управления с двумя состояниями, а следовательно, он зачастую применяется с целью предоставить пользователю возможность выбрать конкретный вариант. Когда же переключатель отпускается, то выбор данного варианта отменяется. Именно поэтому в прикладной программе обычно требуется определить состояние переключателя. С этой целью вызывается метод `isSelected()`, общая форма которого приведена ниже. Этот метод возвращает логическое значение `true`, если переключатель нажат, а иначе — логическое значение `false`.

`final boolean isSelected()`

В следующем кратком примере демонстрируется применение класса `ToggleButton`.


```
// Продемонстрировать применение переключателя

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class ToggleButtonDemo extends Application {

    ToggleButton tbOnOff;
    Label response;

    public static void main(String[] args) {

        // запустить JavaFX-приложение, вызвав метод launch()
        launch(args);
    }

    // переопределить метод start()
    public void start(Stage myStage) {

        // присвоить заголовок подмосткам
        myStage.setTitle("Demonstrate a Toggle Button");
        // Продемонстрировать переключатель

        // Использовать панель поточной компоновки FlowPane
        // в качестве корневого узла. В данном случае с
        // промежутками 10 по вертикали и по горизонтали
        FlowPane rootNode = new FlowPane(10, 10);

        // выровнять элементы управления по центру сцены
        rootNode.setAlignment(Pos.CENTER);

        // создать сцену
        Scene myScene = new Scene(rootNode, 220, 120);

        // установить сцену на подмостках
        myStage.setScene(myScene);

        // создать метку
        response = new Label("Push the Button."); // // Нажать кнопку

        // создать переключатель
        tbOnOff = new ToggleButton("On/Off"); // Включить/Выключить

        // обработать события действия от переключателя
        tbOnOff.setOnAction(new EventHandler<ActionEvent>() {
            public void handle(ActionEvent ae) {
                if(tbOnOff.isSelected()) response.setText("Button is on.");
                // Переключатель нажат
            else response.setText("Button is off.");
                // Переключатель отпущен
            }
        });

        // ввести метку и переключатель в граф сцены
        rootNode.getChildren().addAll(tbOnOff, response);
    }
}
```

```

    // показать подмостки и сцену на них
    myStage.show();
}
}

```

На рис. 35.6 показан вид нажатого переключателя в окне JavaFX-приложения из данного примера.

В данном примере обратите внимание на определение состояния нажатого или отпущенного переключателя. Это делается в следующих строках кода из обработчика действий от переключателя:

```

if(tbOnOff.isSelected()) response.setText("Button is on.");
else response.setText("Button is off.");

```

Когда переключатель нажимается, метод `isSelected()` возвращает логическое значение `true`. А когда переключатель отпускается, метод `isSelected()` возвращает логическое значение `false`.

Следует также иметь в виду, что переключатели можно объединять в группу. В этом случае одновременно может быть нажат только один переключатель. Процесс создания и применения группы переключателей такой же, как и для кнопок-переключателей. Он описывается в следующем разделе.

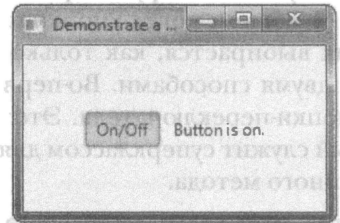


Рис. 35.6. Вид нажатого переключателя в окне приложения `ToggleButtonDemo`

Класс `RadioButton`

Еще одной разновидностью кнопок в JavaFX является *кнопка-переключатель*. Кнопки-переключатели образуют группу взаимоисключающих кнопок, где одновременно может быть выбрана только одна кнопка. Они поддерживаются в классе `RadioButton`, расширяющем классы `ButtonBase` и `ToggleButton`. Этот класс реализует также интерфейс `Toggle`. Следовательно, кнопка-переключатель является особой формой переключателя. Кнопки-переключатели должны быть вам уже знакомы, поскольку они относятся к числу основных элементов управления ГПИ, с помощью которых пользователь может выбрать один и только один вариант из нескольких возможных.

Для создания кнопки-переключателя здесь и далее применяется следующий конструктор класса `RadioButton`:

```
RadioButton(String строка)
```

где параметр *строка* обозначает метку кнопки-переключателя. Как и остальные разновидности кнопок, активизируемая кнопка-переключатель типа `RadioButton` генерирует исключение.

В силу своего взаимоисключающего характера кнопки-переключатели должны быть объединены в группы. При этом одновременно может быть выбрана лишь одна из них. Так, если пользователь щелкает на кнопке-переключателе в группе,

автоматически отменяется выбор любой кнопки-переключателя, нажатой ранее в этой группе. Группа кнопок-переключателей создается средствами класса `ToggleGroup`, входящего в состав пакета `javafx.scene.control`. В этом классе предоставляется только конструктор по умолчанию.

Для ввода кнопок-переключателей в группу служит метод `setToggleGroup()`, определенный в классе `ToggleButton`. Этот метод вызывается для отдельной кнопки-переключателя следующим образом:

```
final void setToggleGroup(ToggleGroup группа)
```

где параметр *группа* обозначает ту группу, в которую вводится данная кнопка-переключатель. Как только кнопки-переключатели будут введены в одну и ту же группу, активизируется режим их взаимоисключающего выбора.

В общем, когда кнопки-переключатели объединяются в одну группу, одна из них выбирается, как только их группа появляется в ГПИ. Этого можно добиться двумя способами. Во-первых, вызвать метод `setSelected()` для выбираемой кнопки-переключателя. Этот метод определяется в классе `ToggleButton`, который служит суперклассом для класса `RadioButton`. Ниже приведена общая форма данного метода.

```
final void setSelected(boolean состояние)
```

Если параметр *состояние* принимает логическое значение `true`, то кнопка-переключатель выбирается, в противном случае ее выбор отменяется. И хотя кнопка-переключатель выбирается, никакого события действия не генерируется.

И во-вторых, чтобы первоначально выбрать кнопку-переключатель, можно вызвать метод `fire()` для этой кнопки-переключателя. Ниже приведена общая форма данного метода. Метод `fire()` генерирует событие действия от данной кнопки-переключателя, если она не была выбрана ранее.

```
void fire()
```

Кнопки-переключатели можно применять самыми разными способами. И простейшим из них, вероятно, является реагирование на событие действия, генерируемое при нажатии кнопки-переключателя, как показано в приведенном ниже примере. В данном примере кнопки-переключатели предоставляют пользователю возможность выбрать тип транспортного средства.

```
// Простой пример, демонстрирующий применение кнопок-переключателей
//
// Данное JavaFX-приложение реагирует на события действия, генерируемые
// выбираемыми кнопками-переключателями. В нем демонстрируется также
// активизация кнопки-переключателя под управлением программы
```

```
import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;
```

```
public class RadioButtonDemo extends Application {
```

```
Label response;

public static void main(String[] args) {
    // запустить JavaFX-приложение, вызвав метод launch()
    launch(args);
}

// переопределить метод start()
public void start(Stage myStage) {

    // присвоить заголовок подмосткам
    myStage.setTitle("Demonstrate Radio Buttons");
    // Продемонстрировать кнопки-переключатели

    // Использовать панель поточной компоновки FlowPane
    // в качестве корневого узла. В данном случае с
    // промежутками 10 по вертикали и по горизонтали
    FlowPane rootNode = new FlowPane(10, 10);

    // выровнять элементы управления по центру сцены
    rootNode.setAlignment(Pos.CENTER);

    // создать сцену
    Scene myScene = new Scene(rootNode, 220, 120);

    // установить сцену на подмостках
    myStage.setScene(myScene);

    // создать метку, извещающую о сделанном выборе
    response = new Label("");

    // создать кнопки-переключатели
    RadioButton rbTrain = new RadioButton("Train"); // Поезд
    RadioButton rbCar = new RadioButton("Car"); // Автомобиль
    RadioButton rbPlane = new RadioButton("Airplane"); // Самолет

    // создать группу кнопок-переключателей
    ToggleGroup tg = new ToggleGroup();

    // ввести каждую кнопку-переключатель в группу
    rbTrain.setToggleGroup(tg);
    rbCar.setToggleGroup(tg);
    rbPlane.setToggleGroup(tg);

    // обработать события действия от кнопок-переключателей
    rbTrain.setOnAction(new EventHandler<ActionEvent>() {
        public void handle(ActionEvent ae) {
            response.setText("Transport selected is train.");
            // Выбранным транспортным средством является поезд
        }
    });

    rbCar.setOnAction(new EventHandler<ActionEvent>() {
        public void handle(ActionEvent ae) {
            response.setText("Transport selected is car.");
            // Выбранным транспортным средством является автомобиль
        }
    });

    rbPlane.setOnAction(new EventHandler<ActionEvent>() {
        public void handle(ActionEvent ae) {
```

```

        response.setText("Transport selected is airplane.");
        // Выбранным транспортным средством является самолет
    }
});

// Инициировать событие от первой выбранной кнопки-переключателя.
// В итоге кнопка-переключатель выбирается и наступает событие
// действия от этой кнопки-переключателя
rbTrain.fire();

// ввести метку и кнопки-переключатели в граф сцены
rootNode.getChildren().addAll(rbTrain, rbCar, rbPlane, response);

// показать подмости и сцену на них
myStage.show();
}
}

```

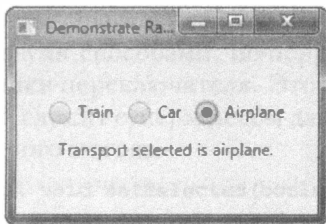


Рис. 35.7. Результат выбора одной из кнопк-переключателей в окне приложения `RadioButtonDemo`

На рис. 35.7 показан результат выбора одной из кнопк-переключателей в окне JavaFX-приложения из данного примера.

Обратите в данном примере особое внимание на порядок создания кнопк-переключателей и их объединения в группу. Сначала создаются кнопки-переключатели, как показано ниже.

```

RadioButton rbTrain = new RadioButton("Train"); // Поезд
RadioButton rbCar   = new RadioButton("Car");   // Автомобиль
RadioButton rbPlane = new RadioButton("Airplane"); // Самолет

```

Затем объект типа `ToggleGroup`, представляющий группу, создается следующим образом:

```
ToggleGroup tg = new ToggleGroup();
```

И наконец, каждая кнопка-переключатель вводится в группу приведенным выше образом. Как пояснялось ранее, кнопки-переключатели должны быть объединены в группу, чтобы активизировать режим их взаимоисключающего выбора.

```

rbTrain.setToggleGroup(tg);
rbCar.setToggleGroup(tg);
rbPlane.setToggleGroup(tg);

```

После определения обработчиков событий от каждой из кнопк-переключателей вызывается метод `fire()`, чтобы выбрать кнопку-переключатель `rbTrain`. В итоге данная кнопка-переключатель инициализируется как выбираемая по умолчанию.

Обработка событий изменения в группе кнопк-переключателей

Хотя в продемонстрированном выше способе управления кнопками-переключателями путем обработки событий действия нет ничего неверного, тем не ме-

нее иногда оказывается удобнее (и проще) принимать и обрабатывать события изменения во всей группе кнопок-переключателей в целом. Когда в такой группе происходит изменение, обработчик событий изменения может без особого труда определить, какая именно кнопка-переключатель была выбрана, и на этом основании предпринять соответствующее действие. Для этого нужно зарегистрировать приемник событий изменения типа `ChangeListener` в группе кнопок-переключателей. Когда же наступит событие изменения, можно определить, какая именно кнопка-переключатель была выбрана. Чтобы опробовать такой способ управления кнопками-переключателями, удалите сначала строки кода, в которых вводятся обработчики событий действия и вызывается метод `fire()`, из предыдущего примера JavaFX-приложения, а затем подставьте вместо них следующие строки кода:

```
// использовать приемник событий изменения, чтобы реагировать
// на изменения при выборе кнопок-переключателей из группы
tg.selectedToggleProperty().addListener(new ChangeListener<Toggle>() {
    public void changed(ObservableValue<? extends Toggle> changed,
                       Toggle oldVal, Toggle newVal) {

        // привести новое значение к типу RadioButton
        RadioButton rb = (RadioButton) newVal;

        // отобразить результат выбора
        response.setText("Transport selected is " + rb.getText());
        // Выбран указанный вид транспорта
    }
});

// выбрать первую кнопку-переключатель, чтобы инициировать
// событие изменения в группе
rbTrain.setSelected(true);
```

Кроме того, в начале исходного кода из данного примера нужно ввести следующий оператор `import` для поддержки интерфейса `ChangeListener`:

```
import javafx.beans.value.*;
```

Эта версия JavaFX-приложения действует таким же образом, как и предыдущая. Всякий раз, когда выбирается кнопка-переключатель, обновляется метка `response`. Но в данном случае в группу кнопок-переключателей требуется ввести лишь один обработчик событий вместо трех для каждой кнопки-переключателя в отдельности. А теперь рассмотрим подробнее приведенный выше фрагмент кода.

Сначала в этом фрагменте кода регистрируется приемник событий изменения в группе кнопок-переключателей. Чтобы принимать события изменения, следует реализовать интерфейс `ChangeListener`. С этой целью вызывается метод `addListener()` для объекта, возвращаемого методом `selectedToggleProperty()`. В интерфейсе `ChangeListener` определяется единственный метод `changed()`. Ниже приведена его общая форма.

```
void changed(ObservableValue<? extends T> изменение,
             T старое_значение, T новое_значение)
```

В данном случае параметр *изменение* обозначает экземпляр интерфейса `ObservableValue<T>`, инкапсулирующего объект, в котором наблюдаются изме-

нения, а параметры *старое_значение* и *новое_значение* — предыдущее и следующее значения соответственно. Таким образом, параметр *новое_значение* содержит ссылку на только что выбранную кнопку-переключатель.

В данном примере для установки первоначально выбираемой кнопки-переключателя вызывается метод `setSelected()`, а не `fire()`. А поскольку такая установка вызывает изменение в группе кнопок-переключателей, то когда приложение из данного примера начинает выполняться, генерируется событие изменения. Для установки первоначально выбираемой кнопки-переключателя можно было бы вызвать и метод `fire()`, но вместо него был выбран метод `setSelected()`, чтобы продемонстрировать то обстоятельство, что при любом изменении в группе кнопок-переключателей генерируется событие изменения.

Другой способ управления кнопками-переключателями

Несмотря на всю пользу от обработки событий, генерируемых кнопками-переключателями, иногда удобнее просто проигнорировать события от них и получить выбранную в настоящий момент кнопку-переключатель, когда такая информация потребуется. Именно такой способ демонстрируется в приведенном ниже примере. Это переделанный вариант предыдущего примера, где дополнительно введена кнопка **Confirm Transport Selection** (Подтвердить выбор транспортного средства). После щелчка на этой кнопке сначала получается выбранная в настоящий момент кнопка-переключатель, а затем на месте метки отображается подтвержденный результат выбора транспортного средства. Опробуя JavaFX-приложение из данного примера, обратите внимание на то, что изменение в выбираемой кнопке-переключателе не влечет за собой изменение подтвержденного результата выбора транспортного средства до тех пор, пока не будет нажата кнопка **Confirm Transport Selection**.

```
// В этом примере применения кнопок-переключателей
// демонстрируется получение кнопки-переключателя,
// выбранной в текущий момент из группы, под управлением
// программы, когда в этом возникает потребность, вместо
// реагирования на события действия или изменения.
//
// В данном примере события, связанные с кнопками-переключателями,
// не обрабатываются. Вместо этого просто получается выбранная в
// данный момент кнопка-переключатель, когда нажимается экранная
// кнопка Confirm Transport Selection
```

```
import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;
```

```
public class RadioButtonDemo2 extends Application {

    Label response;
    ToggleGroup tg;
```

```
public static void main(String[] args) {  
    // запустить JavaFX-приложение, вызвав метод launch()  
    launch(args);  
}  
  
// переопределить метод start()  
public void start(Stage myStage) {  
    // присвоить заголовок подмосткам  
    myStage.setTitle("Demonstrate Radio Buttons");  
    // Продемонстрировать кнопки-переключатели  
  
    // Использовать панель поточной компоновки FlowPane  
    // в качестве корневого узла. В данном случае с  
    // промежутками 10 по вертикали и по горизонтали  
    FlowPane rootNode = new FlowPane(10, 10);  
  
    // выровнять элементы управления по центру сцены  
    rootNode.setAlignment(Pos.CENTER);  
  
    // создать сцену  
    Scene myScene = new Scene(rootNode, 200, 140);  
  
    // установить сцену на подмостках  
    myStage.setScene(myScene);  
  
    // создать две метки  
    Label choose = new Label("    Select a Transport Type    ");  
    // Выбор транспортного средства  
    response = new Label("No transport confirmed");  
    // Выбор транспортного средства не подтвержден  
  
    // создать экранную кнопку для подтверждения выбора  
    // транспортного средства  
    Button btnConfirm = new Button("Confirm Transport Selection");  
    // Подтвердить выбор транспортного средства  
  
    // создать кнопки-переключатели  
    RadioButton rbTrain = new RadioButton("Train"); // Поезд  
    RadioButton rbCar = new RadioButton("Car"); // Автомобиль  
    RadioButton rbPlane = new RadioButton("Airplane"); // Самолет  
  
    // создать группу кнопок-переключателей  
    tg = new ToggleGroup();  
  
    // ввести каждую кнопку-переключатель в группу  
    rbTrain.setToggleGroup(tg);  
    rbCar.setToggleGroup(tg);  
    rbPlane.setToggleGroup(tg);  
  
    // первоначально выбрать одну из кнопок-переключателей  
    rbTrain.setSelected(true);  
  
    // обработать события действия от кнопки подтверждения  
    // выбора транспортного средства  
    btnConfirm.setOnAction(new EventHandler<ActionEvent>() {  
        public void handle(ActionEvent ae) {  
            // получить выбранную в настоящий момент кнопку-переключатель  
            RadioButton rb = (RadioButton) tg.getSelectedToggle();  
  
            // отобразить результат выбора транспортного средства  
            response.setText(rb.getText() + " is confirmed.");  
        }  
    });  
}
```



```

        // Подтверждено указанное транспортное средство
    }
});

// использовать разделитель, чтобы улучшить порядок
// расположения элементов управления
Separator separator = new Separator();
separator.setPrefWidth(180);

// ввести метку и все виды кнопок в граф сцены
rootNode.getChildren().addAll(choose, rbTrain, rbCar, rbPlane,
                                separator, btnConfirm, response);

// показать подмости и сцену на них
myStage.show();
}
}

```



Рис. 35.8. Результат подтверждения выбора одной из кнопок-переключателей в окне приложения `RadioButtonDemo2`

На рис. 35.8 показан результат подтверждения выбора одной из кнопок-переключателей в окне JavaFX-приложения из данного примера.

Большую часть кода из данного примера трудно понять, но особый интерес в нем вызывают два обстоятельства. Во-первых, обратите внимание на следующую строку кода из обработчика событий действия от кнопки `btnConfirm`, где получается выбранная кнопка-переключатель:

```

RadioButton rb = (RadioButton)
tg.getSelectedToggle();

```

где метод `getSelectedToggle()`, определенный в классе `ToggleGroup`, получает результат текущего выбора в группе переключателей (в данном случае кнопок-переключателей). Ниже приведена общая форма этого метода.

```
final Toggle getSelectedToggle()
```

Этот метод возвращает ссылку на выбранный переключатель типа `Toggle`. В данном случае значение, возвращаемое методом `getSelectedToggle()`, приводится к типу `RadioButton`, поскольку группа состоит из кнопок-переключателей.

И во-вторых, обратите внимание на применение визуального разделителя, создаваемого в следующем фрагменте кода:

```

Separator separator = new Separator();
separator.setPrefWidth(180);

```

Сначала в данном фрагменте кода средствами класса `Separator` создается разделяющая линия, которая может быть вертикальной или горизонтальной. По умолчанию создается горизонтальная линия. А второй конструктор этого класса позволяет выбрать вертикальную разделяющую линию. Такая линия улучшает внешний вид компоновки элементов управления ГПИ. Класс `Separator` входит в состав пакета `javafx.scene.control`. Затем в данном фрагменте кода вызывается метод `setPrefWidth()`, чтобы задать ширину разделяющей линии.

Класс CheckBox

Класс `CheckBox` инкапсулирует функциональные возможности флажка. Его непосредственным суперклассом служит класс `ButtonBase`. Флажки вам, без сомнения, хорошо известны, поскольку они широко применяются в ГПИ, но в JavaFX флажок является несколько более сложным элементом управления, чем может показаться на первый взгляд. Дело в том, что в классе `CheckBox` поддерживаются три состояния флажка. Двумя первыми являются установленное и сброшенное состояния, как и следовало ожидать, поскольку это стандартное поведение флажка. А третьим является так называемое *неопределенное* состояние флажка. Оно, как правило, обозначает незаданное или не соответствующее конкретной ситуации состояние флажка. Если требуется неопределенное состояние флажка, его придется разрешить явным образом.

В классе `CheckBox` определяются два конструктора. Первым из них является конструктор по умолчанию. А второй конструктор позволяет указать символьную строку, обозначающую флажок. Ниже приведена форма этого конструктора.

`CheckBox(String строка)`

Он создает флажок с текстом метки, обозначаемым параметром *строка*. Как и остальные разновидности кнопок, флажок типа `CheckBox` генерирует событие действия, когда он устанавливается.

В приведенном ниже примере демонстрируется применение флажков. В окне JavaFX-приложения из этого примера отображаются флажки, дающие пользователю возможность выбирать различные варианты развертывания приложения: **Web** (Веб), **Desktop** (Настольная система) и **Mobile** (Мобильное устройство). Всякий раз, когда флажок изменяет свое состояние, генерируется событие, обработка которого заключается в отображении нового (установленного или сброшенного) состояния флажка, а также списка всех установленных флажков.

// Продемонстрировать применение флажков

```
import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class CheckboxDemo extends Application {

    CheckBox cbWeb;
    CheckBox cbDesktop;
    CheckBox cbMobile;

    Label response;
    Label allTargets;

    String targets = "";

    public static void main(String[] args) {

        // запустить JavaFX-приложение, вызвав метод launch()
```

```

    launch(args);
}

// переопределить метод start()
public void start(Stage myStage) {

    // присвоить заголовок подмосткам
    myStage.setTitle("Demonstrate Checkboxes");
    // Прдемонстрировать флажки

    // Использовать панель поточной компоновки FlowPane
    // в качестве корневого узла. В данном случае с
    // промежутками 10 по вертикали и по горизонтали
    FlowPane rootNode = new FlowPane(10, 10);

    // выровнять элементы управления по центру сцены
    rootNode.setAlignment(Pos.CENTER);

    // создать сцену
    Scene myScene = new Scene(rootNode, 230, 140);

    // установить сцену на подмостках
    myStage.setScene(myScene);

    Label heading = new Label("Select Deployment Options");
    // Выбрать вариант развертывания приложения

    // создать метку, извещающую о состоянии установленного флажка
    response = new Label("No Deployment Selected");
    // Ни один из вариантов развертывания не выбран

    // создать метку, извещающую обо всех установленных флажках
    allTargets = new Label("Target List: <none>");
    // Список целевых флажков

    // создать флажки
    cbWeb = new CheckBox("Web"); // Веб
    cbDesktop = new CheckBox("Desktop"); // Настольная система
    cbMobile = new CheckBox("Mobile"); // Мобильное устройство

    // обработать события действия от флажков
    cbWeb.setOnAction(new EventHandler<ActionEvent>() {
        public void handle(ActionEvent ae) {
            if(cbWeb.isSelected())
                response.setText("Web deployment selected.");
            // Выбрано развертывание приложения в веб
            else
                response.setText("Web deployment cleared.");
            // Развертывание приложения в веб отменено

            showAll();
        }
    });

    cbDesktop.setOnAction(new EventHandler<ActionEvent>() {
        public void handle(ActionEvent ae) {
            if(cbDesktop.isSelected())
                response.setText("Desktop deployment selected.");
            // Выбрано развертывание приложения в настольной системе
            else
                response.setText("Desktop deployment cleared.");
            // Развертывание приложения в настольной системе отменено

            showAll();
        }
    });
}

```

```

    }
    });

    cbMobile.setOnAction(new EventHandler<ActionEvent>() {
        public void handle(ActionEvent ae) {
            if(cbMobile.isSelected())
                response.setText("Mobile deployment selected.");
            // Выбрано развертывание приложения на мобильном устройстве
            else
                response.setText("Mobile deployment cleared.");
            // Развертывание приложения на мобильном устройстве отменено

            showAll();
        }
    });

    // использовать разделитель, чтобы улучшить порядок
    // расположения элементов управления
    Separator separator = new Separator();
    separator.setPrefWidth(200);

    // ввести элементы управления в граф сцены
    rootNode.getChildren().addAll(heading, separator, cbWeb,
                                    cbDesktop, cbMobile, response, allTargets);

    // показать подмостки и сцену на них
    myStage.show();
}

// обновить и показать список целевых флажков
void showAll() {
    targets = "";
    if(cbWeb.isSelected()) targets = "Web ";
    if(cbDesktop.isSelected()) targets += "Desktop ";
    if(cbMobile.isSelected()) targets += "Mobile";

    if(targets.equals("")) targets = "<none>";

    allTargets.setText("Target List: " + targets);
}
}

```

На рис. 35.9 показан результат установки флажков в окне JavaFX-приложения из данного примера.

Принцип действия JavaFX-приложения из данного примера довольно прост. Всякий раз, когда состояние флажка изменяется, формируется команда действия. С целью определить, был ли флажок установлен, вызывается метод `isSelected()`.

Как упоминалось ранее, по умолчанию средствами класса `CheckBox` реализуется флажок с двумя состояниями: установленным и сброшенным. Если же требуется ввести третье, неопределенное состояние флажка, оно должно быть разрешено явным образом. С этой целью вызывается метод `setAllowIndeterminate()`. Ниже приведена его общая форма.

```
final void setAllowIndeterminate(boolean разрешение)
```

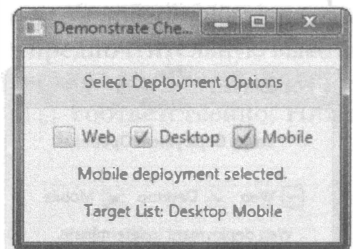


Рис. 35.9. Результат установки флажков в окне приложения `CheckboxDemo`

Если параметр *разрешение* принимает логическое значение `true`, то неопределенное состояние флажка разрешается, а иначе оно запрещается. Когда неопределенное состояние флажка разрешено, пользователь может выбирать между установленным, сброшенным и неопределенным состоянием флажка.

Вызвав метод `isIndeterminate()`, можно определить, находится ли флажок в неопределенном состоянии. Ниже приведена общая форма этого метода. Он возвращает логическое значение `true`, если флажок находится в неопределенном состоянии, а иначе — логическое значение `false`.

```
final boolean isIndeterminate()
```

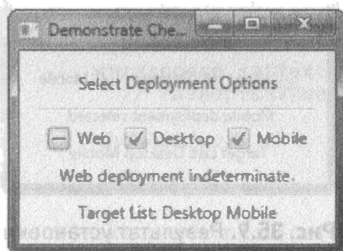
Чтобы опробовать флажок с тремя состояниями на практике, можете немного видоизменить исходный код из предыдущего примера. С этой целью вызовите сначала метод `setAllowIndeterminate()` для каждого флажка, чтобы разрешить его неопределенное состояние, как показано ниже.

```
cbWeb.setAllowIndeterminate(true);
cbDesktop.setAllowIndeterminate(true);
cbMobile.setAllowIndeterminate(true);
```

Затем организуйте обработку событий перехода флажков в неопределенное состояние в соответствующих обработчиках событий действия. В качестве примера ниже приведен видоизмененный обработчик событий от флажка `cbWeb`.

```
cbWeb.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        if(cbWeb.isIndeterminate())
            response.setText("Web deployment indeterminate.");
            // Развертывание приложения в веб не определено
        else if(cbWeb.isSelected())
            response.setText("Web deployment selected.");
            // Выбрано развертывание приложения в веб
        else
            response.setText("Web deployment cleared.");
            // Развертывание приложения в веб отменено

        showAll();
    }
});
```



Теперь проверяются все три состояния флажка. Внесите аналогичные изменения в обработчики событий от двух других флажков. После внесения этих изменений неопределенное состояние флажков может быть выбрано. Как показано на рис. 35.10, флажок **Web** находится в неопределенном состоянии.

Рис. 35.10. Вид окна приложения `CheckboxDemo` с флажком **Web** в неопределенном состоянии

Класс `ListView`

Еще одним элементом управления, часто применяемым при построении ГПИ, является представление списка, которое в JavaFX инкапсулировано в классе `ListView`. Представления списков — это элементы управления, отображающие

списки, из которых можно выбрать один или несколько элементов. Благодаря тому что представления списков эффективно используют полезную площадь экрана, они служат широко распространенной альтернативой другим типам элементов управления выбором.

Класс `ListView` является обобщенным и объявляется следующим образом:

```
class ListView<T>
```

где параметр *T* обозначает тип элементов, хранимых в представлении списка. Зачастую это элементы типа `String`, хотя допускаются элементы других типов.

В классе `ListView` определяются два конструктора. Первым из них является конструктор по умолчанию, создающий пустой объект типа `ListView`. А второй конструктор позволяет указать конкретный список элементов, как показано ниже.

```
ListView(ObservableList<T> список)
```

Здесь параметр *список* обозначает отображаемый список элементов. Он принимает объект типа `ObservableList`, определяющий список наблюдаемых объектов. Класс `ObservableList` наследует от класса `java.util.List`. Следовательно, он поддерживает стандартные методы обработки коллекций. Класс `ObservableList` входит в состав пакета `javafx.collections`.

Едва ли не самый простой способ создать список типа `ObservableList` для применения в представлении списка типа `ListView` — воспользоваться фабричным методом `observableArrayList()`, который определен как статический метод в классе `FXCollections`, также входящем в пакет `javafx.collections`. Ниже приведена применяемая здесь и далее общая форма этого метода, где параметр *E* обозначает тип элементов, передаваемых в качестве параметра *элементы*.

```
static <E> ObservableList<E> observableArrayList(E ... элементы)
```

По умолчанию в элементе управления типа `ListView` допускается одновременный выбор из списка только одного элемента. Но, изменив режим выбора, можно разрешить выбор нескольких элементов из списка. Ниже будет использована стандартная модель выбора одного элемента из списка.

Несмотря на то что в элементе управления типа `ListView` предоставляется размер списка по умолчанию, иногда требуется установить предпочтительную высоту и/или ширину в соответствии с конкретными потребностями. С этой целью можно вызвать методы `setPrefHeight()` и `setPrefWidth()` соответственно. Ниже приведены их общие формы.

```
final void setPrefHeight(double высота)  
final void setPrefWidth(double ширина)
```

С другой стороны, оба предпочтительных размера можно установить сразу, вызвав метод `setPrefSize()`. Его общая форма выглядит следующим образом:

```
void setPrefSize(double ширина, double высота)
```

Элементом управления типа `ListView` можно воспользоваться двумя основными способами. Во-первых, проигнорировать события, генерируемые списком, и просто получить результат выбора из списка, когда это потребуется в прикладной программе. И во-вторых, отслеживать изменения в списке, зарегистрировав

приемник событий изменения. Последний способ дает возможность оперативно реагировать на изменения, вносимые пользователем в выбор элементов из списка. Именно этот способ и применяется далее.

Для приема событий изменения нужно сначала получить модель выбора, применяемую в элементе управления типа `ListView`. С этой целью для списка вызывается метод `getSelectionModel()`, общая форма которого приведена ниже.

```
final MultipleSelectionModel<T> getSelectionModel()
```

Этот метод возвращает ссылку на модель. В классе `MultipleSelectionModel` определяется модель, применяемая для одновременного выбора нескольких элементов из списка. Этот класс наследует от класса `SelectionModel`. Но одновременно выбирать несколько элементов из списка, представленного объектом типа `ListView`, можно только в том случае, если активизирована модель, поддерживающая такой режим выбора.

Используя модель, возвращаемую методом `getSelectionModel()`, можно получить ссылку на свойство элемента, выбранного из списка. Это свойство определяет событие, происходящее при выборе элемента из списка. С этой целью вызывается метод `selectedItemProperty()`, общая форма которого приведена ниже. К полученному в итоге свойству присоединяется приемник событий изменения.

```
final ReadOnlyObjectProperty<T> selectedItemProperty()
```

Все сказанное выше о представлении списка демонстрируется в приведенном ниже примере. В частности, в этом примере создается представление списка, отображающее различные виды транспортных средств, которые пользователь может выбрать из списка. Результат выбора отображается на месте метки.

```
// Продемонстрировать применение представления списка

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.geometry.*;
import javafx.beans.value.*;
import javafx.collections.*;

public class ListViewDemo extends Application {

    Label response;

    public static void main(String[] args) {

        // запустить JavaFX-приложение, вызвав метод launch()
        launch(args);
    }

    // переопределить метод start()
    public void start(Stage myStage) {

        // присвоить заголовок подмосткам
        myStage.setTitle("ListView Demo");
        // Демонстрация представления списка

        // Использовать панель поточной компоновки FlowPane
```

```
// в качестве корневого узла. В данном случае с
// промежутками 10 по вертикали и по горизонтали
FlowPane rootNode = new FlowPane(10, 10);

// выровнять элементы управления по центру сцены
rootNode.setAlignment(Pos.CENTER);

// создать сцену
Scene myScene = new Scene(rootNode, 200, 120);

// установить сцену на подмостках
myStage.setScene(myScene);

// создать метку
response = new Label("Select Transport Type");
// Выбрать вид транспортного средства

// создать список типа ObservableList из элементов
// для представления списка
ObservableList<String> transportTypes =
    FXCollections.observableArrayList("Train", "Car", "Airplane");

// создать представление списка
ListView<String> lvTransport =
    new ListView<String>(transportTypes);

// задать предпочтительную высоту и ширину представления списка
lvTransport.setPrefSize(80, 80);

// получить модель выбора для представления списка
MultipleSelectionModel<String> lvSelModel =
    lvTransport.getSelectionModel();

// ввести приемник событий изменения, чтобы реагировать на
// выбор элементов в представлении списка
lvSelModel.selectedItemProperty().addListener(
    new ChangeListener<String>() {
        public void changed(ObservableValue<? extends String> changed,
            String oldVal, String newVal) {

            // отобразить результат выбора
            response.setText("Transport selected is " + newVal);
            // Выбрано указанное транспортное средство
        }
    });

// ввести метку и представление списка в граф сцены
rootNode.getChildren().addAll(lvTransport, response);

// показать подмостки и сцену на них
myStage.show();
}
```

На рис. 35.11 показан результат выбора транспортного средства из списка в окне JavaFX-приложения из данного примера.

Обратите в данном примере особое внимание на порядок создания представления списка типа `ListView`. Сначала создается список типа `ObservableList`, как показано ниже.

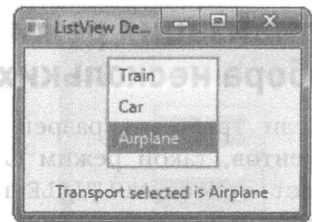


Рис. 35.11. Результат выбора транспортного средства из списка в окне приложения `ListViewDemo`


```
ObservableList<String> transportTypes =
    FXCollections.observableArrayList("Train", "Car", "Airplane");
```

Для составления списка из символьных строк в данном фрагменте кода вызывается метод `observableArrayList()`. Затем составленный список типа `ObservableList` используется для инициализации представления списка типа `ListView` приведенным ниже образом. А после этого задаются предпочтительные размеры данного элемента управления.

```
ListView<String> lvTransport = new ListView<String>(transportTypes);
```

В следующей строке кода показано, каким образом модель выбора получается для представления списка `lvTransport`:

```
MultipleSelectionModel<String> lvSelModel = lvTransport.getSelectionModel();
```

Как пояснялось выше, в элементе управления типа `ListView` применяется модель типа `MultipleSelectionModel`, хотя и разрешается одновременный выбор из списка только одного элемента. Для полученной в итоге модели выбора далее вызывается метод `selectedItemProperty()`, а для возвращаемого свойства регистрируется приемник событий изменения.

Представление списка с полосами прокрутки



Рис. 35.12. Список транспортных средств с полосой прокрутки в окне приложения `ListViewDemo`

К числу очень полезных особенностей класса `ListView` относится возможность автоматически снабжать список полосами прокрутки, когда количество его элементов превышает отображаемое в пределах заданных размеров. Например, объявление списка `transportTypes` можно изменить таким образом, чтобы он включал элементы "Bicycle" (Велосипед) и "Walking" (Пеший ход), как показано ниже.

И тогда список `transportTypes` будет автоматически снабжен полосами прокрутки, как показано на рис. 35.12.

```
ObservableList<String> transportTypes =
    FXCollections.observableArrayList("Train", "Car", "Airplane",
                                      "Bicycle", "Walking");
```

Активизация режима одновременного выбора нескольких элементов из списка

Если требуется разрешить одновременный выбор из списка нескольких элементов, такой режим следует запросить явным образом, указав константу `SelectionMode.MULTIPLE` при вызове метода `setSelectionMode()` для модели выбора в элементе управления типа `ListView`. Ниже приведена общая форма данного метода, где параметр *режим* должен принимать значение константы `SelectionMode.MULTIPLE` или `SelectionMode.SINGLE`.

```
final void setSelectionMode(SelectionMode режим)
```

Если режим одновременного выбора нескольких элементов из списка разрешен, то список выбранных элементов можно получить в двух формах: в виде самих выбранных элементов или их индексов. Здесь и далее будет использоваться список выбранных элементов, но та же самая процедура распространяется и на список индексов выбранных элементов. Следует, однако, иметь в виду, что индексация элементов в представлении списка типа `ListView` начинается с нуля.

Чтобы получить список выбранных элементов, следует вызвать метод `getSelectedItems()` для модели выбора. Ниже приведена общая форма данного метода.

```
ObservableList<T> getSelectedItems()
```

Этот метод возвращает список выбранных элементов типа `ObservableList`. А поскольку класс `ObservableList` расширяет класс `java.util.List`, то доступ к элементам списка можно получить таким же образом, как и к элементам любой другой коллекции типа `List`.

В качестве эксперимента с одновременным выбором нескольких элементов из списка попробуйте внести следующие изменения в исходный код из предыдущего примера. Сначала объявите список `lvTransport` как `final`, чтобы сделать его доступным в пределах обработчика событий изменения. Затем введите следующую строку кода:

```
lvTransport.getSelectionModel().setSelectionMode(SelectionMode.MULTIPLE);
```

В этой строке кода активизируется режим одновременного выбора нескольких элементов из списка типа `lvTransport`. И наконец, замените исходный код обработчика событий изменения на следующий:

```
lvSelModel.selectedItemProperty().addListener(
    new ChangeListener<String>() {
        public void changed(ObservableValue<? extends String> changed,
                           String oldVal, String newVal) {

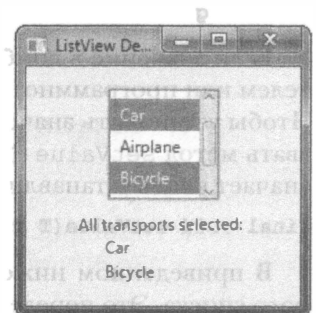
            String selItems = "";
            ObservableList<String> selected =
                lvTransport.getSelectionModel().getSelectedItems();

            // отобразить результаты выбора
            for(int i=0; i < selected.size(); i++)
                selItems += "\n" + selected.get(i);

            response.setText("All transports selected: " + selItems);
            // Все выбранные транспортные средства
        }
    });
```

После внесения упомянутых выше изменений в исходный код приложения `ListViewDemo` все выбранные виды транспортных средств будут отображаться ниже списка, как показано на рис. 35.13.

Рис. 35.13. Перечисление всех выбранных из списка транспортных средств в окне приложения `ListViewDemo`



Класс ComboBox

С представлением списка связан также элемент управления, называемый *комбинированным списком* и реализуемый в JavaFX классом `ComboBox`. В комбинированном списке отображается лишь один выбираемый элемент, а остальные элементы можно выбрать из дополнительно раскрываемого списка. Кроме того, пользователю можно предоставить возможность редактировать выбираемый элемент. Класс `ComboBox` наследует от класса `ComboBoxBase`, предоставляющего большую часть функциональных возможностей комбинированного списка. В отличие от представления списка типа `ListView`, допускающего одновременный выбор нескольких элементов из списка, комбинированный список типа `ComboBox` предназначен для одновременного выбора единственного элемента.

Класс `ComboBox` является обобщенным и объявляется следующим образом:

```
class ComboBox<T>
```

где параметр `T` обозначает тип элементов. Зачастую это элементы типа `String`, хотя допускаются и другие их типы.

В классе `ComboBox` определяются два конструктора. Первым из них является конструктор по умолчанию, создающий пустой комбинированный список типа `ComboBox`. А второй конструктор позволяет указать список элементов, как показано ниже.

```
ComboBox(ObservableList<T> список)
```

В данном случае параметр *список* обозначает отображаемый список элементов. Это объект типа `ObservableList`, определяющий список наблюдаемых объектов. Как пояснялось ранее, класс `ObservableList` наследует от класса `java.util.List`, а для создания списка типа `ObservableList` проще всего вызвать фабричный метод `observableArrayList()`, который определяется как статический метод в классе `FXCollections`.

Комбинированный список типа `ComboBox` генерирует событие действия, когда в нем изменяется выбираемый элемент. Он будет также генерировать событие изменения. С другой стороны, события, наступающие в комбинированном списке, можно проигнорировать и просто получить выбранный в настоящий момент элемент по мере надобности.

Вызвав метод `getValue()`, можно получить элемент, выбранный в настоящий момент из комбинированного списка. Ниже приведена общая форма данного метода.

```
final T getValue()
```

Если значение в комбинированном списке не установлено (вручную пользователем или программно), то метод `getValue()` возвращает пустое значение `null`. Чтобы установить значение в комбинированном списке программно, следует вызвать метод `setValue()`, как показано ниже, где параметр *новое_значение* обозначает вновь устанавливаемое значение.

```
final void setValue(T новое_значение)
```

В приведенном ниже примере демонстрируется применение комбинированного списка. Это переделанный вариант предыдущего примера, в котором демон-

стрировалось применение представления списка. В данном примере обрабатывается событие действия, генерируемое комбинированным списком.

// Продемонстрировать применение комбинированного списка

```
import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.geometry.*;
import javafx.collections.*;
import javafx.event.*;

public class ComboBoxDemo extends Application {

    ComboBox<String> cbTransport;
    Label response;

    public static void main(String[] args) {

        // запустить JavaFX-приложение, вызвав метод launch()
        launch(args);
    }

    // переопределить метод start()
    public void start(Stage myStage) {

        // присвоить заголовок подмосткам
        myStage.setTitle("ComboBox Demo");
        // Демонстрация комбинированного списка

        // Использовать панель поточной компоновки FlowPane
        // в качестве корневого узла. В данном случае с
        // промежутками 10 по вертикали и по горизонтали
        FlowPane rootNode = new FlowPane(10, 10);

        // выполнить выравнивание по центру
        rootNode.setAlignment(Pos.CENTER);

        // создать сцену
        Scene myScene = new Scene(rootNode, 280, 120);

        // установить сцену на подмостках
        myStage.setScene(myScene);

        // создать метку
        response = new Label();

        // создать список типа ObservableList из элементов,
        // предназначенных для комбинированного списка
        ObservableList<String> transportTypes =
            FXCollections.observableArrayList( "Train", "Car", "Airplane" );

        // создать комбинированный список
        cbTransport = new ComboBox<String>(transportTypes);

        // установить значение по умолчанию
        cbTransport.setValue("Train"); // Поезд
```

```

// установить метку ответной реакции для отображения
// результата выбора по умолчанию
response.setText(
    "Selected Transport is " + cbTransport.getValue();
    // Выбрано указанное транспортное средство

// принимать события действия от комбинированного списка
cbTransport.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        response.setText(
            "Selected Transport is " + cbTransport.getValue();
            // Выбрано указанное транспортное средство
        }
    });

// ввести метку и комбинированный список в граф сцены
rootNode.getChildren().addAll(cbTransport, response);

// показать подмостки и сцену на них
myStage.show();
}
}

```

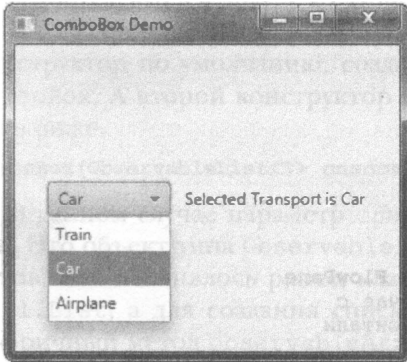


Рис. 35.14. Результат выбора транспортного средства из комбинированного списка в окне приложения ComboBoxDemo

На рис. 35.14 показан результат выбора транспортного средства из комбинированного списка в окне JavaFX-приложения из данного примера.

Как упоминалось выше, комбинированный список можно составить таким образом, чтобы пользователь мог редактировать выбираемый в нем элемент. Так, если комбинированный список составлен только из элементов типа String, то разрешить подобное редактирование совсем не трудно. Для этого достаточно вызвать метод `setEditable()`, общая форма которого приведена ниже.

```
final void setEditable(boolean разрешение)
```

Если параметр *разрешение* принимает логическое значение `true`, то редактирование разрешается, а иначе — запрещается. Чтобы посмотреть результаты редактирования элементов комбинированного списка, введите приведенную ниже строку в исходный код из предыдущего примера. После внесения этого изменения вы сможете отредактировать выбранный элемент комбинированного списка.

```
cbTransport.setEditable(true);
```

Помимо упомянутых выше, в классе `ComboBox` поддерживается немало других средств и функциональных возможностей комбинированных списков. Вам, вероятно, будет интересно изучить их самостоятельно. Иногда в качестве альтернативы комбинированному списку можно выбрать элемент управления типа `ChoiceBox`. Им нетрудно манипулировать, поскольку у него немало общего с элементами управления `ListView` и `ComboBox`.

Класс `TextField`

Безусловно, рассмотренные выше элементы управления очень удобны и нередко присутствуют в ГПИ, тем не менее все они реализуют средства для выбора предопределенного варианта или действия. Но иногда пользователю требуется предоставить возможность ввести избранную им символьную строку. Для организации такого типа ввода пользовательских данных в JavaFX предусмотрено несколько текстовых элементов управления. К их числу относится рассматриваемый здесь элемент управления типа `TextField`. Такой элемент управления позволяет ввести одну текстовую строку, а следовательно, он удобен для ввода имен, идентификаторов, адресов и прочих аналогичных данных. Как и все остальные классы текстовых элементов управления, класс `TextField` наследует от класса `TextInputControl`, в котором определяется большая часть функциональных возможностей для ввода текста.

В классе `TextField` определяются два конструктора. Первым из них является конструктор по умолчанию, создающий пустое текстовое поле стандартных размеров. А второй конструктор позволяет указать исходное содержимое текстового поля. Здесь и далее будет использоваться конструктор по умолчанию.

Хотя стандартных размеров текстового поля иногда оказывается достаточно, зачастую его размеры требуется указывать явным образом. Для этого нужно вызвать метод `setPrefColumnCount()`, общая форма которого приведена ниже, где параметр *столбцы* служит для определения размеров текстового поля в элементе управления типа `TextField`.

```
final void setPrefColumnCount(int столбцы)
```

Вызвав метод `setText()`, можно задать текст, отображаемый в текстовом поле, а вызвав метод `getText()` — получить текст из текстового поля. Помимо этих основных операций с текстовым полем, в элементе управления типа `TextField` поддерживается ряд других функциональных возможностей, которые полезно изучить самостоятельно, в том числе операции вырезания, вставки, присоединения, а также выделения части текста под управлением прикладной программы.

Особенно полезной в элементе управления типа `TextField` является возможность задать наводящее сообщение (т.е. подсказку) в текстовом поле, чтобы пользователь не пытался воспользоваться пустым полем. Для этого достаточно вызвать метод `setPromptText()`, общая форма которого приведена ниже.

```
final void setPromptText(String строка)
```

В данном случае параметр *строка* обозначает символьную строку, отображаемую в текстовом поле, если в нем отсутствует введенный текст. Эта строка отображается светлым (например, светло-серым) цветом.

Когда пользователь нажимает клавишу <Enter> в текстовом поле элемента управления типа `TextField`, генерируется событие действия. Зачастую это событие обрабатывается, но иногда в прикладной программе проще получить введенный текст по мере надобности, чем обрабатывать события действия от текстовых полей. Оба способа демонстрируются в приведенном ниже примере, где создается текстовое поле, в котором вводится строка запроса на поиск информации.

Когда фокус ввода находится в текстовом поле и пользователь нажимает клавишу <Enter> или щелкает на кнопке **Get Search String** (Получить строку запроса на поиск), введенная символьная строка извлекается и отображается. Кроме того, текстовое поле снабжается наводящим сообщением.

```
// Продемонстрировать применение текстового поля

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class TextFieldDemo extends Application {

    TextField tf;
    Label response;

    public static void main(String[] args) {

        // запустить JavaFX-приложение, вызвав метод launch()
        launch(args);
    }

    // переопределить метод start()
    public void start(Stage myStage) {

        // присвоить заголовок подмосткам
        myStage.setTitle("Demonstrate a TextField");
        // Продемонстрировать элемент управления типа TextField

        // Использовать панель поточной компоновки FlowPane
        // в качестве корневого узла. В данном случае с
        // промежутками 10 по вертикали и по горизонтали
        FlowPane rootNode = new FlowPane(10, 10);

        // выровнять элементы управления по центру сцены
        rootNode.setAlignment(Pos.CENTER);

        // создать сцену
        Scene myScene = new Scene(rootNode, 230, 140);

        // установить сцену на подмостках
        myStage.setScene(myScene);

        // создать метку, извещающую о содержимом текстового поля
        response = new Label("Search String: ");
        // Строка запроса на поиск информации

        // создать кнопку для получения текста
        Button btnGetText = new Button("Get Search String");
        // Получить строку запроса на поиск информации

        // создать текстовое поле
        tf = new TextField();

        // задать подсказку
        tf.setPromptText("Enter Search String");
        // Ввести строку запроса на поиск информации
```

```

// задать предпочтительное количество столбцов
tf.setPrefColumnCount(15);

// Обработать события действия от текстового поля.
// События действия генерируются при нажатии клавиши
// <ENTER>, когда фокус ввода находится в текстовом поле.
// В таком случае получается и отображается текст, введенный
// в текстовом поле
tf.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        response.setText("Search String: " + tf.getText());
        // Строка запроса на поиск информации
    }
});

// получить текст из текстового поля, если нажата
// клавиша <ENTER>, а затем отобразить его
btnGetText.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        response.setText("Search String: " + tf.getText());
        // Строка запроса на поиск информации
    }
});

// использовать разделитель, чтобы улучшить порядок
// расположения элементов управления
Separator separator = new Separator();
separator.setPrefWidth(180);

// ввести все элементы управления в граф сцены
rootNode.getChildren().addAll(tf, btnGetText, separator, response);

// показать подмостки и сцену на них
myStage.show();
}
}

```

На рис. 35.15 показан результат ввода строки запроса на поиск информации в текстовом поле окна JavaFX-приложения из данного примера.

К числу других текстовых элементов управления, которые полезно изучить самостоятельно, относятся класс `TextArea` для ввода многострочного текста, а также класс `PasswordField` для ввода паролей. Полезным может также оказаться класс `HTMLEditor`, реализующий редактор HTML-разметки.

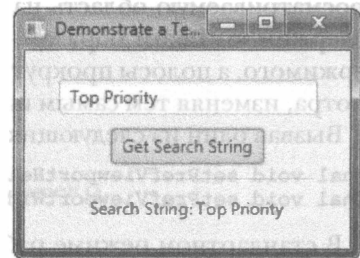


Рис. 35.15. Результат ввода строки запроса в текстовом поле окна приложения `TextFieldDemo`

Класс `ScrollPane`

Содержимое элемента управления порой не вмещается в тех пределах, которые отведены данному элементу управления на экране. Характерными тому примерами могут служить крупное изображение, не вписывающееся в разумные границы, или длинный текст, который требуется отобразить в небольшом окне. Для подобных случаев узлы графа сцены в JavaFX снабжаются полосами прокрутки. С этой

целью соответствующий узел заключается в оболочку класса `ScrollPane`, и тогда такой узел автоматически наделяется полосами для прокрутки его содержимого, не требуя больше ничего от программиста. Благодаря универсальности класса `ScrollPane` пользоваться отдельными элементами управления полосам прокрутки придется крайне редко.

В классе `ScrollPane` определяются два конструктора. Первым из них является конструктор по умолчанию, а второй конструктор позволяет указать узел, который требуется прокручивать. Этот конструктор объявляется следующим образом:

```
ScrollPane (Node содержимое)
```

В данном случае параметр *содержимое* обозначает прокручиваемые данные. Если же применяется конструктор по умолчанию, то для ввода прокручиваемого узла следует вызвать метод `setContent()`. Ниже приведена общая форма данного метода.

```
final void setContent (Node содержимое)
```

Как только будет задано прокручиваемое содержимое, следует ввести панель прокрутки в граф сцены. В итоге содержимое может быть прокручено, когда оно отображается.

На заметку! Чтобы изменить содержимое, прокручиваемое на панели прокрутки, можно также вызвать метод `setContent()`. В итоге прокручиваемое содержимое может быть изменено во время выполнения прикладной программы.

Как правило, размеры *окна просмотра* задаются явным образом, хотя по умолчанию выбираются стандартные размеры. Окно просмотра представляет собой просматриваемую область панели прокрутки, где отображается прокручиваемое содержимое. Таким образом, в окне просмотра отображается видимая часть содержимого, а полосы прокрутки позволяют прокручивать его в пределах окна просмотра, изменяя тем самым видимую часть содержимого.

Вызвав один из следующих методов, можно задать размеры окна просмотра:

```
final void setPrefViewportHeight(double высота)  
final void setPrefViewportWidth(double ширина)
```

В стандартном режиме работы элемент управления типа `ScrollPane` динамически вводит или удаляет полосы прокрутки по мере надобности. Так, если компонент оказывается выше, чем окно просмотра, в последнее автоматически вводится вертикальная полоса прокрутки. А если компонент полностью вписывается в окно просмотра, то полосы прокрутки удаляются из окна просмотра.

Замечательной особенностью элемента управления типа `ScrollPane` является его способность панорамировать содержимое перетаскиванием мыши. По умолчанию режим панорамирования отключен. Чтобы включить его, следует вызвать метод `setPannable()`, общая форма которого показана ниже. Если параметр *разрешение* принимает логическое значение `true`, то панорамирование разрешается, а иначе оно запрещается.

```
final void setPannable(boolean разрешение)
```

Вызвав методы `setHvalue()` и `setVvalue()`, можно установить расположение полос прокрутки под управлением прикладной программы. Ниже приведены общие формы этих методов.

```
final void setHvalue(double НППГ)
final void setVvalue(double НПВ)
```

Новая позиция полос прокрутки по горизонтали обозначается параметром *НППГ*, а по вертикали — параметром *НПВ*. По умолчанию позиции полос прокрутки по горизонтали и по вертикали начинаются с нуля.

В элементе управления типа `ScrollPane` поддерживаются и другие возможности. В частности, можно задать минимальные и максимальные позиции полос прокрутки, а также установить правила их появления в определенный момент. А вызвав методы `getHvalue()` и `getVvalue()`, можно получить текущие позиции полос прокрутки по горизонтали и по вертикали соответственно.

Применение элемента управления типа `ScrollPane` демонстрируется ниже на примере прокрутки многострочной метки. При этом разрешается также панорамирование прокручиваемого содержимого.

```
// Продемонстрировать применение панели прокрутки.
// В данном JavaFX-приложении прокручивается содержимое
// многострочной метки, хотя прокручиваться может любой
// узел графа сцены

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;

public class ScrollPaneDemo extends Application {

    ScrollPane scrlPane;

    public static void main(String[] args) {

        // запустить JavaFX-приложение, вызвав метод launch()
        launch(args);
    }

    // переопределить метод start()
    public void start(Stage myStage) {

        // присвоить заголовок подмосткам
        myStage.setTitle("Demonstrate a ScrollPane");
        // Продемонстрировать элемент управления ScrollPane

        // Использовать панель поточной компоновки FlowPane
        FlowPane rootNode = new FlowPane(10, 10);

        // выровнять элементы управления по центру сцены
        rootNode.setAlignment(Pos.CENTER);

        // создать сцену
        Scene myScene = new Scene(rootNode, 200, 200);
```

```

// установить сцену на подмостках
myStage.setScene(myScene);

// создать многострочную прокручиваемую метку, где
// отмечаются преимущества элемента управления ScrollPane
// над отдельными элементами управления полосами прокрутки
Label scrLLabel = new Label(
    "A ScrollPane streamlines the process of\n" +
    "adding scroll bars to a window whose\n" +
    "contents exceed the window's dimensions.\n" +
    "It also enables a control to fit in a\n" +
    "smaller space than it otherwise would.\n" +
    "As such, it often provides a superior\n" +
    "approach over using individual scroll bars.");

// создать панель прокрутки, установив в качестве
// содержимого метку scrLLabel
scrLPane = new ScrollPane(scrLLabel);

// задать ширину и высоту окна просмотра
scrLPane.setPrefViewportWidth(130);
scrLPane.setPrefViewportHeight(80);

// разрешить панорамирование прокручиваемого содержимого
scrLPane.setPannable(true);

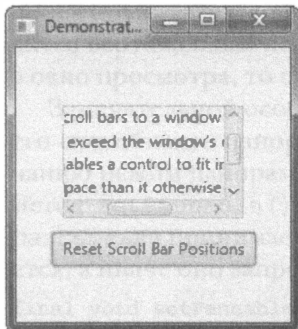
// создать кнопку сброса
Button btnReset = new Button("Reset Scroll Bar Positions");
// Установить полосы прокрутки в исходное положение

// обработать события действия от кнопки сброса
btnReset.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        // установить полосы прокрутки на нулевые позиции
        scrLPane.setVvalue(0);
        scrLPane.setHvalue(0);
    }
});

// ввести метку и кнопку сброса в граф сцены
rootNode.getChildren().addAll(scrLPane, btnReset);

// показать подмостки и сцену на них
myStage.show();
}

```



На рис. 35.16 показан вид многострочной метки с полосами прокрутки и кнопкой их установки в исходное положение в окне JavaFX-приложения из данного примера.

Рис. 35.16. Вид многострочной метки с полосами прокрутки в окне приложения ScrollPaneDemo

Класс `TreeView`

Одним из самых сложных в JavaFX считается элемент управления `TreeView`. Он реализует иерархическое представление данных в древовидной форме. В данном контексте термин *иерархический* означает, что одни элементы древовидной структуры подчиняются другим. Например, древовидная иерархия часто применяется для отображения содержимого файловой системы. В этом случае отдельные файлы подчиняются каталогу, который их содержит. В элементе управления типа `TreeView` отдельные ветви древовидной структуры могут разворачиваться или сворачиваться по требованию пользователя. Это дает возможность представить иерархические данные в компактной, но разворачиваемой по мере надобности форме. И хотя элемент управления типа `TreeView` поддерживает разнообразную специальную настройку, зачастую вполне достаточно и стандартного стиля и возможностей древовидного представления данных. Таким образом, деревья достаточно просты в применении, хотя и поддерживают довольно сложную иерархическую структуру.

В элементе управления типа `TreeView` реализуется принципиально простая древовидная структура, которая начинается с одного корневого узла, обозначающего начало дерева. Под корневым узлом располагается один или несколько *потомственных узлов*. Имеются два типа потомственных узлов: *концевые узлы*, называемые иначе *оконечными* и не имеющие потомственных узлов, и *узлы ветвления*, образующие корневые узлы *поддеревьев*. Поддерево представляет собой обычное дерево, являющееся частью более крупного дерева. Последовательность узлов, простирающаяся от корневого узла к указанному узлу, называется *путем*.

Одной из удобных особенностей элемента управления типа `TreeView` является автоматическое предоставление полос прокрутки, когда размеры дерева превышают размеры представления. Если полностью свернутое дерево может быть достаточно маленьким, то его развернутая форма — достаточно крупной. Автоматически снабжая дерево полосами прокрутки, элемент управления типа `TreeView` позволяет использовать меньше полезного пространства экрана, чем обычно требуется.

Класс `TreeView` является обобщенным и определяется следующим образом:

```
class TreeView<T>
```

где параметр `T` обозначает тип значения, находящегося в узле дерева. Зачастую это значения типа `String`. В классе `TreeView` определяются два конструктора. Ниже приведен применяемый здесь и далее конструктор.

```
TreeView(TreeItem<T> корневой_узел)
```

В данном случае параметр *корневой_узел* обозначает корень дерева. Это единственный параметр, который требуется передать конструктору класса типа `TreeView`, поскольку все узлы дерева происходят от корневого узла.

Узлы, образующие дерево, являются объектами типа `TreeItem`. Прежде всего следует заметить, что класс `TreeItem` не наследует от класса `Node`. Поэтому объекты типа `TreeItem` не являются объектами общего назначения. И хотя их можно использовать в элементе управления типа `TreeView`, они не являются автономными элементами управления. Класс `TreeItem` является обобщенным и объявляется

приведенным ниже образом, где параметр **T** обозначает тип значения, находящегося в объекте типа `TreeItem`.

```
class TreeItem<T>
```

Прежде чем воспользоваться элементом управления типа `TreeView`, следует построить дерево, которое он должен отображать. Для этого нужно сначала создать корневой узел дерева, а затем добавить к нему другие узлы. С этой целью вызывается метод `add()` или `addAll()` для списка, возвращаемого методом `getChildren()`. Добавляемые узлы могут быть концевыми узлами или поддеревьями. Как только дерево будет построено, можно создать объект класса `TreeView`, передав его конструктору корневой узел в качестве параметра.

События выбора, наступающие в элементе управления типа `TreeView`, могут быть обработаны таким же образом, как и в элементе управления типа `ListView`, с помощью приемника событий изменения. Для этого следует сначала получить модель выбора, вызвав метод `getSelectionModel()`, а затем свойство выбранного элемента, вызвав метод `selectedItemProperty()`, и наконец ввести приемник событий изменения, вызвав метод `addListener()`. Всякий раз, когда делается выбор, приемнику событий `changed()` в качестве нового значения передается ссылка на вновь выбранный элемент. (Более подробно этот процесс описан ранее при обсуждении обработки событий изменения в элементе управления типа `ListView`.)

Значение элемента типа `TreeItem` можно получить, вызвав метод `getValue()`. Кроме того, можно последовать по пути к элементу, перемещаясь по дереву в прямом или обратном направлении. В частности, вызвав метод `getParent()`, можно дойти до родительского узла, а вызвав метод `getChildren()` — до порожденных узлов.

В приведенном ниже примере демонстрируется построение и применение дерева типа `TreeView`, представляющего иерархию пищевых продуктов. В узлах этого дерева находятся элементы типа символьных строк. Корневой узел дерева обозначен меткой **Food** (Пища), а под ним располагаются три производных узла: **Fruit** (Фрукты), **Vegetables** (Овощи) и **Nuts** (Орехи). Ниже узла **Fruit** располагаются три порожденных узла: **Apples** (Яблоки), **Pears** (Груши) и **Oranges** (Апельсины), а ниже узла **Apples** — три концевых узла с названиями сортов яблок **Fuji** (Фудзи), **Winesap** (Уайнсэп) и **Jonathan** (Джонатан). Всякий раз, когда элемент выбирается из дерева, он отображается вместе с путем к нему от корневого узла дерева. С этой целью неоднократно вызывается метод `getParent()`.

```
// Продемонстрировать применение элемента управления TreeView
```

```
import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.beans.value.*;
import javafx.geometry.*;

public class TreeViewDemo extends Application {
```

```
Label response;

public static void main(String[] args) {

    // запустить JavaFX-приложение, вызвав метод launch()
    launch(args);
}

// переопределить метод start()
public void start(Stage myStage) {

    // присвоить заголовок подмосткам
    myStage.setTitle("Demonstrate a TreeView");
    // Протестируем элемент управления TreeView

    // Использовать панель поточной компоновки FlowPane
    // в качестве корневого узла. В данном случае с
    // промежутками 10 по вертикали и по горизонтали
    FlowPane rootNode = new FlowPane(10, 10);

    // выровнять элементы управления по центру сцены
    rootNode.setAlignment(Pos.CENTER);

    // создать сцену
    Scene myScene = new Scene(rootNode, 310, 460);

    // установить сцену на подмостках
    myStage.setScene(myScene);

    // создать метку, извещающую о состоянии элемента,
    // выбранного из дерева
    response = new Label("No Selection");
    // Ничего не выбрано

    // создать узлы дерева, начиная с корневого узла
    TreeItem<String> tiRoot = new TreeItem<String>("Food");

    // ввести поддерева, начиная с узла фруктов
    TreeItem<String> tiFruit = new TreeItem<String>("Fruit");

    // построить узел яблок
    TreeItem<String> tiApples = new TreeItem<String>("Apples");

    // ввести порожденные узлы сортов яблок в узел яблок
    tiApples.getChildren().add(new TreeItem<String>("Fuji"));
    tiApples.getChildren().add(new TreeItem<String>("Winesap"));
    tiApples.getChildren().add(new TreeItem<String>("Jonathan"));

    // ввести порожденные узлы видов фруктов в узел фруктов
    tiFruit.getChildren().add(tiApples);
    tiFruit.getChildren().add(new TreeItem<String>("Pears"));
    tiFruit.getChildren().add(new TreeItem<String>("Oranges"));

    // и наконец, ввести узел фруктов в корневой узел
    tiRoot.getChildren().add(tiFruit);

    // а теперь ввести аналогичным образом узел овощей
    TreeItem<String> tiVegetables = new TreeItem<String>("Vegetables");
    tiVegetables.getChildren().add(new TreeItem<String>("Corn"));
    tiVegetables.getChildren().add(new TreeItem<String>("Peas"));
    tiVegetables.getChildren().add(new TreeItem<String>("Broccoli"));
    tiVegetables.getChildren().add(new TreeItem<String>("Beans"));
    tiRoot.getChildren().add(tiVegetables);
}
```

```

// и наконец, ввести аналогичным образом узел орехов
TreeItem<String> tiNuts = new TreeItem<String>("Nuts");
tiNuts.getChildren().add(new TreeItem<String>("Walnuts"));
tiNuts.getChildren().add(new TreeItem<String>("Peanuts"));
tiNuts.getChildren().add(new TreeItem<String>("Pecans"));
tiRoot.getChildren().add(tiNuts);

// создать древовидное представление, используя только
// что построенное дерево
TreeView<String> tvFood = new TreeView<String>(tiRoot);

// получить модель выбора для древовидного представления
MultipleSelectionModel<TreeItem<String>> tvSelModel =
    tvFood.getSelectionModel();

// использовать приемник событий изменения, чтобы оперативно
// реагировать на выбор элементов в древовидном представлении
tvSelModel.selectedItemProperty().addListener(
    new ChangeListener<TreeItem<String>>() {
        public void changed(
            ObservableValue<? extends TreeItem<String>> changed,
            TreeItem<String> oldVal, TreeItem<String> newVal) {

            // отобразить выбранный элемент и полный путь от
            // него к корневому узлу
            if(newVal != null) {

                // построить весь путь к выбранному элементу
                String path = newVal.getValue();
                TreeItem<String> tmp = newVal.getParent();
                while(tmp != null) {
                    path = tmp.getValue() + " -> " + path;
                    tmp = tmp.getParent();
                }

                // отобразить выбранный элемент и полный путь к нему
                response.setText("Selection is " + newVal.getValue() +
                    "\nComplete path is " + path);

                // Выбран указанный элемент
                // Полный путь к нему
            }
        }
    });

// ввести элементы управления в граф сцены
rootNode.getChildren().addAll(tvFood, response);

// показать подмостки и сцену на них
myStage.show();
}
}

```

На рис. 35.17 показан вид древовидного представления пищевых продуктов в окне JavaFX-приложения из данного примера.

В данном примере особое внимание необходимо обратить на две особенности. Во-первых, это порядок построения дерева. Сначала создается корневой узел дерева, как показано ниже.

```
TreeItem<String> tiRoot = new TreeItem<String>("Food");
```

Затем строятся узлы, располагающиеся ниже корневого узла. Эти узлы образуют корни следующих поддеревьев: фруктов, овощей и орехов. Далее к этим под-

деревьям добавляются листья. Но одно из них (поддереву фруктов) содержит другое поддерево — сортов яблок. Суть здесь в том, что каждая ветвь дерева приводит к листу или корню поддерева. Как только все узлы дерева будут построены, корневые узлы каждого поддерева добавляются к корневому узлу дерева. В качестве примера ниже показано, каким образом поддерево орехов добавляется к корневому узлу `tiRoot`. Аналогичным образом любой порожденный узел добавляется к своему родительскому узлу.

```
tiRoot.getChildren().add(tiNuts);
```

И во-вторых, это порядок построения пути от корневого узла к выбранному узлу в обработке событий изменения. Ниже показано, как это делается.

```
String path = newVal.getValue();
TreeItem<String> tmp = newVal.getParent();
while(tmp != null) {
    path = tmp.getValue() + " -> " + path;
    tmp = tmp.getParent();
}
```

Этот фрагмент кода действует следующим образом. Сначала получается значение из вновь выбранного узла дерева. В данном примере это строковое значение с именем узла. Полученная в итоге символьная строка присваивается строковой переменной `path`. Затем создается временная переменная `tmp` типа `TreeItem<String>`, которая инициализируется ссылкой на родительский узел вновь выбранного узла дерева. Если у этого узла отсутствует родительский узел, то переменной `tmp` присваивается пустое значение `null`. В противном случае начинает выполняться цикл, в котором к содержимому переменной `path` добавляется значение из каждого родительского узла (в данном случае имя этого узла). Этот процесс продолжается до тех пор, пока не будет достигнут корневой узел дерева, у которого отсутствует родительский узел.

Выше был представлен лишь самый основной порядок обращения с элементом управления типа `TreeView`. Следует, однако, иметь в виду, что этот элемент управления поддерживает разнообразную специальную настройку. Поэтому все потенциальные возможности элемента управления типа `TreeView` вам придется изучить самостоятельно.

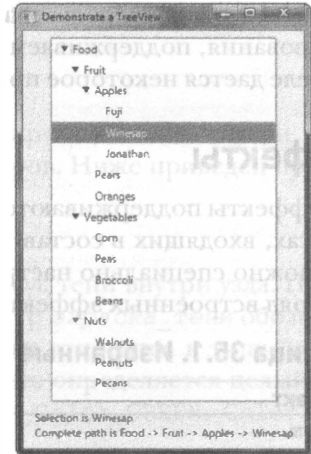


Рис. 35.17. Вид древовидного представления пищевых продуктов в окне приложения `TreeViewDemo`

Эффекты и преобразования

Главное преимущество каркаса JavaFX заключается в том, что он позволяет изменять внешний вид элементов управления (или узлов графа сцены) с помощью применяемых *эффектов* и/или выполняемых *преобразований*. Эффекты и преобразования придают ГПИ изощренный современный внешний вид, который уже

привыкли ожидать пользователи. И хотя рассмотрение каждого эффекта и преобразования, поддерживаемого в JavaFX, выходит за рамки данной книги, в этом разделе дается некоторое представление о тех преимуществах, которые они дают.

Эффекты

Эффекты поддерживаются в абстрактном классе `Effect` и его конкретных подклассах, входящих в состав пакета `javafx.scene.effect`. Применяя эти эффекты, можно специально настроить внешний вид узлов в графе сцены. Имеется целый ряд встроенных эффектов. В табл. 35.1 перечислены лишь некоторые из них.

Таблица 35.1. Избранные эффекты из класса `Effect`

Эффект	Описание
Bloom	Увеличивает яркость самых ярких частей узла
BoxBlur	Размывает узел, делая его нерезким
DropShadow	Отображает падающую тень позади узла
Glow	Воспроизводит эффект свечения
InnerShadow	Отображает тень внутри узла
Lighting	Создает теневые эффекты источника света
Reflection	Воспроизводит отражение

Эти и другие эффекты просты в применении и доступны для использования с любым объектом типа `Node`, включая и элементы управления. Разумеется, одни эффекты подходят отдельным элементам управления больше, чем другие.

Чтобы задать эффект для узла, следует вызвать метод `setEffect()`, определяемый в классе `Node`. Ниже приведена общая форма данного метода.

```
final void setEffect(Effect эффект)
```

Здесь параметр *эффект* обозначает применяемый эффект. Если же никакого эффекта применять не требуется, то в качестве этого параметра следует передать пустое значение `null`. Следовательно, чтобы применить эффект к узлу, сначала нужно создать экземпляр объекта типа `Effect`, а затем передать его методу `setEffect()` в качестве параметра. Как только это будет сделано, эффект станет применяться к узлу всякий раз, когда он воспроизводится, при условии, что он поддерживается исполняющей средой. Чтобы продемонстрировать потенциальные возможности эффектов, здесь и далее будут подробно рассмотрены два из них: свечение (класс `Glow`) и внутренняя тень (класс `InnerShadow`). Но процесс применения, по существу, одинаков для всех эффектов.

Класс `Glow` воспроизводит эффект, придающий узлу светящийся внешний вид. Степень свечения можно регулировать вручную. Чтобы применить эффект свечения, нужно создать сначала экземпляр класса `Glow`. Ниже приведен применяемый здесь и далее конструктор этого класса, где параметр *уровень_свечения* обозначает степень свечения в пределах от 0.0 до 1.0.

```
Glow(double уровень_свечения)
```

Как только будет создан экземпляр класса `Glow`, уровень свечения можно изменить, вызвав метод `setLevel()`. Ниже приведена общая форма данного мето-

да, где параметр *уровень_свечения*, как и прежде, обозначает степень свечения в пределах от 0.0 до 1.0.

```
final void setLevel(double уровень_свечения)
```

Класс `InnerShadow` воспроизводит эффект, имитирующий тень внутри узла. В этом классе предоставляется целый ряд конструкторов. Ниже приведен применяемый здесь и далее конструктор данного класса.

```
InnerShadow(double радиус, Color окраска_тени)
```

Здесь параметр *радиус* обозначает указанный радиус тени внутри узла. По существу, радиус тени определяет ее размеры. А параметр *окраска_тени* обозначает цвет, в который окрашивается тень. Этот параметр относится к типу `Color`, т.е. к классу `javafx.scene.paint.Color`. В этом классе определяется целый ряд констант, описывающих отдельные цвета, например `Color.GREEN`, `Color.RED` и `Color.BLUE`, что упрощает задание цвета.

Преобразования

Преобразования поддерживаются в абстрактном классе `Transform`, входящем в состав пакета `javafx.scene.transform`. У него имеются четыре конкретных подкласса, `Rotate`, `Scale`, `Shear` и `Translate`, для выполнения преобразований вращением, масштабированием, наклоном и перемещением соответственно. Имеется также подкласс `Affine`, но, как правило, применяются перечисленные выше подклассы преобразований в определенном сочетании. Это означает, что над узлом можно выполнить несколько преобразований, например, вращать или масштабировать его. Ниже подробно рассматриваются те преобразования, которые поддерживаются в классе `Node`.

Чтобы выполнить над узлом преобразование, его можно, например, ввести в список преобразований, поддерживаемых в данном узле. Для получения этого списка следует вызвать метод `getTransforms()`, определяемый в классе `Node`. Ниже приведена общая форма данного метода.

```
final ObservableList<Transform> getTransforms()
```

Этот метод возвращает ссылку на список преобразований. Чтобы ввести преобразование в этот список, достаточно вызвать метод `add()`, чтобы очистить список — метод `clear()`, а для того чтобы удалить конкретный элемент из списка — метод `remove()`.

Иногда преобразование можно указать непосредственно, установив свойства класса `Node`. Например, установить угол поворота узла с точкой вращения в его центре можно, вызвав метод `setRotate()` и передав ему требуемое значение угла; задать масштаб — вызвав методы `setScaleX()` и `setScaleY()`, а переместить узел — вызвав методы `setTranslateX()` и `setTranslateY()`. Но намного удобнее пользоваться списком преобразований, как будет продемонстрировано далее.

На заметку! Любые преобразования, указанные для узла непосредственно, будут выполняться только после всех преобразований из списка.

Для демонстрации преобразований здесь и далее будут использованы классы `Rotate` и `Scale`. Аналогичным образом выполняются и остальные преобразования. В классе `Rotate` поддерживается вращение узла вокруг указанной точки. В этом классе поддерживается несколько конструкторов. Ниже в качестве примера приведен один из них.

```
Rotate(double угол, double x, double y)
```

Здесь параметр *угол* обозначает количество градусов, на которое следует повернуть узел, а центр вращения, иначе называемый *точкой вращения*, определяется параметрами *x* и *y*. Все эти параметры вращения можно установить и после создания объекта типа `Rotate` с помощью конструктора по умолчанию, как демонстрируется в приведенном далее примере. Для этой цели достаточно вызвать методы `setAngle()`, `setPivotX()` и `setPivotY()`, общие формы которых приведены ниже. Как и прежде, параметр *угол* обозначает количество градусов, на которое следует повернуть узел, а центр вращения определяется параметрами *x* и *y*.

```
final void setAngle(double угол)  
final void setPivotX(double x)  
final void setPivotY(double y)
```

В классе `Scale` поддерживается масштабирование узла по указанному масштабу коэффициенту. В этом классе определяется несколько конструкторов. Ниже приведен применяемый здесь и далее конструктор.

```
Scale(double масштаб_по_ширине, double масштаб_по_высоте)
```

Здесь параметр *масштаб_по_ширине* обозначает коэффициент для масштабирования узла по ширине, а параметр *масштаб_по_высоте* — коэффициент для его масштабирования по высоте. Эти масштабные коэффициенты можно изменить после создания экземпляра класса `Scale`, вызвав методы `setX()` и `setY()`, общие формы которых приведены ниже. Как и прежде, параметр *масштаб_по_ширине* обозначает коэффициент для масштабирования узла по ширине, а параметр *масштаб_по_высоте* — коэффициент для его масштабирования по высоте.

```
final void setX(double масштаб_по_ширине)  
final void setY(double масштаб_по_высоте)
```

Демонстрация эффектов и преобразований

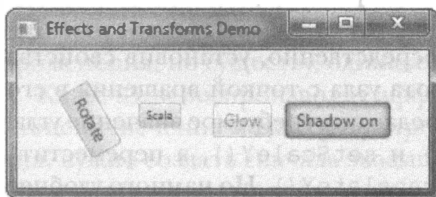


Рис. 35.18. Эффекты и преобразования над кнопками

В приведенном ниже примере демонстрируется применение эффектов и выполнение преобразований. С этой целью создаются четыре кнопки — `Rotate` (Вращение), `Scale` (Масштабирование), `Glow` (Свечение) и `Shadow` (Тень). Всякий раз, когда нажимается одна из этих кнопок, применяется соответствующий эффект или выполняется указанное преобразование над выбранной кнопкой, как показано на рис. 35.18.

Анализируя исходный код JavaFX-приложения из данного примера, вы сможете сами убедиться, насколько просто осуществить специальную настройку ГПИ своего приложения. Поэкспериментируйте с JavaFX-приложением из данного примера, опробова другие виды преобразований или эффектов или применяя их к другим типам узлов, кроме кнопок.

```
// Продемонстрировать вращение, масштабирование,
// свечение и внутреннюю тень

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;
import javafx.scene.transform.*;
import javafx.scene.effect.*;
import javafx.scene.paint.*;

public class EffectsAndTransformsDemo extends Application {
    double angle = 0.0;
    double glowVal = 0.0;
    boolean shadow = false;
    double scaleFactor = 1.0;

    // создать первоначальные эффекты и преобразования
    Glow glow = new Glow(0.0);
    InnerShadow innerShadow = new InnerShadow(10.0, Color.RED);
    Rotate rotate = new Rotate();
    Scale scale = new Scale(scaleFactor, scaleFactor);

    // создать четыре экранные кнопки
    Button btnRotate = new Button("Rotate");
    Button btnGlow = new Button("Glow");
    Button btnShadow = new Button("Shadow off");
    Button btnScale = new Button("Scale");

    public static void main(String[] args) {

        // запустить JavaFX-приложение, вызвав метод launch()
        launch(args);
    }

    // переопределить метод start()
    public void start(Stage myStage) {

        // присвоить заголовок подмосткам
        myStage.setTitle("Effects and Transforms Demo");
        // Демонстрация эффектов и преобразований

        // Использовать панель поточной компоновки FlowPane
        // в качестве корневого узла. В данном случае с
        // промежутками 10 по вертикали и по горизонтали
        FlowPane rootNode = new FlowPane(10, 10);

        // выровнять элементы управления по центру сцены
        rootNode.setAlignment(Pos.CENTER);

        // создать сцену
        Scene myScene = new Scene(rootNode, 300, 100);
```

```

// установить сцену на подмостках
myStage.setScene(myScene);

// задать первоначальный эффект свечения
btnGlow.setEffect(glow);

// ввести вращение в список преобразований для кнопки Rotate
btnRotate.getTransforms().add(rotate);

// ввести масштабирование в список преобразований для кнопки Scale
btnScale.getTransforms().add(scale);

// обработать события действия от кнопки Rotate
btnRotate.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        // Всякий раз, когда кнопка нажимается, она поворачивается
        // на 30 градусов вокруг своего центра.
        angle += 30.0;

        rotate.setAngle(angle);
        rotate.setPivotX(btnRotate.getWidth()/2);
        rotate.setPivotY(btnRotate.getHeight()/2);
    }
});

// обработать события действия от кнопки Scale
btnScale.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        // Всякий раз, когда кнопка нажимается, изменяется ее масштаб
        scaleFactor += 0.1;
        if(scaleFactor > 1.0) scaleFactor = 0.4;

        scale.setX(scaleFactor);
        scale.setY(scaleFactor);
    }
});

// обработать события действия от кнопки Glow
btnGlow.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        // Всякий раз, когда кнопка нажимается, изменяется
        // степень ее свечения
        glowVal += 0.1;
        if(glowVal > 1.0) glowVal = 0.0;

        // установить новое значение свечения
        glow.setLevel(glowVal);
    }
});

// обработать события действия от кнопки Shadow
btnShadow.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        // Всякий раз, когда кнопка нажимается, изменяется
        // состояние ее затенения
        shadow = !shadow;
        if(shadow) {
            btnShadow.setEffect(innerShadow);
            btnShadow.setText("Shadow on");
        } else {

```

```

        btnShadow.setEffect(null);
        btnShadow.setText("Shadow off");
    }
}
});

// ввести метку и кнопки в граф сцены
rootNode.getChildren().addAll(
    btnRotate, btnScale, btnGlow, btnShadow);

// показать подмости и сцену на них
myStage.show();
}
}

```

В завершение темы эффектов и преобразований следует заметить, что некоторые из них дают особенно привлекательные результаты применительно к узлу типа `Text`. Класс `Text` входит в состав пакета `javafx.scene.text` и создает узел, состоящий из текста. Благодаря тому что это узел, текстом легко манипулировать как единым целым, применяя к нему различные эффекты и выполняя над ним разнообразные преобразования.

Ввод всплывающих подсказок

Одним из самых распространенных элементов ГПИ является *всплывающая подсказка* — короткое сообщение, отображаемое при наведении курсора на элемент управления. В JavaFX совсем не трудно добавить всплывающую подсказку в любой элемент управления. Откровенно говоря, особых причин не пользоваться всплывающими подсказками не существует, принимая во внимание их преимущества и простоту внедрения в ГПИ приложения.

Чтобы ввести всплывающую подсказку, следует вызвать метод `setTooltip()`, определяемый в классе `Control`, который является базовым для всех элементов управления. Ниже приведена общая форма данного метода.

```
final void setTooltip(Tooltip подсказка)
```

В данном случае параметр *подсказка* обозначает экземпляр класса `Tooltip`, определяющего всплывающую подсказку. Как только всплывающая подсказка будет задана, она автоматически отображается при наведении курсора на элемент управления, не требуя больше ничего от программиста.

Класс `Tooltip` инкапсулирует всплывающую подсказку. Ниже приведен рассматриваемый здесь конструктор этого класса, где параметр *строка* обозначает сообщение, отображаемое во всплывающей подсказке.

```
Tooltip(String строка)
```

Чтобы опробовать всплывающие подсказки на практике, введите приведенный ниже фрагмент кода в приложение `CheckboxDemo` из представленного ранее примера. После этих дополнений всплывающие подсказки будут отображаться для каждого флажка.

```
cbWeb.setTooltip(new Tooltip("Deploy to Web"));
// Развернуть приложение в веб

```

```
cbDesktop.setTooltip(new Tooltip("Deploy to Desktop"));  
// Развернуть приложение в настольной системе  
cbMobile.setTooltip(new Tooltip("Deploy to Mobile"));  
// Развернуть приложение на мобильном устройстве
```

Отключение элементов управления

Прежде чем завершить рассмотрение элементов управления в JavaFX, следует упомянуть еще одну возможность манипулировать ими. Любой узел в графе сцены, в том числе и элемент управления, можно отключить под управлением прикладной программы. С этой целью нужно вызвать метод `setDisable()`, определяемый в классе `Node`. Ниже приведена общая форма данного метода.

```
final void setDisable(boolean отключение)
```

Если параметр *отключение* принимает логическое значение `true`, соответствующий элемент управления отключается, а иначе он включается. Следовательно, используя метод `setDisable()`, можно отключить элемент управления, а в дальнейшем включить его снова.

Введение в меню JavaFX

Меню являются неотъемлемой частью ГПИ многих приложений, поскольку они предоставляют пользователю доступ к основным функциональным возможностям прикладной программы. Более того, надлежащая реализация меню считается необходимой составляющей построения удачного ГПИ приложения. В связи с тем что меню играют главную роль во многих приложениях, в JavaFX обеспечивается обширная поддержка меню. И к счастью, эта поддержка не только мощная, но и рациональная.

Как поясняется в этой главе, меню в JavaFX имеют немало общего с меню в Swing, описанными в главе 33. Поэтому если вы уже знаете, как создаются меню в Swing, то научиться делать это в JavaFX вам будет нетрудно. Но у обеих разновидностей меню имеется ряд отличий, поэтому очень важно не спешить с выводами относительно системы меню в JavaFX.

Система меню в JavaFX поддерживает ряд основных элементов, включая следующие.

- Строка меню, в которой находится главное меню приложения.
- Стандартное меню, которое может содержать выбираемые пункты или другие меню, называемые подменю.
- Контекстное меню, которое нередко активизируется щелчком правой кнопкой мыши. Контекстные меню называют иначе всплывающими.

В системе меню JavaFX поддерживаются также *оперативные клавиши*, позволяющие выбирать пункты меню, не активизируя его, а также *мнемоника*, допускающая выбор пунктов с клавиатуры после раскрытия меню. Помимо обычных меню, в JavaFX поддерживается *панель инструментов*, предоставляющая быстрый доступ к функциональным возможностям прикладной программы и зачастую действующая параллельно пунктам меню.

Основные положения о меню

Система меню в JavaFX опирается на группу взаимосвязанных классов, входящих в состав пакета `javafx.scene.control`. В этой главе рассматриваются классы, перечисленные в табл. 36.1. Они составляют ядро системы меню в JavaFX. Несмотря

на то что в JavaFX допускается специальная настройка меню в широких пределах, как правило, классы меню применяются в исходном виде, поскольку они обеспечивают стандартный стиль оформления меню, который обычно и требуется.

Таблица 36.1. Основные классы меню в JavaFX

Класс	Описание
CheckMenuItem	Отмечаемый флажком пункт меню
ContextMenu	Всплывающее меню, которое обычно активизируется щелчком правой кнопкой мыши
Menu	Стандартное меню, состоящее из одного или нескольких пунктов типа MenuItem
MenuBar	Объект, содержащий меню верхнего уровня, т.е. главное меню приложения
MenuItem	Объект, наполняющий меню
RadioMenuItem	Отмечаемый кнопкой-переключателем пункт меню
SeparatorMenuItem	Визуальный разделитель пунктов меню

Сделаем краткий обзор совместного применения перечисленных выше классов. Чтобы создать главное меню приложения, нужно сначала получить экземпляр класса **MenuBar**. Проще говоря, этот класс служит контейнером для меню. В экземпляр класса **MenuBar** обычно вводятся экземпляры класса **Menu**, причем каждый объект типа **Menu** определяет отдельное меню. Это означает, что каждый объект типа **Menu** содержит один или несколько выбираемых пунктов. А пункты, отображаемые объектом типа **Menu**, в свою очередь, являются объектами класса **MenuItem**. Следовательно, класс **MenuItem** определяет пункт меню, выбираемый пользователем.

В меню можно вводить не только стандартные пункты, но и пункты, отмечаемые флажками или кнопками-переключателями. Такие пункты меню действуют параллельно с элементами управления флажками и кнопками-переключателями. В частности, отмечаемый флажком пункт меню создается средствами класса **CheckMenuItem**, а отмечаемый кнопкой-переключателем пункт меню — средствами класса **RadioMenuItem**. Оба эти класса расширяют класс **MenuItem**.

Служебный класс **SeparatorMenuItem** позволяет создавать линию, разделяющую пункты меню. Он наследует от класса **CustomMenuItem**, упрощающего встраивание других типов элементов управления в пункты меню. Класс **CustomMenuItem** расширяет класс **MenuItem**.

В отношении меню в JavaFX следует заметить, что класс **MenuItem** *не* наследует от класса **Node**. Следовательно, экземпляры класса **MenuItem** можно применять только в меню. Их нельзя иным образом внедрять в граф сцены. С другой стороны, класс **MenuBar** *наследует* от класса **Node**, что позволяет вводить строку меню в граф сцены.

Следует также иметь в виду, что класс **MenuItem** служит суперклассом для класса **Menu**. Это дает возможность создавать подменю, которые, по существу, являются одними меню, вложенными в другие. Чтобы создать подменю, нужно сначала создать объект типа **Menu** и заполнить его объектами типа **MenuItem**, представля-

ющими пункты меню, а затем ввести его в другой объект типа `Menu`. Этот процесс будет продемонстрирован в представленных далее примерах.

Когда выбирается пункт меню, генерируется событие действия. Текст, связанный с выбранным пунктом меню, обозначает его наименование. Следовательно, проанализировав наименование пункта меню при обработке событий действия от всех выбираемых пунктов меню в одном обработчике, можно выяснить, какой именно пункт меню был выбран. Безусловно, для обработки событий действия от каждого пункта меню можно также воспользоваться отдельными анонимными классами или лямбда-выражениями. В этом случае выбранный пункт меню уже известен, и поэтому нет нужды анализировать его наименование, чтобы выяснить, какой именно пункт меню был выбран.

Помимо меню, раскрывающихся из строки меню, можно создавать автономные контекстные меню, которые всплывают, когда активизируются. С этой целью следует сначала создать объект типа `ContextMenu`, а затем ввести в него объекты типа `MenuItem` в виде пунктов меню. Как правило, контекстное меню активизируется щелчком правой кнопкой мыши, когда курсор находится на том элементе управления, для которого определено всплывающее меню. Следует, однако, иметь в виду, что класс `ContextMenu` является производным не от класса `MenuItem`, а от класса `PopupControl`.

С меню связано еще одно средство, называемое *панелью инструментов*. В JavaFX панели инструментов поддерживаются в классе `ToolBar`, создающем автономный компонент, который нередко служит для быстрого доступа к функциональным возможностям, имеющимся в меню приложения. Например, панель инструментов может предоставлять быстрый доступ к командам форматирования, которые поддерживаются в текстовом редакторе.

Краткий обзор классов `MenuBar`, `Menu` и `MenuItem`

Прежде чем создавать меню, нужно знать некоторые особенности трех основных классов меню: `MenuBar`, `Menu` и `MenuItem`. Эти классы составляют тот минимум средств, которые требуются для создания главного меню приложения. Кроме того, объекты класса `MenuItem` служат для создания контекстных (т.е. всплывающих) меню. Таким образом, эти классы образуют основание системы меню в JavaFX.

Класс `MenuBar`

Класс `MenuBar`, по существу, служит контейнером для меню. Он реализует элемент управления, предоставляющий главное меню приложения. Как и все остальные классы элементов управления JavaFX, он наследует от класса `Node`. Следовательно, объект этого класса можно ввести как строку меню в граф сцены. У класса `MenuBar` имеется единственный конструктор по умолчанию. Таким образом, строка меню первоначально будет пустой, и поэтому ее придется наполнить меню, прежде чем воспользоваться ею. Как правило, у каждого приложения имеется одна и только одна строка меню.

В классе `MenuBar` определяется несколько методов, но зачастую применяется только метод `getMenus()`. Этот метод возвращает список меню, управление которыми осуществляется из строки меню. Именно в этот список и следует вводить создаваемые меню. Ниже приведена общая форма метода `getMenus()`.

```
final ObservableList<Menu> getMenus()
```

Для ввода экземпляра класса `Menu` в упомянутый выше список меню вызывается метод `add()`. А для одновременного ввода двух и более экземпляров класса `Menu` в этот список можно вызвать метод `addAll()`. Меню, вводимые в строку меню, располагаются слева направо в том порядке, в каком они вводятся. Если же меню требуется ввести в конкретном месте, то с этой целью следует воспользоваться приведенным ниже вариантом метода `add()`, где заданное меню вводится по указанному индексу, причем индексация меню начинается с нуля, а нулевой индекс обозначает крайнее слева меню.

```
void add(int индекс, Menu меню)
```

Иногда из строки меню требуется удалить меню, которое больше не нужно. С этой целью можно вызвать метод `remove()` для списка типа `ObservableList`, возвращаемого методом `getMenus()`. У этого метода имеются следующие общие формы:

```
void remove(Menu меню)
void remove(int индекс)
```

где параметр *меню* обозначает ссылку на удаляемое меню, а параметр *индекс* — указанный индекс удаляемого меню. Индексация меню начинается с нуля.

Иногда оказывается полезно получить количество элементов, находящихся в строке меню. С этой целью вызывается метод `size()` для списка типа `ObservableList`, возвращаемого методом `getMenus()`.

На заметку! Напомним, что класс `ObservableList` реализует интерфейс коллекций `List`, предоставляя тем самым доступ ко всем методам, определенным в этом интерфейсе.

Как только строка меню будет создана и наполнена, ее можно ввести в граф сцены обычным образом.

Класс Menu

Класс `Menu` инкапсулирует меню, наполняемое пунктами в виде объектов типа `MenuItem`. Как упоминалось ранее, этот класс наследует от класса `MenuItem`. Это означает, что одно меню типа `Menu` можно выбирать из другого. Следовательно, одно меню может служить подменю для другого. В классе `Menu` определяются три конструктора, но наиболее употребительным из них, вероятно, является следующий конструктор:

```
Menu(String имя)
```

Этот конструктор создает меню с заголовком, обозначаемым параметром *имя*. Вместе с текстом заголовка можно также указать изображение, используя следующий конструктор:

```
Menu(String имя, Node изображение)
```

где параметр *изображение* обозначает отображаемое изображение. Но в любом случае меню остается пустым до тех пор, пока в него не будут введены отдельные пункты. И наконец, присваивать наименование меню совсем не обязательно. Для создания безымянного меню можно воспользоваться следующим конструктором по умолчанию:

```
Menu()
```

В этом случае наименование и/или изображение можно ввести задним числом, вызвав метод `setText()` или `setGraphic()`.

В каждом меню поддерживается свой список пунктов меню, которые оно содержит. Чтобы добавить пункт в меню, нужно составить прежде список из его пунктов. С этой целью сначала вызывается метод `getItems()`, общая форма которого приведена ниже.

```
final ObservableList<MenuItem> getItems()
```

Этот метод возвращает список пунктов, связанных в данный момент с меню. Для ввода пунктов в этот список вызывается метод `add()` или `addAll()`. Среди прочих действий можно удалить пункт из меню, вызвав метод `remove()`, а также получить размер списка, вызвав метод `size()`.

И наконец, в список пунктов меню можно ввести разделитель, представленный объектом типа `SeparatorMenuItem`. Разделители помогают лучше организовать длинные меню, группируя связанные элементы вместе, а также отделить важные пункты в меню, например, пункт `Exit` в меню `File`.

Класс `MenuItem`

Класс `MenuItem` инкапсулирует пункт меню. Этот пункт может выбирать и связываться с конкретным действием в прикладной программе, например `Save` (Сохранить) или `Close` (Заккрыть), или же приводить к появлению подменю. В классе `MenuItem` определяются следующие конструкторы:

```
MenuItem()  
MenuItem(String имя)  
MenuItem(String имя, Node изображение)
```

Первый конструктор создает пустой пункт меню, второй позволяет указать наименование пункта меню, а третий конструктор — дополнить пункт меню изображением.

Когда пункт меню типа `MenuItem` выбирается, генерируется событие действия. Для его обработки можно зарегистрировать соответствующий обработчик, вызвав метод `setOnAction()` таким же образом, как и при обработке событий от кнопок. Ради удобства изложения ниже еще раз приведена общая форма этого метода.

```
final void setOnAction(EventHandler<ActionEvent> обработчик)
```

Здесь параметр *обработчик* обозначает регистрируемый обработчик событий. Чтобы инициировать событие действия в пункте меню, следует вызвать метод `fire()`.

В классе `MenuItem` определяется несколько методов. Наиболее употребительным из них является метод `setDisable()`, с помощью которого можно включать или отключать отдельные пункты меню. Ниже приведена общая форма данного метода.

```
final void setDisable(boolean отключение)
```

Если параметр *отключение* принимает логическое значение `true`, то соответствующий пункт меню отключается и становится недоступным для выбора. А если параметр *отключение* принимает логическое значение `false`, то соответствующий пункт меню включается. Используя метод `setDisable()`, можно делать доступными или недоступными отдельные пункты меню в зависимости режима работы прикладной программы.

Создание главного меню

Как правило, наиболее употребительным считается *главное меню*. Оно доступно из строки меню и определяет все (или почти все) функциональные возможности приложения. Как станет ясно в дальнейшем, JavaFX значительно упрощает создание главного меню и управление им. Ниже будет показано, как создать простое главное меню, а в последующих разделах — как ввести в него различные варианты выбора.

На заметку! С целью продемонстрировать сходства и отличия систем меню в Swing и JavaFX в этой главе представлены примеры, являющиеся переделанными вариантами примеров из главы 33. Если вы уже знакомы с системой меню в Swing, вам будет полезно сравнить ее с системой меню в JavaFX.

Процесс создания главного меню состоит из нескольких стадий. Сначала создается объект типа `MenuBar`, который будет содержать отдельные меню. Затем создается каждое меню, располагаемое в строке меню. В общем, создание меню начинается с создания объекта типа `Menu`, в который затем вводятся пункты меню в виде объектов типа `MenuItem`. Как только отдельные меню будут созданы, они вводятся в строку меню. После этого сама строка меню вводится в граф сцены. И наконец, каждый пункт меню должен быть дополнен приемником действий, реагирующим на событие действия, наступающее при выборе отдельного пункта из меню.

Процесс создания меню и управления ими станет понятнее, если продемонстрировать его на конкретном примере. Ниже приведено JavaFX-приложение, в котором создается простая строка меню, состоящая из трех меню. Первым из них является меню **File** (Файл), состоящее из выбираемых пунктов **Open** (Открыть), **Close** (Заккрыть), **Save** (Сохранить) и **Exit** (Выход). Второе меню называется **Options** (Параметры) и состоит из двух подменю: **Colors** (Цвета) и **Priority** (Приоритет). А третье меню называется **Help** (Справка) и состоит из единственного пункта **About** (О программе). Когда выбирается пункт меню, его наименование отображается на месте метки.

// Продемонстрировать меню

```
import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;
```

```
public class MenuDemo extends Application {

    Label response;

    public static void main(String[] args) {

        // запустить JavaFX-приложение, вызвав метод launch()
        launch(args);
    }

    // переопределить метод start()
    public void start(Stage myStage) {

        // присвоить заголовок подмосткам
        myStage.setTitle("Demonstrate Menus"); // Демонстрация меню

        // использовать панель граничной компоновки BorderPane
        // в качестве корневого узла
        BorderPane rootNode = new BorderPane();

        // создать сцену
        Scene myScene = new Scene(rootNode, 300, 300);

        // установить сцену на подмостках
        myStage.setScene(myScene);

        // создать метку для отображения результатов выбора из меню
        response = new Label("Menu Demo"); // // Демонстрация меню

        // создать строку меню
        MenuBar mb = new MenuBar();

        // создать меню File
        Menu fileMenu = new Menu("File"); // Файл
        MenuItem open = new MenuItem("Open"); // Открыть
        MenuItem close = new MenuItem("Close"); // Закрыть
        MenuItem save = new MenuItem("Save"); // Сохранить
        MenuItem exit = new MenuItem("Exit"); // Выход
        fileMenu.getItems().addAll(
            open, close, save, new SeparatorMenuItem(), exit);

        // ввести меню File в строку меню
        mb.getMenus().add(fileMenu);

        // создать меню Options
        Menu optionsMenu = new Menu("Options"); // Параметры

        // создать подменю Colors
        Menu colorsMenu = new Menu("Colors"); // Цвета
        MenuItem red = new MenuItem("Red"); // Красный
        MenuItem green = new MenuItem("Green"); // Зеленый
        MenuItem blue = new MenuItem("Blue"); // Синий

        colorsMenu.getItems().addAll(red, green, blue);
        optionsMenu.getItems().add(colorsMenu);

        // создать подменю Priority
        Menu priorityMenu = new Menu("Priority"); // Приоритет
        MenuItem high = new MenuItem("High"); // Высокий
        MenuItem low = new MenuItem("Low"); // Низкий
        priorityMenu.getItems().addAll(high, low);
    }
}
```

```

optionsMenu.getItems().add(priorityMenu);

// ввести разделитель
optionsMenu.getItems().add(new SeparatorMenuItem());

// создать пункт меню Reset
MenuItem reset = new MenuItem("Reset"); // Сбросить
optionsMenu.getItems().add(reset);

// ввести меню Options в строку меню
mb.getMenus().add(optionsMenu);

// создать меню Help
Menu helpMenu = new Menu("Help"); // Справка
MenuItem about = new MenuItem("About"); // О программе
helpMenu.getItems().add(about);

// ввести меню Help в строку меню
mb.getMenus().add(helpMenu);

// создать один приемник действий для обработки всех
// событий действия, наступающих в меню
EventHandler<ActionEvent> MEHandler =
    new EventHandler<ActionEvent>() {
        public void handle(ActionEvent ae) {
            String name = ((MenuItem)ae.getTarget()).getText();

            // выйти из программы, если выбран пункт меню Exit
            if(name.equals("Exit")) Platform.exit();

            response.setText( name + " selected");
            // Выбран указанный пункт меню
        }
    };

// установить приемники действий от пунктов меню
open.setOnAction(MEHandler);
close.setOnAction(MEHandler);
save.setOnAction(MEHandler);
exit.setOnAction(MEHandler);
red.setOnAction(MEHandler);
green.setOnAction(MEHandler);
blue.setOnAction(MEHandler);
high.setOnAction(MEHandler);
low.setOnAction(MEHandler);
reset.setOnAction(MEHandler);
about.setOnAction(MEHandler);

// ввести строку меню в верхней области панели
// граничной компоновки, а метку response –
// в центре этой панели
rootNode.setTop(mb);
rootNode.setCenter(response);

// показать подмости и сцену на них
myStage.show();
}
}

```

Вид главного меню в окне JavaFX-приложения из данного примера приведен на рис. 36.1.

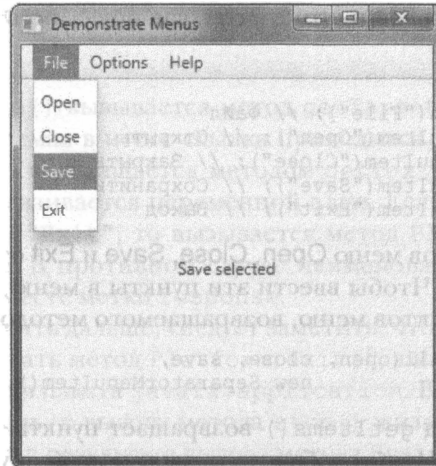


Рис. 36.1. Вид главного меню в окне приложения MenuDemo

Рассмотрим подробнее, каким образом в данном примере приложения создаются меню. Прежде всего следует заметить, что в классе `MenuDemo` используется экземпляр класса `BorderPane` в качестве корневого узла. Класс `BorderPane` действует аналогично классу `BorderLayout` из библиотеки AWT, обсуждавшемуся в главе 26. В этом классе определяется окно со следующими пятью областями: верхней, нижней, левой, правой и центральной. Ниже приведены методы для установки узла, назначаемого для каждой из этих областей.

```
final void setTop(Node узел)
final void setBottom(Node узел)
final void setLeft(Node узел)
final void setRight(Node узел)
final void setCenter(Node узел)
```

Здесь параметр *узел* обозначает элемент (например, элемент управления), который должен отображаться в каждой из упомянутых выше областей окна. Далее в рассматриваемом здесь примере приложения строка меню располагается в верхней области окна, а метка, отображающая результат выбора из меню, — в центральной области. Верхнее расположение строки меню гарантирует, что оно будет отображаться в верхней части окна приложения, автоматически изменяя свои размеры по ширине окна. Именно поэтому в примерах меню из этой главы применяется класс `BorderPane`. Разумеется, для компоновки меню можно воспользоваться и другими классами, например `VBox`.

Большая часть кода данного приложения служит для построения строки меню, размещаемых в ней меню, а также отдельных пунктов меню, и поэтому этот код требует более подробного рассмотрения. Сначала создается строка меню, а ссылка на нее присваивается переменной `mb`, как показано ниже.

```
// создать строку меню
MenuBar mb = new MenuBar();
```

На данной стадии строка меню пока еще пуста. Она будет заполнена отдельными меню, когда они будут созданы.

Затем создается меню **File** и его пункты, как показано в следующем фрагменте кода:

```
// создать меню File
Menu fileMenu = new Menu("File"); // Файл
MenuItem open = new MenuItem("Open"); // Открыть
MenuItem close = new MenuItem("Close"); // Закрыть
MenuItem save = new MenuItem("Save"); // Сохранить
MenuItem exit = new MenuItem("Exit"); // Выход
```

Наименования пунктов меню **Open**, **Close**, **Save** и **Exit** отображаются на местах их выбора из меню **File**. Чтобы ввести эти пункты в меню **File**, вызывается метод `addAll()` для списка пунктов меню, возвращаемого методом `getItems()`:

```
fileMenu.getItems().addAll(open, close, save,
                           new SeparatorMenuItem(), exit);
```

Напомним, что метод `getItems()` возвращает пункты меню, связанные с экземпляром класса `Menu`. Чтобы добавить пункты в меню, нужно составить прежде список из его пунктов. Следует также заметить, что для визуального отделения пункта **Exit** от остальных пунктов меню **File** служит разделитель.

И наконец, меню **File** вводится в строку меню в следующем фрагменте кода:

```
// ввести меню File в строку меню
mb.getMenus().add(fileMenu);
```

По завершении этого фрагмента кода строка меню будет содержать единственный элемент: меню **File**. В свою очередь, меню **File** будет содержать следующие выбираемые пункты: **Open**, **Close**, **Save** и **Exit**.

Меню **Options** создается таким же образом, как и меню **File**. Но оно состоит из двух подменю **Colors** и **Priority**, а также отдельного пункта **Reset**. Как пояснялось ранее, класс `Menu` наследует от класса `MenuItem`, и поэтому одно меню типа `Menu` может быть введено в другое. Именно таким образом и создаются подменю. А пункт **Reset** вводится последним. После этого меню **Options** вводится в строку меню. Аналогичным образом создается и меню **Help**.

Как только будут созданы все упомянутые выше меню, создается обработчик событий `EventHandler` типа `ActionEvent`, предназначенный для реагирования на события выбора из меню и последующей их обработки. В целях демонстрации все события выбора из меню обрабатываются одним обработчиком, но в реальном приложении нередко оказывается проще указать отдельный обработчик событий для каждого выбора из меню, используя анонимные внутренние классы или лямбда-выражения. Ниже показано, каким образом создается обработчик событий типа `ActionEvent`.

```
// создать один приемник действий для обработки всех
// событий действия, наступающих в меню
EventHandler<ActionEvent> MEHandler =
    new EventHandler<ActionEvent>() {
    public void handle(ActionEvent ae) {
        String name = ((MenuItem)ae.getTarget()).getText();

        // выйти из программы, если выбран пункт меню Exit
        if(name.equals("Exit")) Platform.exit();

        response.setText( name + " selected");
```

```

    // Выбран указанный пункт меню
}
};

```

В теле метода `handle()` вызывается метод `getTarget()` для получения адреса события. Возвращаемая в итоге ссылка приводится к типу `MenuItem`, а наименование пункта меню возвращается методом `getText()`. Полученная в итоге символьная строка присваивается переменной `name`. Если эта переменная содержит символьную строку "Exit", то вызывается метод `Platform.exit()`, чтобы завершить приложение. В противном случае наименование выбранного пункта меню отображается на месте метки `response`.

Прежде чем продолжить дальше, следует заметить, что для завершения JavaFX-приложения нужно вызвать метод `Platform.exit()`, а не `System.exit()`. Класс `Platform` входит в состав пакета `javafx.application`. Вызов его метода `exit()` приводит, в свою очередь, к вызову метода `stop()` жизненного цикла приложения, чего не происходит в результате вызова метода `System.exit()`.

И наконец, обработчик `MEHandler` регистрируется как обработчик событий действия от каждого пункта меню в следующем фрагменте кода:

```

// установить приемники действий от пунктов меню
open.setOnAction(MEHandler);
close.setOnAction(MEHandler);
save.setOnAction(MEHandler);
exit.setOnAction(MEHandler);
red.setOnAction(MEHandler);
green.setOnAction(MEHandler);
blue.setOnAction(MEHandler);
high.setOnAction(MEHandler);
low.setOnAction(MEHandler);
reset.setOnAction(MEHandler);
about.setOnAction(MEHandler);

```

Как видите, ни один из приемников действий не вводится в пункты меню `Colors` или `Priority`. Ведь они на самом деле являются подменю, а не выбираемыми пунктами меню.

И наконец, в корневой узел вводится строка меню, как показано ниже. В итоге строка меню размещается в верхней области окна.

```
rootNode.setTop(mb);
```

Можете на данной стадии поэкспериментировать немного с приложением `MenuDemo`, попробовав ввести в него еще одно меню или дополнительные пункты в уже существующее меню. Это поможет вам лучше понять основные принципы создания меню, прежде чем продолжить дальше, поскольку данное приложение будет постепенно усложняться по ходу изложения материала этой главы.

Ввод мнемоники и оперативных клавиш в меню

Меню, созданное в предыдущем примере, вполне работоспособно, но его можно усовершенствовать. Меню реальных приложений обычно поддерживают клавиатурные сокращения, предоставляя опытным пользователям возможность быстро выбирать пункты меню. Клавиатурные сокращения принимают две формы:

мнемонику и оперативные клавиши. Применительно к меню мнемоника определяет клавишу, нажатием которой выбирается отдельный пункт активного меню. Следовательно, *мнемоника* позволяет выбирать с клавиатуры пункты того меню, которое уже отображается, а *оперативная клавиша* — сделать то же самое, не активизируя предварительно меню.

Оперативная клавиша может быть связана с меню типа `Menu` или пунктом меню типа `MenuItem`. Для ее указания вызывается метод `setAccelerator()`, общая форма которого приведена ниже.

```
final void setAccelerator(KeyCombination комбинация_клавиш)
```

Здесь параметр *комбинация_клавиш* обозначает комбинацию клавиш, нажимаемых для выбора пункта меню. Класс `KeyCombination` инкапсулирует комбинацию клавиш, например `<Ctrl+S>`. Он входит в состав пакета `javafx.scene.input`.

В классе `KeyCombination` определяются два защищенных (`protected`) конструктора, но зачастую вместо них применяется фабричный метод `keyCombination()`. Ниже приведена общая форма данного метода.

```
static KeyCombination keyCombination(String клавиши)
```

В данном случае параметр *клавиши* обозначает символьную строку, в которой указывается конкретная комбинация клавиш. Обычно она состоит из модифицирующей клавиши, в частности `<Ctrl>`, `<Alt>`, `<Shift>` или `<Meta>`, и клавиши буквы, например `<S>`. Имеется специальное значение `shortcut`, предназначенное для обозначения клавиши `<Ctrl>` в Windows и клавиши `<Meta>` в Mac OS, а в других операционных системах оно преобразуется в типичную для них комбинацию клавиш. Так, если требуется указать комбинацию клавиш `<Ctrl+S>` для оперативного выбора пункта меню **Save**, то при вызове метода `keyCombination()` следует указать символьную строку `"shortcut+S"` в качестве его параметра. Таким образом, данная комбинация клавиш будет действовать в Windows, Mac OS и других операционных системах.

В приведенном ниже фрагменте кода оперативные клавиши вводятся в меню **File**, созданное в примере приложения `MenuDemo` из предыдущего раздела. После внесения этих изменений в исходный код данного приложения можно оперативно выбирать пункты меню **File**, нажимая комбинации клавиш `<Ctrl+O>`, `<Ctrl+C>`, `<Ctrl+S>` или `<Ctrl+E>`, даже не открывая это меню.

```
// ввести оперативные клавиши для быстрого выбора пунктов меню File
open.setAccelerator(KeyCombination.keyCombination("shortcut+O"));
close.setAccelerator(KeyCombination.keyCombination("shortcut+C"));
save.setAccelerator(KeyCombination.keyCombination("shortcut+S"));
exit.setAccelerator(KeyCombination.keyCombination("shortcut+E"));
```

Мнемонику нетрудно указать как для пункта меню типа `MenuItem`, так и для меню типа `Menu`. Для этого достаточно предварить наименование меню или пункта меню знаком подчеркивания. Например, чтобы ввести мнемонику **F** в меню **File** приложения `MenuDemo`, это меню достаточно объявить следующим образом:

```
Menu fileMenu = new Menu("_File"); // теперь в наименовании меню
// определяется мнемоника
```

После внесения этого изменения в исходный код данного приложения для быстрого выбора меню **File** достаточно нажать сначала клавишу `<Alt>`, а затем клави-

шу <F>. Но мнемоника активизируется лишь в том случае, если включен режим ее синтаксического анализа, что делается по умолчанию. Включить или выключить этот режим можно, вызвав метод `setMnemonicParsing()`, общая форма которого приведена ниже.

```
final void setMnemonicParsing(boolean включить)
```

Если параметр *включить* принимает логическое значение `true`, то режим синтаксического анализа мнемоники включен. В противном случае этот режим выключен.

После внесения описанных выше изменений в исходный код приложения MenuDemo меню файл File будет выглядеть так, как показано на рис. 36.2.

Ввод изображений в пункты меню

Изображения можно вводить в пункты меню или использовать их вместо текста. Самый простой способ ввести изображение в пункт меню — указать его при создании этого пункта меню с помощью следующего конструктора:

```
MenuItem(String имя, Node изображение)
```

Этот конструктор создает пункт меню, обозначаемый параметром *имя* и отображающий указанное *изображение*. Например, в приведенном ниже фрагменте кода создается пункт меню About вместе с указанным изображением значка.

```
ImageView aboutIV = new ImageView("aboutIcon.gif");
MenuItem about = new MenuItem("About", aboutIV);
```

После этого дополнения исходного кода приложения MenuDemo указанное изображение значка aboutIV появится рядом с текстом "About" в соответствующем пункте меню Help, как показано на рис. 36.3.

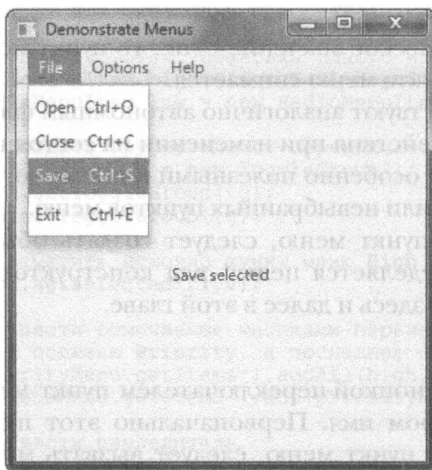


Рис. 36.2. Меню File, усовершенствованное мнемоникой и оперативными клавишами в окне приложения MenuDemo

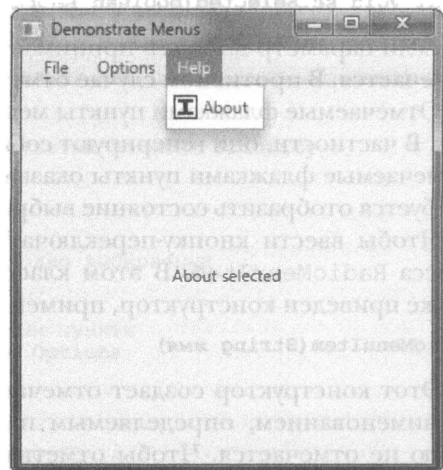


Рис. 36.3. Пункт меню About в окне приложения MenuDemo после ввода значка

И последнее замечание: изображение можно ввести в пункт меню и после его создания, вызвав метод `setGraphic()`. Это дает возможность изменять изображение по ходу выполнения прикладной программы.

Классы `RadioMenuItem` и `CheckMenuItem`

В приведенных выше примерах демонстрировались наиболее употребительные типы пунктов меню, но в JavaFX определяются также два других типа отмечаемых пунктов меню: с флажками и кнопками-переключателями. Эти отмечаемые пункты меню упрощают построение ГПИ, делая доступными из меню такие функциональные возможности, для предоставления которых в противном случае потребовались бы дополнительные автономные компоненты. Кроме того, наличие флажков и кнопок-переключателей в меню иногда кажется вполне естественным для выбора конкретного ряда средств. Но независимо от конкретных причин для применения флажков и кнопок-переключателей в меню каркас JavaFX позволяет сделать это довольно просто, как поясняется ниже.

Для ввода отмечаемого флажком пункта в меню служит класс `CheckMenuItem`, в котором определяются три конструктора, действующих параллельно с конструкторами из класса `MenuItem`. Ниже приведен применяемый здесь и далее конструктор данного класса.

```
CheckMenuItem(String имя)
```

Здесь параметр *имя* обозначает наименование пункта меню. Исходно флажок находится в сброшенном состоянии. Если же требуется явно указать исходное состояние флажка, следует вызвать метод `setSelected()`, общая форма которого приведена ниже.

```
final void setSelected(boolean выбрать)
```

Если параметр *выбрать* принимает логическое значение `true`, то пункт меню отмечается. В противном случае отметка пункта меню снимается.

Отмечаемые флажками пункты меню действуют аналогично автономным флажкам. В частности, они генерируют события действия при изменении их состояния. Отмечаемые флажками пункты оказываются особенно полезными в тех меню, где требуется отобразить состояние выбранных или невыбранных пунктов меню.

Чтобы ввести кнопку-переключатель в пункт меню, следует создать объект класса `RadioMenuItem`. В этом классе определяется целый ряд конструкторов. Ниже приведен конструктор, применяемый здесь и далее в этой главе.

```
RadioMenuItem(String имя)
```

Этот конструктор создает отмечаемый кнопкой-переключателем пункт меню с наименованием, определяемым параметром *имя*. Первоначально этот пункт меню не отмечается. Чтобы отметить этот пункт меню, следует вызвать метод `setSelected()`, передав ему логическое значение `true` в качестве аргумента, как это делается и для отмечаемых флажками пунктов меню.

Отмечаемый пункт меню типа `RadioMenuItem` действует аналогично автономной кнопке-переключателю, генерируя событие от элемента и событие действия.

Как и автономные кнопки-переключатели, в меню кнопки-переключатели должны быть объединены в группу, чтобы они действовали в режиме взаимоисключающего выбора.

Классы `CheckMenuItem` и `RadioMenuItem` наследуют от класса `MenuItem`, и поэтому функциональные возможности последнего доступны каждому из них. Помимо дополнительных возможностей флажков и кнопок-переключателей, отмечаемые пункты меню ничем не отличаются от обычных пунктов меню.

Чтобы опробовать на практике пункты меню, отмечаемые флажками и кнопками-переключателями, удалите сначала из примера приложения `MenuDemo` фрагмент кода, в котором создается меню `Options`. Затем подставьте приведенный ниже фрагмент кода, в котором создаются подменю `Colors` и `Priority` с флажками и кнопками-переключателями соответственно.

```
// создать меню Options
Menu optionsMenu = new Menu("Options");

// создать подменю Colors
Menu colorsMenu = new Menu("Colors");

// использовать отмечаемые флажками пункты меню, чтобы
// пользователь мог выбрать сразу несколько цветов
CheckMenuItem red = new CheckMenuItem("Red");
CheckMenuItem green = new CheckMenuItem("Green");
CheckMenuItem blue = new CheckMenuItem("Blue");
colorsMenu.getItems().addAll(red, green, blue);
optionsMenu.getItems().add(colorsMenu);

// выбрать по умолчанию зеленый цвет
green.setSelected(true);

// создать подменю Priority
Menu priorityMenu = new Menu("Priority");

// использовать отмечаемые кнопками-переключателями пункты
// меню для установки приоритета. Благодаря этому в меню не
// только отображается установленный приоритет, но и
// гарантируется установка одного и только одного приоритета
RadioMenuItem high = new RadioMenuItem("High");
RadioMenuItem low = new RadioMenuItem("Low");

// создать группу кнопок-переключателей в пунктах подменю Priority
ToggleGroup tg = new ToggleGroup();
high.setToggleGroup(tg);
low.setToggleGroup(tg);

// отметить исходно пункт меню High как исходно выбираемый
high.setSelected(true);

// ввести отмечаемые кнопками-переключателями пункты
// в подменю Priority, а последнее – в меню Options
priorityMenu.getItems().addAll(high, low);
optionsMenu.getItems().add(priorityMenu);

// ввести разделитель
optionsMenu.getItems().add(new SeparatorMenuItem());

// создать пункт меню Reset
MenuItem reset = new MenuItem("Reset");
optionsMenu.getItems().add(reset);
```

```
// и наконец, ввести меню Options в строку меню
mb.getMenus().add(optionsMenu);
```

После этой замены подменю **Colors** будет выглядеть так, как показано на рис. 36.4, а подменю **Priority** — как показано на рис. 36.5.

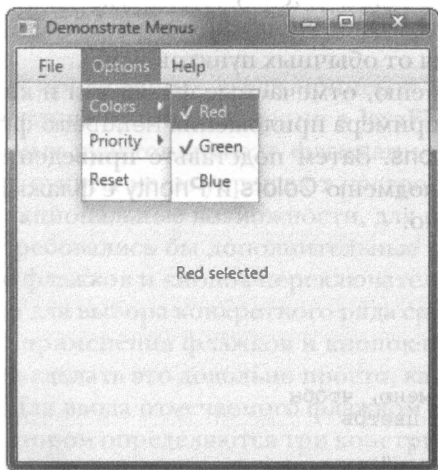


Рис. 36.4. Вид подменю **Colors** с отмеченными пунктами в окне приложения MenuDemo

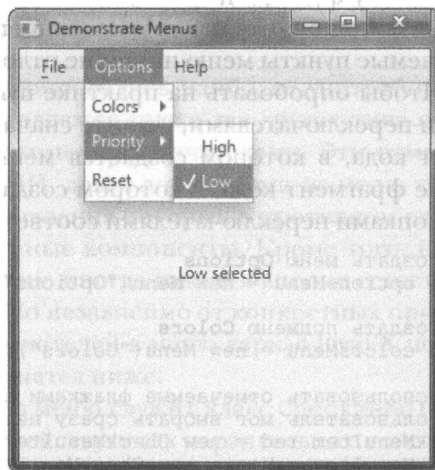


Рис. 36.5. Вид подменю **Priority** с отмеченным пунктом в окне приложения MenuDemo

Создание контекстного меню

Контекстное меню служит весьма распространенной альтернативой или дополнением строки меню. Как правило, контекстное меню активируется щелчком правой кнопкой мыши на элементе управления. В JavaFX всплывающие контекстные меню поддерживаются в классе `ContextMenu`. Его непосредственным суперклассом служит класс `PopupControl`, а косвенным суперклассом — класс `javafx.stage.PopupWindow`, предоставляющим большую часть его основных функциональных возможностей.

У класса `ContextMenu` имеются два конструктора. Здесь и далее в этой главе применяется следующий конструктор:

```
ContextMenu(MenuItem ... пункты_меню)
```

где параметр *пункты_меню* обозначает те пункты меню, которые составляют контекстное меню. А второй конструктор класса `ContextMenu` создает пустое меню, в которое должны быть введены нужные пункты.

В общем, контекстные меню создаются таким же образом, как и обычные меню. Сначала создаются отдельные пункты, а затем они вводятся в меню. Обработка результатов выбора пунктов контекстного меню выполняется тем же самым способом: приемом событий действия. А отличаются контекстные меню от обычных процессом их активизации.

Связать контекстное меню с элементом управления не составляет особого труда. С этой целью достаточно вызвать для конкретного элемента управления метод

`setContextMenu()`, передав ему ссылку на то меню, которое должно всплывать после щелчка правой кнопкой мыши на данном элементе управления. Ниже приведена общая форма метода `setContextMenu()`, где параметр *меню* обозначает контекстное меню, связанное с вызывающим элементом управления.

```
final void setContextMenu(ContextMenu меню)
```

Ниже приведен фрагмент кода, который требуется ввести в приложение `MenuDemo`, чтобы продемонстрировать создание и применение контекстного меню. Это контекстное меню называется **Edit** (Правка), содержит пункты **Cut** (Вырезать), **Copy** (Скопировать) и **Paste** (Вставить) и связывается с элементом управления текстовым полем. Оно всплывает, если щелкнуть правой кнопкой мыши на текстовом поле. Введите сначала в исходный код приложения `MenuDemo` следующий фрагмент кода, чтобы создать данное контекстное меню:

```
// создать пункты контекстного меню
MenuItem cut = new MenuItem("Cut"); // Вырезать
MenuItem copy = new MenuItem("Copy"); // Скопировать
MenuItem paste = new MenuItem("Paste"); // Вставить

// создать контекстное (т.е. всплывающее) меню с пунктами
// для выбора команд редактирования
final ContextMenu editMenu = new ContextMenu(cut, copy, paste);
```

Сначала в данном фрагменте кода создаются отдельные пункты (объекты типа `MenuItem`), составляющие меню. А затем создается контекстное меню `editMenu` (экземпляр класса `ContextMenu`), наполняемое этими пунктами.

Далее введите обработчик событий действия в пункты контекстного меню, как показано ниже.

```
cut.setOnAction(MEHandler);
copy.setOnAction(MEHandler);
paste.setOnAction(MEHandler);
```

На этом создание контекстного меню завершается, но оно еще не связано с конкретным элементом управления. Поэтому введите в исходный код приложения `MenuDemo` следующий фрагмент кода, чтобы создать текстовое поле:

```
// создать текстовое поле, задав ширину его столбца равной 20
TextField tf = new TextField();
tf.setPrefColumnCount(20);
```

Далее установите контекстное меню в текстовом поле, как показано ниже. Если теперь щелкнуть правой кнопкой мыши на данном текстовом поле, всплывет контекстное меню.

```
// ввести контекстное меню в текстовое поле
tf.setContextMenu(editMenu);
```

Но чтобы это действительно произошло в окне приложения `MenuDemo`, нужно сначала создать панель поточной компоновки и ввести в нее текстовое поле, а также метку ответной реакции на действия пользователя и затем расположить эту панель по центру панели граничной компоновки типа `BorderPane`. Этот шаг необходимо предпринять потому, что в любой области панели граничной компоновки типа `BorderPane` можно расположить лишь один узел графа сцены. С этой целью удалите из исходного кода приложения `MenuDemo` следующую строку:


```
rootNode.setCenter(response);
```

А вместо нее введите следующий фрагмент кода:

```
// создать панель поточной компоновки, которая должна содержать
// текстовое поле и метку ответной реакции на действия пользователя
FlowPane fpRoot = new FlowPane(10, 10);

// выровнять элементы управления по центру сцены
fpRoot.setAlignment(Pos.CENTER);

// ввести текстовое поле и метку на панели поточной компоновки
fpRoot.getChildren().addAll(response, tf);

// расположить панель поточной компоновки по центру панели
// граничной компоновки
rootNode.setCenter(fpRoot);
```

Разумеется, строка меню по-прежнему располагается вдоль верхнего края панели граничной компоновки. После внесения упомянутых выше изменений контекстное меню должно появиться в окне приложения MenuDemo, если щелкнуть правой кнопкой мыши на текстовом поле (рис. 36.6).

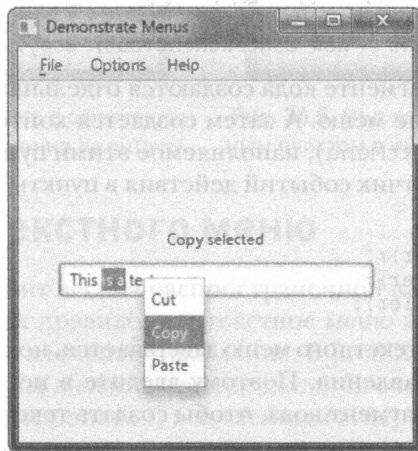


Рис. 36.6. Контекстное меню, всплывающее в окне приложения MenuDemo после щелчка правой кнопкой мыши на текстовом поле

Контекстное меню можно также связать непосредственно со сценой. С этой целью можно, например, вызвать метод `setOnContextMenuRequested()` для корневого узла сцены. Этот метод определяется в классе `Node` следующим образом:

```
final void setOnContextMenuRequested(
    EventHandler<? super ContextMenuEvent> обработчик_событий)
```

где параметр *обработчик_событий* обозначает тот обработчик событий, который должен вызываться при получении запроса на всплывание контекстного меню. В данном случае обработчик событий должен вызвать метод `show()`, определенный в классе `ContextMenu`, чтобы отобразить контекстное меню. Здесь и далее применяется следующая общая форма метода `show()`:

```
final void show(Node узел, double верх_X, double верх_Y)
```

где параметр *узел* обозначает элемент управления, с которым связано контекстное меню, а параметры *верх_X* и *верх_Y* — координаты верхнего левого угла контекстного меню относительно экрана. Как правило, в качестве этих двух параметров методу `show()` передаются координаты точки, в которой был произведен щелчок правой кнопкой мыши. Для получения координат этой точки вызываются методы `getScreenX()` и `getScreenY()`, определенные в классе `ContextMenuEvent`. Ниже приведены их общие формы. Результаты вызова этих методов передаются в качестве параметров *верх_X* и *верх_Y* методу `show()`.

```
final double getScreenX()
final double getScreenY()
```

Все сказанное выше можно применить на практике, введя контекстное меню в корневой узел графа сцены. Если после этого щелкнуть правой кнопкой мыши на любом месте в сцене, появится контекстное меню. Для этого введите сначала следующий фрагмент кода в исходный код приложения `MenuDemo`:

```
// ввести контекстное меню непосредственно в граф сцены
rootNode.setOnContextMenuRequested(
    new EventHandler<ContextMenuEvent>() {
        public void handle(ContextMenuEvent ae) {
            // отобразить контекстное меню в том месте, где был
            // произведен щелчок кнопкой мыши
            editMenu.show(rootNode, ae.getScreenX(), ae.getScreenY());
        }
    });
```

Затем объявите корневой узел `rootNode` как `final` (т.е. завершенный), чтобы сделать его доступным в анонимном внутреннем классе. После внесения всех этих изменений и дополнений контекстное меню может быть активизировано щелчком правой кнопкой мыши на любом месте в сцене приложения `MenuDemo`. В качестве примера на рис. 36.7 показано контекстное меню, всплывающее после щелчка правой кнопкой мыши слева вверху в окне данного приложения.

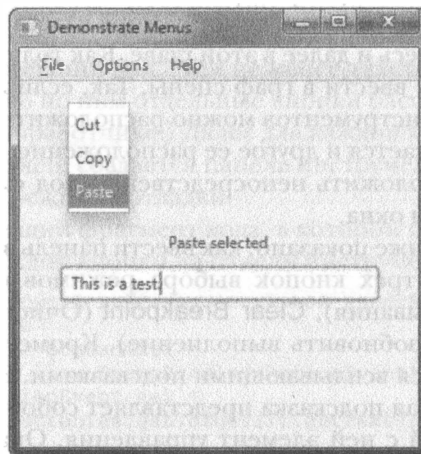


Рис. 36.7. Контекстное меню, всплывающее после щелчка правой кнопкой мыши слева вверху в окне приложения `MenuDemo`

Создание панели инструментов

Панель инструментов является компонентом, служащим как альтернативой, так и дополнением меню. Такая панель состоит из ряда кнопок (или других компонентов), предоставляющих пользователю возможность немедленного доступа к различным параметрам и режимам работы программы. Например, панель инструментов может содержать кнопки для выбора параметров шрифтового оформления текста, в том числе полужирного или наклонного начертания, выделения или подчеркивания текста. Эти параметры можно выбрать, не раскрывая меню. Как правило, кнопки на панели инструментов обозначены значками, а не текстовыми надписями, хотя допускается и то и другое. Кроме того, кнопки на панели инструментов нередко снабжаются всплывающим подсказками.

В JavaFX панели инструментов являются экземплярами класса `ToolBar`. В этом классе определяются два конструктора:

```
ToolBar()  
ToolBar(Node ... узлы)
```

Первый конструктор создает пустую горизонтальную панель инструментов, а второй конструктор — горизонтальную панель, содержащую указанные *узлы*, которые обычно представлены в некоторой форме кнопки. Если же требуется вертикальная панель инструментов, следует вызвать метод `setOrientation()` для уже имеющейся панели инструментов. Ниже приведена общая форма данного метода, где параметр *ориентация* должен принимать значение одной из констант: `Orientation.VERTICAL` или `Orientation.HORIZONTAL`.

```
final void setOrientation(Orientation ориентация)
```

Кнопки (или другие компоненты) вводятся на панели инструментов таким же образом, как и в строке меню. Для этого достаточно вызвать метод `add()` по ссылке, возвращаемой методом `getItems()`. Но зачастую элементы, составляющие панель инструментов, проще указать в конструкторе класса `ToolBar`. Именно такой подход и применяется здесь и далее в этой главе. Как только панель инструментов будет создана, ее следует ввести в граф сцены. Так, если применяется граничная компоновка, то панель инструментов можно расположить в нижней области этой компоновки, хотя допускается и другое ее расположение. В частности, панель инструментов можно расположить непосредственно под строкой меню или вдоль одной из боковых сторон окна.

В качестве примера ниже показано, как ввести панель в приложение `MenuDemo`. Эта панель состоит из трех кнопок выбора режимов отладки: **Set Breakpoint** (Установить точку прерывания), **Clear Breakpoint** (Очистить точку прерывания) и **Resume Execution** (Возобновить выполнение). Кроме того, кнопки на панели инструментов снабжаются всплывающими подсказками. Как пояснялось в предыдущей главе, всплывающая подсказка представляет собой небольшое сообщение, описывающее связанный с ней элемент управления. Она автоматически всплывает на некоторое время при наведении курсора мыши на элемент управления. Чтобы снабдить пункт меню или элемент управления на панели инструментов всплывающей подсказкой, следует вызвать метод `setTooltip()`. Всплывающие

подсказки особенно удобны для элементов управления, представленных только значками на панели инструментов, поскольку не всегда удастся оформить такие значки, которые интуитивно понятны всем пользователям.

Прежде всего введите следующий фрагмент кода, в котором создается панель инструментов отладки:

```
// определить панель инструментов;
// создать сначала кнопки на этой панели
Button btnSet = new Button("Set Breakpoint",
                           new ImageView("setBP.gif"));
Button btnClear = new Button("Clear Breakpoint",
                             new ImageView("clearBP.gif"));
Button btnResume = new Button("Resume Execution",
                              new ImageView("resume.gif"));

// затем отключить текстовые надписи на кнопках
btnSet.setContentDisplay(ContentDisplay.GRAPHIC_ONLY);
btnClear.setContentDisplay(ContentDisplay.GRAPHIC_ONLY);
btnResume.setContentDisplay(ContentDisplay.GRAPHIC_ONLY);

// задать всплывающие подсказки
btnSet.setTooltip(new Tooltip("Set a breakpoint."));
// Установить точку прерывания
btnClear.setTooltip(new Tooltip("Clear a breakpoint."));
// Очистить точку прерывания
btnResume.setTooltip(new Tooltip("Resume execution."));
//Возобновить выполнение

// создать панель инструментов
ToolBar tbDebug = new ToolBar(btnSet, btnClear, btnResume);
```

Рассмотрим подробнее приведенный выше фрагмент кода. Сначала в нем создаются три кнопки для выбора режимов отладки на панели инструментов, причем у каждой из них имеется свой значок вместо текстовой надписи. Затем для каждой кнопки отключается режим отображения текстовой надписи, для чего вызывается метод `setContentDisplay()`. Любопытно, что отображение текстовых надписей на кнопках можно было бы оставить, но тогда панель инструментов имела бы не совсем привычный внешний вид. (Тем не менее текстовые надписи на кнопках нужны, поскольку именно по ним отдельные кнопки распознаются как источники событий действия в их обработчике.) Далее для каждой кнопки задаются всплывающие подсказки. И наконец, создается панель инструментов, содержащая указанные кнопки для выбора режимов отладки.

Далее введите следующий фрагмент кода, в котором определяется обработчик событий действия от каждой кнопки на панели инструментов:

```
// создать обработчик событий действия от каждой кнопки
// на панели инструментов
EventHandler<ActionEvent> btnHandler =
    new EventHandler<ActionEvent>() {
        public void handle(ActionEvent ae) {
            response.setText(((Button)ae.getTarget()).getText());
        }
    };

// установить обработчики событий действия для отдельных кнопок
// на панели инструментов
```

```
btnSet.setOnAction(btnHandler);
btnClear.setOnAction(btnHandler);
btnResume.setOnAction(btnHandler);
```

И наконец, введите следующую строку кода, чтобы расположить панель инструментов в нижней области граничной компоновки:

```
rootNode.setBottom(tbDebug);
```

После внесения всех этих дополнений событие действия будет наступать всякий раз, когда пользователь щелкнет на кнопке панели инструментов. Обработка этого события состоит в отображении текстовой надписи (т.е. наименования) кнопки на месте метки `response`, как показано на рис. 36.8.

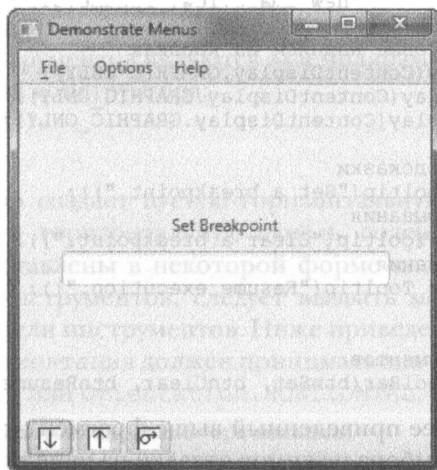


Рис. 36.8. Вид панели инструментов отладки с выбранной кнопкой в окне приложения MenuDemo

Составление окончательного варианта приложения MenuDemo

По ходу изложения материала этой главы в исходный вариант приложения MenuDemo было внесено немало изменений и дополнений. Прежде чем завершить эту главу, целесообразно составить окончательный вариант данного приложения с учетом всех изменений и дополнений. Это позволит не только избежать недоразумений относительно согласованности отдельных частей программы, но и получить рабочее приложение для демонстрации меню и дальнейшего экспериментирования с ними.

В приведенный ниже окончательный вариант приложения MenuDemo включены все изменения и дополнения, описанные ранее в этой главе. Ради большей ясности исходный код этого приложения был реорганизован таким образом, чтобы создавать разные меню и панель инструментов, используя отдельные методы. Следует также иметь в виду, что некоторые переменные, связанные с меню, в том

числе `mb` и `tbDebug`, были превращены в переменные экземпляра, чтобы сделать их доступными в любой части класса.

```
// Продемонстрировать меню - окончательный вариант

import javafx.application.*;
import javafx.scene.*;
import javafx.stage.*;
import javafx.scene.layout.*;
import javafx.scene.control.*;
import javafx.event.*;
import javafx.geometry.*;
import javafx.scene.input.*;
import javafx.scene.image.*;
import javafx.beans.value.*;

public class MenuDemoFinal extends Application {

    MenuBar mb;
    EventHandler<ActionEvent> MEHandler;
    ContextMenu editMenu;
    ToolBar tbDebug;

    Label response;

    public static void main(String[] args) {

        // запустить JavaFX-приложение, вызвав метод launch()
        launch(args);
    }

    // переопределить метод start()
    public void start(Stage myStage) {

        // присвоить заголовок подмосткам
        myStage.setTitle("Demonstrate Menus -- Final Version");
        // Демонстрация меню - окончательный вариант

        // использовать панель граничной компоновки BorderPane
        // в качестве корневого узла
        final BorderPane rootNode = new BorderPane();

        // создать сцену
        Scene myScene = new Scene(rootNode, 300, 300);

        // установить сцену на подмостках
        myStage.setScene(myScene);

        // создать метку для отображения результатов выбора
        // из разных элементов управления ГПИ приложения
        response = new Label();

        // создать один приемник действий для обработки всех
        // событий действия, наступающих в меню
        MEHandler = new EventHandler<ActionEvent>() {
            public void handle(ActionEvent ae) {
                String name = ((MenuItem)ae.getTarget()).getText();

                if(name.equals("Exit")) Platform.exit();

                response.setText( name + " selected");
            }
        };
    }
}
```

```

// создать строку меню
mb = new MenuBar();

// создать меню File
makeFileMenu();

// создать меню Options
makeOptionsMenu();

// создать меню Help
makeHelpMenu();

// создать контекстное меню
makeContextMenu();

// создать текстовое поле, задав ширину его столбца равной 20
TextField tf = new TextField();
tf.setPrefColumnCount(20);

// ввести контекстное меню в текстовое поле
tf.setContextMenu(editMenu);

// создать панель инструментов
makeToolBar();

// ввести контекстное меню непосредственно в граф сцены
rootNode.setOnContextMenuRequested(
    new EventHandler<ContextMenuEvent>() {
        public void handle(ContextMenuEvent ae) {
            // отобразить всплывающее контекстное меню на том месте,
            // где был произведен щелчок правой кнопкой мыши
            editMenu.show(rootNode, ae.getScreenX(), ae.getScreenY());
        }
    });

// ввести строку меню в верхней области панели
rootNode.setTop(mb);

// создать панель поточной компоновки для хранения
// текстового поля и метки ответной реакции на
// действия пользователя
FlowPane fpRoot = new FlowPane(10, 10);

// выровнять элементы управления по центру сцены
fpRoot.setAlignment(Pos.CENTER);

// использовать разделитель, чтобы улучшить порядок
// расположения элементов управления
Separator separator = new Separator();
separator.setPrefWidth(260);

// ввести метку, разделитель и текстовое поле на
// панели поточной компоновки
fpRoot.getChildren().addAll(response, separator, tf);

// ввести панель инструментов в нижней области панели
// граничной компоновки
rootNode.setBottom(tbDebug);

// ввести панель поточной компоновки в центральной области
// панели граничной компоновки
rootNode.setCenter(fpRoot);

```

```
// показать подмостки и сцену на них
myStage.show();
}

// создать меню File с мнемоникой
void makeFileMenu() {
    Menu fileMenu = new Menu("_File"); // Файл

    // создать отдельные пункты меню File
    MenuItem open = new MenuItem("Open"); // Открыть
    MenuItem close = new MenuItem("Close"); // Закрыть
    MenuItem save = new MenuItem("Save"); // Сохранить
    MenuItem exit = new MenuItem("Exit"); // Выход

    // ввести пункты в меню File
    fileMenu.getItems().addAll(
        open, close, save, new SeparatorMenuItem(), exit);

    // ввести оперативные клавиши для быстрого выбора
    // пунктов из меню File
    open.setAccelerator(KeyCombination.keyCombination("shortcut+O"));
    close.setAccelerator(KeyCombination.keyCombination("shortcut+C"));
    save.setAccelerator(KeyCombination.keyCombination("shortcut+S"));
    exit.setAccelerator(KeyCombination.keyCombination("shortcut+E"));

    // установить обработчики событий действия для пунктов меню File
    open.setOnAction(MEHandler);
    close.setOnAction(MEHandler);
    save.setOnAction(MEHandler);
    exit.setOnAction(MEHandler);

    // ввести меню File в строку меню
    mb.getMenus().add(fileMenu);
}

// создать меню Options
void makeOptionsMenu() {
    Menu optionsMenu = new Menu("Options"); // Параметры

    // создать подменю Colors
    Menu colorsMenu = new Menu("Colors"); // Цвета

    // использовать отмечаемые флажками пункты меню, чтобы
    // пользователь мог выбрать сразу несколько цветов
    CheckMenuItem red = new CheckMenuItem("Red"); // Красный
    CheckMenuItem green = new CheckMenuItem("Green"); // Зеленый
    CheckMenuItem blue = new CheckMenuItem("Blue"); // Синий

    // ввести отмечаемые флажками пункты в подменю Colors, а
    // само подменю Colors – в меню Options
    colorsMenu.getItems().addAll(red, green, blue);
    optionsMenu.getItems().add(colorsMenu);

    // задать зеленый цвет в качестве исходно выбираемого
    green.setSelected(true);

    // создать подменю Priority
    Menu priorityMenu = new Menu("Priority"); // Приоритет

    // использовать отмечаемые кнопками-переключателями пункты
    // меню для установки приоритета. Благодаря этому в меню не
    // только отображается установленный приоритет, но и
```



```

// гарантируется установка одного и только одного приоритета
RadioMenuItem high = new RadioMenuItem("High");
RadioMenuItem low = new RadioMenuItem("Low");

// создать группу кнопок-переключателей в пунктах подменю Priority
ToggleGroup tg = new ToggleGroup();
high.setToggleGroup(tg);
low.setToggleGroup(tg);

// отметить пункт меню High как исходно выбираемый
high.setSelected(true);

// ввести отмечаемые кнопками-переключателями пункты
// в подменю Priority, а последнее – в меню Options
priorityMenu.getItems().addAll(high, low);
optionsMenu.getItems().add(priorityMenu);

// ввести разделитель
optionsMenu.getItems().add(new SeparatorMenuItem());

// создать пункт меню Reset и ввести его в меню Options
MenuItem reset = new MenuItem("Reset"); // Сбросить
optionsMenu.getItems().add(reset);

// установить обработчики событий действия для
// пунктов меню Options
red.setOnAction(MEHandler);
green.setOnAction(MEHandler);
blue.setOnAction(MEHandler);
high.setOnAction(MEHandler);
low.setOnAction(MEHandler);
reset.setOnAction(MEHandler);

// использовать приемник событий изменения, чтобы оперативно
// реагировать на изменения в отметке пунктов подменю Priority
// кнопками-переключателями
tg.selectedToggleProperty().addListener(new ChangeListener<Toggle>() {
    public void changed(ObservableValue<? extends Toggle> changed,
                       Toggle oldVal, Toggle newVal) {
        if(newVal==null) return;

        // привести значение newVal к типу RadioButton
        RadioMenuItem rmi = (RadioMenuItem) newVal;

        // отобразить результат выбора приоритета
        response.setText("Priority selected is " + rmi.getText());
        // Выбран указанный приоритет
    }
});

// ввести меню Options в строку меню
mb.getMenus().add(optionsMenu);
}

// создать меню Help
void makeHelpMenu() {

    // создать представление типа ImageView для изображения
    ImageView aboutIV = new ImageView("aboutIcon.gif");

    // создать меню Help
    Menu helpMenu = new Menu("Help"); // Справка

```

```
// создать пункт About и ввести его в меню Help
MenuItem about = new MenuItem("About", aboutIV); // О программе
helpMenu.getItems().add(about);

// установить обработчик событий действия для
// пункта About меню Help
about.setOnAction(MEHandler);

// ввести меню Help в строку меню
mb.getMenus().add(helpMenu);
}

// создать пункты контекстного меню
void makeContextMenu() {

    // создать пункты для выбора команд редактирования
    // из контекстного меню
    MenuItem cut = new MenuItem("Cut"); // Вырезать
    MenuItem copy = new MenuItem("Copy"); // Копировать
    MenuItem paste = new MenuItem("Paste"); // Вставить

    // создать контекстное (т.е. всплывающее) меню
    // с пунктами для выбора команд редактирования
    editMenu = new ContextMenu(cut, copy, paste);

    // установить обработчики событий действия
    // для пунктов контекстного меню
    cut.setOnAction(MEHandler);
    copy.setOnAction(MEHandler);
    paste.setOnAction(MEHandler);
}

// создать панель инструментов
void makeToolBar() {
    // создать кнопки для панели инструментов
    Button btnSet = new Button(
        "Set Breakpoint", new ImageView("setBP.gif"));
    Button btnClear = new Button(
        "Clear Breakpoint", new ImageView("clearBP.gif"));
    Button btnResume = new Button(
        "Resume Execution", new ImageView("resume.gif"));

    // отключить текстовые надписи на кнопках
    btnSet.setContentDisplay(ContentDisplay.GRAPHIC_ONLY);
    btnClear.setContentDisplay(ContentDisplay.GRAPHIC_ONLY);
    btnResume.setContentDisplay(ContentDisplay.GRAPHIC_ONLY);

    // задать всплывающие подсказки для кнопок
    btnSet.setTooltip(new Tooltip("Set a breakpoint."));
    // Установить точку прерывания
    btnClear.setTooltip(new Tooltip("Clear a breakpoint."));
    // Очистить точку прерывания
    btnResume.setTooltip(new Tooltip("Resume execution."));
    // Возобновить выполнение

    // создать панель инструментов
    tbDebug = new ToolBar(btnSet, btnClear, btnResume);

    // создать обработчик событий от кнопок на панели инструментов
    EventHandler<ActionEvent> btnHandler =
        new EventHandler<ActionEvent>() {
            public void handle(ActionEvent ae) {
                response.setText(((Button)ae.getTarget()).getText());
            }
        }
}
```

```

    }
};

// установить обработчики событий действия для кнопок
// на панели инструментов
btnSet.setOnAction(btnHandler);
btnClear.setOnAction(btnHandler);
btnResume.setOnAction(btnHandler);
}
}

```

Дальнейшее изучение JavaFX

Каркас JavaFX предоставляет самые совершенные средства для построения ГПИ приложений в Java. В нем также переопределяются средства платформы Java. В трех главах части IV были представлены лишь некоторые из основных средств JavaFX, а остальные средства вам придется изучить самостоятельно. К их числу относится ряд дополнительных элементов управления, включая ползунки, автономные полосы прокрутки и таблицы. Можете также поэкспериментировать с доступными в JavaFX классами компоновок, например VBox и HBox, а также с различными эффектами из пакета `javafx.scene.effect` и преобразованиями из пакета `javafx.scene.transform`. Не менее интересными возможностями обладает класс `WebView`, позволяющий без особого труда внедрить веб-содержимое в граф сцены. Откровенно говоря, весь арсенал средств JavaFX заслуживает серьезного изучения. И во многих отношениях каркас JavaFX намечает дальнейший ход развития Java.

ЧАСТЬ

V

Применение Java

ГЛАВА 37

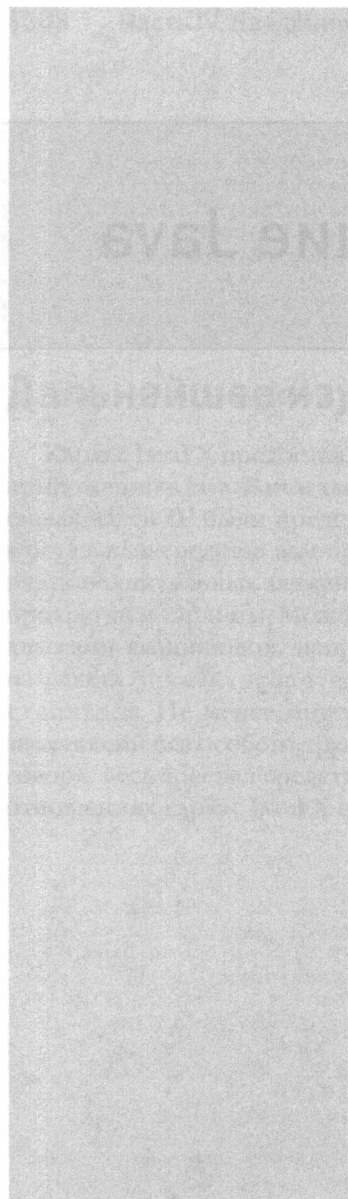
Компоненты Java Beans

ГЛАВА 38

Введение в сервлеты

ПРИЛОЖЕНИЕ

Применение
документирующих
комментариев в Java



Компоненты Java Beans

В этой главе представлен краткий обзор компонентов Java Beans. Особое значение этих программных компонентов состоит в том, что на их основе можно создавать сложные системы. Эти компоненты можно создавать самостоятельно, а также приобретать в готовом виде у сторонних производителей. Компоненты Java Beans определяют архитектуру, устанавливающую порядок взаимодействия стандартных блоков.

Чтобы стало понятнее особое значение компонентов Java Beans, обратимся к аналогии. В распоряжении разработчиков оборудования вычислительных систем имеются разнообразные компоненты, из которых можно построить вычислительную систему. Резисторы, конденсаторы и катушки индуктивности служат примерами простых стандартных блоков. Интегральные схемы предлагают еще больше функциональных возможностей. При этом каждый из этих компонентов можно использовать многократно. Их не нужно компоновать заново всякий раз, когда требуется создать новую систему. Кроме того, одни и те же компоненты можно использовать в разнотипных схемах. И все это возможно потому, что поведение подобных компонентов заранее известно и хорошо документировано.

И в отрасли разработки программного обеспечения предпринимались попытки воспользоваться преимуществами многократного использования компонентов и их способностью к взаимодействию. Для этого необходимо было разработать такую архитектуру компонентов, которая позволила бы составлять программы из стандартных блоков, в том числе и предлагаемых сторонними производителями. Кроме того, разработчик должен был иметь возможность выбрать компонент, разобраться в его функциях и внедрить его в приложение. А после выхода новой версии компонента нужно было обеспечить простоту внедрения его функциональных возможностей в уже существующий прикладной код. Именно такую архитектуру и предлагают компоненты Java Beans.

Общее представление о компонентах Java Beans

Java Beans — это компоненты программного обеспечения, предназначенные для многократного применения в самых разных средах. На функциональные возможности компонентов Java Beans не накладывается никаких ограничений. Они могут выполнять простую функцию, например получать стоимость товарно-материальных запасов, а могут реализовать и более сложную функцию, в частности

прогнозировать котировки акций на фондовой бирже. Компоненты Java Beans могут быть доступны для конечного пользователя. Одним из примеров таких компонентов служит экранная кнопка в ГПИ. В то же время компоненты Java Beans могут быть недоступны для пользователя. Примером таких компонентов служит программное обеспечение для декодирования потока мультимедийной информации в реальном времени. И наконец, компоненты Java Beans могут быть предназначены для работы в автономном режиме на рабочей станции пользователя или вместе с другими распределенными компонентами. Примером компонента, способного работать автономно, служит программное обеспечение для построения круговой диаграммы по точкам исходных данных. А компонент, предоставляющий сведения о котировках акций на фондовой бирже или ценах на товарной бирже в реальном времени, может работать только вместе с другим распределенным программным обеспечением, получая от него необходимые данные.

Преимущества компонентов Java Beans

Ниже перечислены некоторые преимущества, которые дает применение технологии Java Beans разработчику компонентов.

- Компоненты Java Beans получают все преимущества от реализуемого в Java принципа “написано однажды, работает везде”.
- Свойствами, событиями и методами компонентов Java Beans, доступными в другом приложении, можно управлять.
- Для настройки компонентов Java Beans можно применять вспомогательное программное обеспечение. Оно требуется только для установки параметров на стадии разработки компонента. А включать его в исполняющую среду не нужно.
- Состояние компонентов Java Beans можно хранить в постоянном хранилище и восстанавливать по мере надобности.
- Компоненты Java Beans можно регистрировать на получение событий от других объектов. Они могут и сами генерировать события, отправляя их другим объектам.

Самоанализ

В основу компонентов Java Beans положено *самоанализ* — процесс анализа компонентов Java Beans, в ходе которого определяются их характеристики. На самом деле это очень важная особенность прикладного программного интерфейса Java Beans API, поскольку с ее помощью другое приложение, например инструментальное средство разработки, может получить сведения о конкретном компоненте. Без самоанализа технология Java Beans неработоспособна.

Имеются два способа, с помощью которых разработчик компонентов Java Beans может указать, какие именно свойства, события и методы компонента должны быть доступны. Первый способ состоит в том, чтобы использовать простые

условные обозначения. Они позволяют механизмам самоанализа логически вывести сведения о компонентах Java Beans. Второй способ подразумевает предоставление дополнительного класса, расширяющего интерфейс BeanInfo. Этот класс, в свою очередь, предоставляет нужные сведения о компоненте явным образом. Оба способа рассматриваются в последующих разделах.

Шаблоны проектирования для свойств компонентов Java Beans

Свойства компонентов Java Beans являются подмножеством их состояния. Значения, присваиваемые свойствам, определяют поведение и внешний вид компонентов Java Beans. Значение свойства устанавливается *методом записи*, а получается оно *методом получения*. Имеются две разновидности свойств: простые и индексированные.

Простые свойства

У простого свойства имеется единственное значение. Его можно распознать по приведенным ниже шаблонам проектирования, где *N* — имя свойства; *T* — его тип.

```
public T getN()
public void setN(T аргумент);
```

Оба указанных метода имеются у каждого свойства, допускающего чтение и запись, для доступа к его значению. У свойства, допускающего только чтение, имеется лишь метод получения, а у свойства, допускающего только запись, — лишь метод установки.

В приведенном ниже примере демонстрируются три простых свойства, допускающих чтение и запись, а также их методы получения и установки.

```
private double depth, height, width;

public double getDepth() {
    return depth;
}

public void setDepth(double d) {
    depth = d;
}

public double getHeight() {
    return height;
}

public void setHeight(double h) {
    height = h;
}

public double getWidth() {
    return width;
}

public void setWidth(double w) {
    width = w;
}
```

На заметку! Для доступа к свойству типа boolean может быть также использован метод типа `isPropertyName()`.

Индексированные свойства

Индексированное свойство состоит из нескольких значений. Его можно распознать по приведенным ниже шаблонам проектирования, где *N* — имя свойства; *T* — его тип.

```
public T getN(int индекс);
public void setN(int индекс, T значение);
public T[] getN();
public void setN(T значения[]);
```

В приведенном ниже примере демонстрируется применение индексированного свойства *data*, а также его методов получения и установки.

```
private double data[];
public double getData(int index) {
    return data[index];
}

public void setData(int index, double value) {
    data[index] = value;
}

re public double[] getData() {
    turn data;
}

public void setData(double[] values) {
    data = new double[values.length];
    System.arraycopy(values, 0, data, 0, values.length);
}
```

Шаблоны проектирования для событий

В компонентах Java Beans применяется модель делегирования событий, рассматривавшаяся ранее в этой книге. Они способны генерировать события и посылать их другим объектам. События можно распознать по приведенным ниже шаблонам проектирования, где *T* обозначает тип события.

```
public void addTListener(TListener приемник_событий)
public void addTListener(TListener приемник_событий)
    throws java.util.TooManyListenersException
public void removeTListener(TListener приемник_событий)
```

Перечисленные выше методы служат для ввода или удаления слушателя указанного события. Вариант метода *AddTListener()*, не генерирующий исключение, можно использовать для *групповой рассылки* событий. Это означает, что на получение извещений о наступивших событиях может быть зарегистрировано несколько приемников. А вариант метода *AddTListener()*, генерирующий исключение типа *TooManyListenersException*, предназначен для *целевой рассылки* событий. Это означает, что извещение о наступивших событиях может получить только один приемник. Но в любом случае метод *removeTListener()* служит для удаления приемника событий. Так, если имеется интерфейс события типа *TemperatureListener*, то компонент Java Bean, следящий за температурой, может предоставить следующие методы:

```

public void addTemperatureListener(TemperatureListener tl) {
    ...
}
public void removeTemperatureListener(TemperatureListener tl) {
    ...
}

```

Методы и шаблоны проектирования

Шаблоны проектирования не используются для именования методов, не связанных со свойствами. Механизм самоанализа находит все открытые методы компонента Java Bean. А защищенные и закрытые методы остаются недоступными.

Применение интерфейса BeanInfo

Как упоминалось выше, шаблоны проектирования *неявно* определяют сведения, доступные пользователю компонента Java Bean. А интерфейс BeanInfo позволяет *явно* управлять доступом к этим сведениям. В интерфейсе BeanInfo определяется несколько методов, включая следующие:

```

PropertyDescriptor[] getPropertyDescriptors()
EventSetDescriptor[] getEventSetDescriptors()
MethodDescriptor[] getMethodDescriptors()

```

Эти методы возвращают массивы объектов, содержащие сведения о свойствах, событиях и методах компонентов Java Beans. Классы PropertyDescriptor, EventSetDescriptor и MethodDescriptor определяются в пакете java.beans и описывают указанные элементы. Реализуя эти методы, разработчик может точно определить, что именно доступно пользователю, не прибегая к самоанализу, выполняемому на основе шаблонов проектирования.

Если создается класс, реализующий интерфейс BeanInfo, он должен называться по форме *имяBeanInfo*, где *имя* обозначает имя компонента Java Bean. Так, если компонент Java Bean называется MyBean, то информационный класс должен называться MyBeanBeanInfo.

Чтобы упростить применение интерфейса BeanInfo, в компонентах Java Beans предоставляется класс SimpleBeanInfo. Он обеспечивает стандартные реализации интерфейса BeanInfo, включая перечисленные выше три метода. Этот класс можно расширить и переопределить в нем один или больше методов, чтобы явно управлять доступными свойствами компонентов Java Beans. Если не переопределить такой метод, то будет выполнен самоанализ из шаблона проектирования. А если не переопределить метод getPropertyDescriptors(), то для определения свойств компонентов Java Beans будут применяться шаблоны проектирования. Применение класса SimpleBeanInfo на практике будет продемонстрировано далее в этой главе.

Привязанные и ограниченные свойства

Компоненты Java Beans, имеющие *привязанное* свойство, генерируют события при изменении свойства. Наступающее событие относится к типу PropertyChangeEvent и посылается тем объектам, которые были предварительно зарегистри-

стрированы на получение подобных уведомлений. Класс, выполняющий обработку событий данного типа, должен реализовать интерфейс `PropertyChangeListener`.

Компоненты Java Beans, имеющие *ограниченное* свойство, генерируют события при попытке изменить значение этого свойства. Они также передают извещение о событии, имеющее тип `PropertyChangeEvent`. Это извещение о событии посылается другим объектам, предварительно зарегистрировавшимся на получение таких уведомлений. Но эти объекты могут наложить запрет на предлагаемое изменение, сгенерировав исключение типа `PropertyVetoException`. Это позволяет компонентам Java Beans действовать по-разному в зависимости от конкретной исполняющей среды. Класс, выполняющий обработку события данного типа, должен реализовать интерфейс `VetoableChangeListener`.

Сохраняемость компонентов Java Beans

Сохраняемость — это способность сохранять текущее состояние компонентов Java Beans, в том числе значения их свойств и переменные экземпляра, на энерго-независимом запоминающем устройстве и извлекать его по мере необходимости. Для сохраняемости компонентов Java Beans используются средства сериализации, предоставляемые библиотеками классов Java.

Сериализовать компоненты Java Beans проще всего, реализовав в них интерфейс `java.io.Serializable`, который является просто маркерным интерфейсом. Реализация интерфейса `java.io.Serializable` обеспечит автоматическое выполнение сериализации, больше ничего не требуя от компонентов Java Beans. Автоматическую сериализацию можно также наследовать. Иными словами, если какой-нибудь суперкласс компонентов Java Beans реализует интерфейс `java.io.Serializable`, он приобретет возможность для автоматической сериализации.

При автоматической сериализации можно выборочно отключить сохранение отдельного поля с помощью ключевого слова `transient`. Таким образом, элементы данных компонентов Java Beans, определенные как `transient`, не будут сериализованы.

Если же компоненты Java Beans не реализуют интерфейс `java.io.Serializable`, то возможность сериализации придется обеспечить самостоятельно (например, с помощью интерфейса `java.io.Externalizable`). В противном случае контейнеры не смогут сохранить конфигурацию компонентов Java Beans.

Настройщики

Разработчик компонентов Java Beans может предусмотреть возможность использования *настройщика*, с помощью которого другой разработчик сможет настроить эти компоненты Java Beans. Настройщик может обеспечивать пошаговое руководство всем процессом. Выполняя его указания, можно добиться применения компонента в определенном контексте, а также предоставить оперативно доступную документацию. Разработчику компонентов Java Beans предоставляется достаточно возможностей, чтобы разработать такой настройщик, который способен представить его программный продукт в выгодном свете на рынке.

Прикладной программный интерфейс Java Beans API

Функциональные возможности компонентов Java Beans обеспечиваются классами и интерфейсами из пакета `java.beans`. В этом разделе делается краткий обзор содержимого данного пакета. В табл. 37.1 перечислены интерфейсы из пакета `java.beans` с кратким описанием их функциональных возможностей, а в табл. 37.2 — классы из этого же пакета.

Таблица 37.1. Интерфейсы из пакета `java.beans`

Интерфейс	Описание
AppletInitializer	Методы этого интерфейса служат для инициализации компонентов Java Beans, которые являются также апплетами
BeanInfo	Позволяет разработчику указать сведения о свойствах, событиях и методах компонентов Java Beans
Customizer	Позволяет разработчику предоставить ГПИ для настройки компонентов Java Beans
DesignMode	Методы этого интерфейса определяют, выполняются ли компоненты Java Beans в режиме проектирования
ExceptionListener	Метод этого интерфейса вызывается, когда возникает исключение
PropertyChangeListener	Метод этого интерфейса вызывается при изменении привязанного свойства
PropertyEditor	Объекты классов, реализующих этот интерфейс, позволяют разработчикам изменять и отображать значения свойств компонентов Java Beans
VetoableChangeListener	Метод этого интерфейса вызывается при изменении ограниченного свойства
Visibility	Методы этого интерфейса позволяют компонентам Java Beans выполняться в тех средах, где отсутствует ГПИ

Таблица 37.2. Классы из пакета `java.beans`

Класс	Описание
BeanDescriptor	Предоставляет сведения о компонентах Java Beans. Позволяет также связывать настройщики с компонентами Java Beans
Beans	Служит для получения сведений о компонентах Java Beans
DefaultPersistenceDelegate	Служит подклассом, производным от класса PersistenceDelegate
Encoder	Зашифровывает состояние совокупности компонентов Java Beans. Может использоваться для записи этих данных в поток вывода
EventHandler	Поддерживает динамическое создание приемника событий
EventSetDescriptor	Экземпляры этого класса описывают событие, которое может генерироваться компонентами Java Beans

Класс	Описание
Expression	Инкапсулирует вызов метода, возвращающего результат
PropertyDescriptor	Служит суперклассом для классов PropertyDescriptor , EventSetDescriptor и MethodDescriptor
IndexedPropertyChangeEvent	Служит подклассом, производным от класса PropertyChangeEvent , представляя изменения в индексированном свойстве
IndexedPropertyDescriptor	Экземпляры этого класса описывают индексированное свойство компонентов Java Beans
IntrospectionException	Составляет выражение, если при анализе компонентов Java Beans возникает ошибка
Introspector	Анализирует компонент Java Bean и создает объект типа BeanInfo , описывающий этот компонент
MethodDescriptor	Экземпляры этого класса описывают методы компонентов Java Beans
ParameterDescriptor	Экземпляры этого класса описывают параметр метода
PersistenceDelegate	Обрабатывает данные состояния объекта
PropertyChangeEvent	Это событие генерируется при изменении привязанных или ограниченных свойств. Оно посылается тем объектам, которые зарегистрированы на получение событий данного типа, а их классы реализуют один из интерфейсов PropertyChangeListener или VetoableChangeListener
PropertyChangeListenerProxy	Расширяет класс EventListenerProxy и реализует интерфейс PropertyChangeListener
PropertyChangeSupport	Компоненты Java Beans, поддерживающие ограниченные свойства, могут использовать этот класс для уведомления объектов типа PropertyChangeListener
PropertyDescriptor	Экземпляры этого класса описывают свойства компонентов Java Beans
PropertyEditorManager	Обнаруживает редактор свойств типа PropertyEditor для заданного типа
PropertyEditorSupport	Предоставляет функциональные возможности для написания редакторов свойств
PropertyVetoException	Исключение данного типа генерируется, если изменять ограниченное свойство запрещено
SimpleBeanInfo	Предоставляет функциональные возможности для написания классов, расширяющих интерфейс BeanInfo
Statement	Инкапсулирует вызов метода
VetoableChangeListenerProxy	Расширяет класс EventListenerProxy и реализует интерфейс VetoableChangeListener
VetoableChangeSupport	Компоненты Java Beans, поддерживающие ограниченные свойства, могут использовать этот класс для уведомления объектов типа VetoableChangeListener
XMLDecoder	Служит для чтения компонентов Java Beans из XML-документа
XMLEncoder	Служит для записи компонентов Java Beans в XML-документ

В рамках одной главы невозможно описать все перечисленные выше классы. Тем не менее в последующих разделах будут вкратце рассмотрены следующие четыре класса, представляющие особый интерес: `Introspector`, `PropertyDescriptor`, `EventSetDescriptor` и `MethodDescriptor`.

Класс `Introspector`

В этом классе предоставляется несколько статических методов, поддерживающих самоанализ. Наиболее интересным из них является метод `getBeanInfo()`. Он возвращает объект типа `BeanInfo`, который служит для получения сведений о компонентах Java Beans.

У метода `getBeanInfo()` имеется несколько общих форм, включая приведенную ниже. Возвращаемый объект содержит сведения о компоненте Java Bean, обозначаемом параметром `bean`.

```
static BeanInfo getBeanInfo(Class<?> bean) throws IntrospectionException
```

Класс `PropertyDescriptor`

Класс `PropertyDescriptor` описывает свойство компонента Java Bean и предоставляет ряд методов для управления свойствами и их описания. Например, вызвав метод `isBound()`, можно выяснить, является ли свойство привязанным, а вызвав метод `isConstrained()` — является ли оно ограниченным. Имя свойства можно получить, вызвав метод `getName()`.

Класс `EventSetDescriptor`

Класс `EventSetDescriptor` представляет событие в компоненте Java Bean. Он предоставляет ряд методов для доступа к методам, предназначенным для ввода или удаления приемников событий или управления иным способом событиями в компонентах Java Beans. Например, чтобы получить метод, предназначенный для ввода приемников событий, следует вызвать метод `getAddListenerMethod()`; чтобы получить метод, предназначенный для удаления приемников событий, — метод `getRemoveListenerMethod()`; а для того чтобы получить тип приемника событий — метод `getListenerType()`. Имя события можно получить, вызвав метод `getName()`.

Класс `MethodDescriptor`

Класс `MethodDescriptor` представляет метод компонента Java Bean. Чтобы получить имя метода, следует вызвать метод `getName()`. А сведения о методе можно получить, вызвав метод `getMethod()`, общая форма которого показана ниже. Этот метод возвращает объект типа `Method`, описывающий искомый метод.

```
Method getMethod()
```

Пример компонента Java Bean

В завершение этой главы рассмотрим пример, демонстрирующий различные аспекты программирования компонентов Java Beans, включая самоанализ и применение интерфейса BeanInfo, а также классов Introspector, PropertyDescriptor и EventSetDescriptor. Данный пример состоит из трех классов. Первый класс является компонентом Java Bean и называется Colors. Ниже показано, каким образом определяется этот класс.

```

/// Простой компонент Java Bean
import java.awt.*;
import java.awt.event.*;
import java.io.Serializable;

public class Colors extends Canvas implements Serializable {
    transient private Color color; // не сохраняемая переменная
    private boolean rectangular;   // сохраняемая переменная

    public Colors() {
        addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent me) {
                change();
            }
        });
        rectangular = false;
        setSize(200, 100);
        change();
    }

    public boolean getRectangular() {
        return rectangular;
    }

    public void setRectangular(boolean flag) {
        this.rectangular = flag;
        repaint();
    }

    public void change() {
        color = randomColor();
        repaint();
    }

    private Color randomColor() {
        int r = (int)(255*Math.random());
        int g = (int)(255*Math.random());
        int b = (int)(255*Math.random());
        return new Color(r, g, b);
    }

    public void paint(Graphics g) {
        Dimension d = getSize();
        int h = d.height;
        int w = d.width;
        g.setColor(color);
        if(rectangular) {
            g.fillRect(0, 0, w-1, h-1);
        }
        else {
            g.fillOval(0, 0, w-1, h-1);
        }
    }
}

```

Компонент `Colors` отображает окрашенный в определенный цвет объект в прямоугольной рамке. Цвет компонента определяется закрытой переменной `color` типа `Color`, а его форма — закрытой переменной `rectangular` типа `boolean`. В конструкторе класса `Colors` определяется анонимный внутренний класс, расширяющий класс `MouseAdapter`, а также переопределяется метод `mousePressed()`. Метод `change()` вызывается в ответ на щелчок кнопкой мыши. Он выбирает произвольный цвет и окрашивает им компонент. Методы `getRectangular()` и `setRectangular()` предоставляют доступ к единственному свойству данного компонента Java Bean. Метод `change()` вызывает сначала метод `randomColor()`, чтобы выбрать произвольный цвет, а затем метод `repaint()`, чтобы сделать изменение цвета видимым. Обратите внимание на то, что с помощью переменных `rectangular` и `color` в методе `paint()` определяется, каким образом должен быть представлен компонент Java Bean.

Следующий класс `ColorsBeanInfo` является производным от класса `SimpleBeanInfo` и предоставляет подробные сведения о цвете. В этом классе переопределяется метод `getPropertyDescriptors()`, чтобы обозначить свойства, доступные пользователю компонента Java Bean. В данном случае пользователю доступно только свойство `rectangular`. Этот метод создает и возвращает объект типа `PropertyDescriptor` для свойства `rectangular`. Ниже приведен применяемый здесь конструктор класса `PropertyDescriptor`.

```
PropertyDescriptor(String свойство, Class<?> класс_bean)  
throws IntrospectionException
```

Параметры `свойство` и `класс_bean` обозначают имя свойства и класс компонента Java Bean соответственно. Ниже показано, каким образом определяется этот класс.

```
// Класс сведений о компоненте Java Bean
import java.beans.*;
public class ColorsBeanInfo extends SimpleBeanInfo {
    public PropertyDescriptor[] getPropertyDescriptors() {
        try {
            PropertyDescriptor rectangular = new
                PropertyDescriptor("rectangular", Colors.class);
            PropertyDescriptor pd[] = {rectangular};
            return pd;
        } catch (Exception e) {
            System.out.println("Перехвачено событие. " + e);
        }
        return null;
    }
}
```

И последним в рассматриваемом здесь примере компонента Java Bean является класс `IntrospectorDemo`. Он производит самоанализ для отображения свойств и событий, доступных в компоненте `Colors`. Ниже показано, каким образом определяется этот класс.

```
// Продемонстрировать свойства и события
import java.awt.*;
import java.beans.*;

public class IntrospectorDemo {
```



```

public static void main(String args[]) {
    try {
        Class<?> c = Class.forName("Colors");
        BeanInfo beanInfo = Introspector.getBeanInfo(c);

        System.out.println("Свойства:");
       PropertyDescriptor[] propertyDescriptor =
            beanInfo.getPropertyDescriptors();
        for(int i = 0; i < propertyDescriptor.length; i++) {
            System.out.println(
                "\t" + propertyDescriptor[i].getName());
        }

        System.out.println("События:");
        EventSetDescriptor[] eventSetDescriptor =
            beanInfo.getEventSetDescriptors();
        for(int i = 0; i < eventSetDescriptor.length; i++) {
            System.out.println(
                "\t" + eventSetDescriptor[i].getName());
        }
    } catch (Exception e) {
        System.out.println("Перехвачено событие. " + e);
    }
}

```

Ниже приведен результат выполнения программы из данного примера.

```

Свойства:
    rectangular
События:
    mouseWheel
    mouse
    mouseMotion
    component
    hierarchyBounds
    focus
    hierarchy
    propertyChange
    inputMethod
    key

```

В этом результате следует обратить внимание на следующее. Во-первых, в классе `ColorsBeanInfo` метод `getPropertyDescriptors()` переопределяется таким образом, чтобы возвращать единственное свойство `rectangular`, и поэтому выводится только это свойство. И во-вторых, метод `getEventSetDescriptors()` не переопределяется в классе `ColorsBeanInfo`, и поэтому производится самоанализ по шаблону проектирования и обнаруживаются все события, в том числе и те, которые относятся к классу `Colors` суперкласса `Canvas`. Напомним, что если не переопределить один из методов получения, определенных в классе `SimpleBeanInfo`, то по умолчанию производится самоанализ по шаблону проектирования. Чтобы понаблюдать за изменениями, которые вносит класс `ColorsBeanInfo`, следует удалить файл этого класса и еще раз запустить программу `IntrospectorDemo` на выполнение. На этот раз будет выведено больше свойств.

Введение в сервлеты

В этой главе речь пойдет о сервлетах. *Сервлетами* называют небольшие программы, которые выполняются на стороне сервера веб-подключения. Апплеты динамически расширяют функциональные возможности веб-браузера, а сервлеты — возможности веб-сервера. Тема сервлетов довольно обширна, и ее невозможно рассмотреть полностью в рамках одной главы. Поэтому в этой главе будут представлены основные принципы, интерфейсы и классы, а также некоторые примеры создания сервлетов.

Предпосылки для разработки сервлетов

Чтобы стали понятнее преимущества сервлетов, следует дать хотя бы общее представление о взаимодействии веб-браузеров и сервлетов для предоставления содержимого пользователю. Рассмотрим обработку запроса статической веб-страницы. Пользователь вводит в поле адреса, обычно находящегося в верхней части окна браузера, URL (Uniform Resource Locator — унифицированный указатель информационного ресурса). Браузер формирует запрос по сетевому протоколу HTTP, направляя его соответствующему веб-серверу. А веб-сервер сопоставляет этот запрос с конкретным файлом, возвращая его браузеру в виде ответа, посылаемого по сетевому протоколу HTTP. В этом ответе HTTP-заголовок обозначает тип содержимого. Для этой цели служит стандарт MIME (Multipurpose Internet Mail Extensions — многоцелевые расширения электронной почты). Например, обычный текст в формате ASCII имеет тип MIME, обозначаемый как `text/plain`. Исходный код HTML (Hypertext Markup Language — язык разметки гипертекста) имеет тип MIME, обозначаемый как `text/html`.

А теперь рассмотрим динамическое содержимое. Допустим, база данных применяется в интернет-магазине для хранения сведений о его коммерческой деятельности. В базе данных могут храниться продаваемые товары, прейскуранты, сведения о наличии товара, заказах и т.п. Владелец этого интернет-магазина решил сделать эту информацию доступной покупателям через веб-страницы. Содержимое этих веб-страниц должно создаваться динамически, чтобы отражать самые последние сведения, хранящиеся в базе данных.

В первые годы существования веб-сервер мог динамически формировать страницу, создавая отдельный процесс для обработки каждого запроса клиента. Чтобы получить необходимую информацию, процесс мог устанавливать соедине-

ние с одной или несколькими базами данных. Связь с сервером осуществлялась через интерфейс CGI (Common Gateway Interface — общий шлюзовой интерфейс). Интерфейс CGI позволял отдельным процессам вводить данные из HTTP-запроса и выводить их в HTTP-ответ. Для написания CGI-программ применялись разные языки программирования, в том числе C, C++ и Perl.

Но интерфейс CGI страдал серьезными недостатками, связанными с производительностью. Он был неэффективным с точки зрения ресурсов ЦП и оперативной памяти, потребляемых для создания отдельного процесса, а также установления и разрыва соединений с базой данных по каждому запросу клиента. Кроме того, CGI-программы зависели от конкретной платформы. Потому для преодоления подобных недостатков были внедрены другие методики, к числу которых относятся сервлеты.

По сравнению с интерфейсом CGI сервлеты обладают рядом преимуществами. Во-первых, их производительность заметно выше. Сервлеты выполняются в адресном пространстве веб-сервера. Чтобы выполнить обработку каждого запроса клиента, не обязательно создавать отдельный процесс. Во-вторых, сервлеты не зависят от конкретной платформы, поскольку они разрабатываются на Java. В-третьих, диспетчер безопасности Java на сервере накладывает ряд ограничений для защиты ресурсов на серверной машине. И наконец, для сервлета доступны абсолютно все функциональные возможности библиотек классов Java. Сервлет мочерез механизмы сокетов и удаленног жет связываться с апплетами, базой данных и другим программным обеспечением о вызова методов (RMI), рассматривавшиеся ранее в данной книге.

Жизненный цикл сервлета

Жизненный цикл сервлета определяют следующие основные методы: `init()`, `service()` и `destroy()`. Они реализуются каждым сервлетом и в нужный момент вызываются сервером. Рассмотрим обычный сценарий взаимодействия с пользователем, который поможет лучше понять, когда именно происходит вызов этих методов.

Итак, допустим, что пользователь ввел URL в окне браузера. Исходя из этого URL, браузер формирует HTTP-запрос, посылаемый соответствующему серверу. Затем этот HTTP-запрос принимает веб-сервер и сопоставляет его с конкретным сервлетом. Обнаруженный сервлет динамически загружается в адресное пространство сервера.

Далее сервер вызывает метод `init()` из сервлета. Этот метод вызывается только в том случае, если сервлет впервые загружается в оперативную память. Сервлету можно передавать параметры инициализации, поэтому он допускает самонастройку.

После этого сервер вызывает метод `service()` из сервлета. Этот метод вызывается для обработки HTTP-запроса. Как будет показано далее, сервлет может считывать данные, содержащиеся в HTTP-запросе, а также сформировать HTTP-ответ клиенту. Сервлет остается в адресном пространстве сервера и доступен для обработки любых других HTTP-запросов, получаемых от клиентов. Метод `service()` вызывается по каждому HTTP-запросу.

И наконец, сервер может принять решение выгрузить сервлет из оперативной памяти. Для принятия такого решения на каждом сервере применяются разные алгоритмы. А для освобождения таких ресурсов, как дескрипторы файлов, выделяемых для сервлета, сервер вызывает метод `destroy()`. Важные данные могут быть сохра-

нены в постоянном хранилище. Оперативная память, выделяемая для сервлета и его объектов, может быть впоследствии утилизирована в процессе “сборки мусора”.

Варианты разработки сервлетов

Для разработки сервлетов потребуется доступ к контейнеру сервлетов или серверу приложений. Наиболее популярными из них являются сервер приложений Glassfish и контейнер сервлетов Tomcat. Сервер приложений Glassfish предоставляется компанией Oracle в комплекте Java EE SDK. Он поддерживается в ИСР NetBeans. А контейнер сервлетов Tomcat — это программный продукт с открытым исходным кодом, поддерживаемый организацией Apache Software Foundation. Он также поддерживается в ИСР NetBeans. Контейнером сервлетов Tomcat и сервером приложений Glassfish можно пользоваться и в других ИСР, например Eclipse. В примерах и пояснениях, приводимых в этой главе, применяется контейнер сервлетов Tomcat по причинам, которые вам скоро станут понятны.

Несмотря на то что такие ИСР, как NetBeans и Eclipse, способны значительно упростить разработку сервлетов, здесь и далее в этой главе они не применяются. В разных ИСР разработка и развертывание сервлетов имеет свои отличия, и поэтому здесь просто невозможно рассмотреть все эти отличия. Кроме того, многие читатели, скорее всего, будут пользоваться инструментальными средствами командной строки, а не ИСР. Поэтому, если вы пользуетесь ИСР, за справкой о разработке и развертывании сервлетов обращайтесь к документации на эту ИСР. А все приведенные далее пояснения подразумевают, что для разработки сервлетов применяются только инструментальные средства командной строки. Таким образом, они подойдут практически для любого читателя.

Контейнер сервлетов Tomcat применяется в этой главе потому, что выполнять примеры сервлетов, используя только инструментальные средства командной строки и текстовый редактор, по мнению автора, проще и доступнее в разных средах программирования. Кроме того, не придется загружать и устанавливать ИСР только для того, чтобы экспериментировать с сервлетами, поскольку для этого достаточно инструментальных средств командной строки. Следует, однако, иметь в виду, что рассматриваемые здесь основные принципы разработки сервлетов вполне пригодны и в той среде, где применяется сервер приложений Glassfish. Немного отличаться будет лишь механизм подготовки сервлета к тестированию.

Помните! Наставления по разработке и развертыванию сервлетов, представленные далее в этой главе, подразумевают использование только контейнера сервлетов Tomcat и инструментов командной строки. Если вы пользуетесь ИСР и другим контейнером сервлетов или сервером приложений, обратитесь к документации на свою среду.

Применение контейнера сервлетов Tomcat

В состав контейнера сервлетов Tomcat входят библиотеки классов, документация и исполняющая среда для разработки и проверки сервлетов. На момент напи-

сания данной книги для использования было доступно несколько версий контейнера сервлетов Tomcat, но в приведенном далее описании подразумевается применение версии 7.0.47. Контейнер сервлетов Tomcat можно загрузить по адресу `tomcat.apache.org`. Выберите ту версию, которая подходит для вашей среды.

В примерах этой главы подразумевается применение 64-разрядной версии операционной системы Windows. Кроме того, предполагается, что 64-разрядная версия контейнера Tomcat 7.0.47 была распакована из корневого каталога непосредственно в заданное по умолчанию место, как показано ниже.

```
C:\apache-tomcat-7.0.47-windows-x64\apache-tomcat-7.0.47\
```

Именно это место расположения контейнера сервлетов Tomcat подразумевается в рассматриваемых далее примерах. Если же загрузить контейнер сервлетов Tomcat в другое место (или выбрать другую его версию), то в эти примеры придется внести соответствующие изменения, а возможно, и установить переменную среды окружения `JAVA_HOME`, указав в ней каталог, где установлен комплект разработки Java Development Kit.

На заметку! Все каталоги, представленные в этом разделе, подразумевают применение версии 7.0.47 контейнера сервлетов Tomcat. Если установить другую версию этого контейнера, то придется откорректировать имена используемых каталогов и пути к ним, чтобы они соответствовали установленной версии.

После установки контейнера сервлетов Tomcat следует запустить его на выполнение, выбрав файл `startup.bat` в каталоге `\apache-tomcat-7.0.47\bin`. Чтобы остановить контейнер Tomcat, следует выполнить файл `shutdown.bat`, который также находится в каталоге `bin`.

Классы и интерфейсы, требующиеся для создания сервлетов, содержатся в архивном файле `servlet-api.jar`, который находится в следующем каталоге:

```
C:\apache-tomcat-7.0.47-windows-x64\apache-tomcat-7.0.47\lib
```

Для доступа к архивному файлу `servlet-api.jar` следует обновить переменную окружения `CLASSPATH` таким образом, чтобы она включала следующую строку:

```
C:\Program Files\Apache Software Foundation\Tomcat 5.5\common\lib\servlet-api.jar
```

С другой стороны, этот файл можно указать во время компиляции сервлетов. Так, пример сервлета из этой главы компилируется по следующей команде:

```
javac HelloServlet.java -classpath "C:\apache-tomcat-7.0.4-windowsx64\apache-tomcat-7.0.4\lib\servlet-api.jar"
```

По завершении компиляции сервлета нужно сделать так, чтобы контейнер сервлетов Tomcat нашел его. Для этого следует разместить сервлет в подкаталоге `\webapps\имя_каталога` установки контейнера Tomcat и ввести его имя в файл `web.xml`. Ради простоты в примерах этой главы применяется каталог и файл `web.xml`, который контейнер сервлетов Tomcat использует для своих образцов сервлетов. Это избавляет от необходимости создавать файлы или каталоги только для экспериментов с примерами сервлетов. Ниже поясняются действия, которые потребуются выполнить для этого.

Скопируйте файл класса сервлета в следующий каталог:

```
C:\apache-tomcat-7.0.47-windows-x64\apache-tomcat-7.0.47\webapps\
  examples\WEB-INF\classes
```

Введите имя сервлета и его сопоставление с файлом `web.xml` в каталог

```
C:\apache-tomcat-7.0.47-windows-x64\apache-tomcat-7.0.47\webapps\
  examples\WEB-INF
```

Так, если первый пример сервлета называется `HelloServlet`, то в раздел файла `web.xml`, где описываются сервлеты, нужно ввести следующие строки кода разметки:

```
<servlet>
  <servlet-name>HelloServlet</servlet-name>
  <servlet-class>HelloServlet</servlet-class>
</servlet>
```

Введите приведенные ниже строки кода разметки в тот раздел файла `web.xml`, где определяются сопоставления сервлета с шаблоном. Эти же действия следует выполнить и для всех остальных примеров сервлетов.

```
<servlet-mapping>
  <servlet-name>HelloServlet</servlet-name>
  <url-pattern>/servlet/HelloServlet</url-pattern>
</servlet-mapping>
```

Простой пример сервлета

Чтобы стали понятнее основные принципы разработки сервлетов, обратимся к простому примеру создания и проверки сервлета. Для этого выполните следующие основные действия.

1. Создайте и скомпилируйте исходный код сервлета. После этого скопируйте файл класса сервлета в соответствующий каталог и введите имя сервлета и его сопоставления в соответствующий файл `web.xml`.
2. Запустите на выполнение контейнер сервлетов `Tomcat`.
3. Запустите на выполнение веб-браузер и запросите сервлет.

А теперь рассмотрим подробнее каждое из этих основных действий.

Создание и компиляция исходного кода сервлета

Прежде всего создайте файл `HelloServlet.java`, который будет содержать исходный код следующей программы:

```
import java.io.*;
import javax.servlet.*;

public class HelloServlet extends GenericServlet {

    public void service(ServletRequest request,
                       ServletResponse response)
```

```

throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.println("<B>Hello!");
    pw.close();
}
}

```

Проанализируем исходный код этой программы. Прежде всего обратите внимание на то, что в ней импортируется пакет `javax.servlet`. Этот пакет содержит классы и интерфейсы, требующиеся для создания сервлетов. Мы еще вернемся к ним далее в этой главе. В данной программе определяется также класс `HelloServlet`, производный от класса `GenericServlet`. Класс `GenericServlet` обладает функциональными возможностями, упрощающими создание сервлетов. В частности, он предоставляет версии методов `init()` и `destroy()`, которые можно использовать в исходном виде.

Из всех методов, наследуемых из класса `GenericServlet`, в классе `HelloServlet` переопределяется только метод `service()`, который обрабатывает запросы от клиента. В качестве его первого аргумента указывается объект типа `ServletRequest`, чтобы сервлет вводил данные из запроса клиента, а в качестве второго аргумента — объект типа `ServletResponse`, чтобы сервлет формировал ответ клиенту.

При вызове метода `setContentType()` определяется тип MIME для HTTP-ответа. В данной программе употребляется тип MIME, обозначаемый как `text/html`. Он обозначает, что браузер должен интерпретировать содержимое как исходный код HTML-разметки.

Метод `getWriter()` получает объект типа `PrintWriter`. Все, что направляется в поток вывода, посылается клиенту как часть HTTP-ответа. Затем вызывается метод `println()` для вывода простого фрагмента кода HTML-разметки в качестве HTTP-ответа.

Скомпилируйте исходный код данной программы и разместите файл `HelloServlet.class` в соответствующем каталоге контейнера сервлетов Tomcat, как пояснялось в предыдущем разделе. Кроме того, введите класс `HelloServlet` в файл `web.xml` описанным ранее способом.

Запуск контейнера сервлетов Tomcat на выполнение

Запустите контейнер сервлетов Tomcat на выполнение, как пояснялось ранее. Контейнер сервлетов Tomcat должен действовать еще до выполнения сервлета.

Запуск веб-браузера и запрос сервлета

Запустите веб-браузер на выполнение и введите в его окне следующий URL:
<http://localhost:8080/examples/servlets/servlet/HelloServlet>

С другой стороны, можно ввести и такой URL:
<http://127.0.0.1:8080/examples/servlets/servlet/HelloServlet>

Это вполне допустимый URL, поскольку IP-адрес `127.0.0.1` определяется как адрес локальной машины.

В окне браузера должны появиться данные, выводимые сервлетом. Они будут состоять из символьной строки "Hello!", выделенной полужирным.

Прикладной программный интерфейс Servlet API

Рассматриваемые в этой главе классы и интерфейсы, требующиеся для разработки сервлетов, содержатся в двух пакетах, `javax.servlet` и `javax.servlet.http`, и образуют интерфейс Servlet API. Следует, однако, иметь в виду, что эти пакеты не относятся к основным пакетам Java. Следовательно, они не входят в состав версии Java SE. Вместо этого они предоставляются контейнером сервлетов Tomcat, а также доступны в версии Java EE.

Прикладной программный интерфейс Servlet API по-прежнему находится на стадии разработки и усовершенствования. Текущая спецификация сервлетов относится к версии 3.1, и именно она используется в примерах из этой главы. Но поскольку в Java все постоянно меняется, то поинтересуйтесь, не появились ли какие-нибудь дополнения или видоизменения данной спецификации. В этой главе обсуждается ядро прикладного программного интерфейса Servlet API, которое доступно большинству читателей и поддерживается во всех современных версиях спецификации сервлетов.

Пакет `javax.servlet`

Пакет `javax.servlet` содержит целый ряд интерфейсов и классов, образующих каркас для функционирования сервлетов. В табл. 38.1 перечислены основные интерфейсы, предоставляемые в данном пакете. Самым главным из них является интерфейс `Servlet`. Все сервлеты должны реализовывать этот интерфейс или расширять реализующий его класс. Не менее важны интерфейсы `ServletRequest` и `ServletResponse`.

Таблица 38.1. Основные интерфейсы из пакета `javax.servlet`

Интерфейс	Описание
<code>Servlet</code>	Объявляет методы жизненного цикла сервлета
<code>ServletConfig</code>	Позволяет получать параметры инициализации сервлетов
<code>ServletContext</code>	Позволяет регистрировать события в сервлетах и обращаться к сведениям об их среде
<code>ServletRequest</code>	Служит для ввода данных из запроса клиента
<code>ServletResponse</code>	Служит для вывода данных в ответ клиенту

В табл. 38.2 перечислены основные классы из пакета `javax.servlet`. Интерфейсы и классы, перечисленные в обеих таблицах, будут рассмотрены далее более подробно.

Таблица 38.2. Основные классы из пакета `javax.servlet`

Класс	Описание
<code>GenericServlet</code>	Реализует интерфейсы <code>Servlet</code> и <code>ServletConfig</code>
<code>ServletInputStream</code>	Предоставляет поток для ввода запросов клиента
<code>ServletOutputStream</code>	Предоставляет поток для вывода ответов клиенту
<code>ServletException</code>	Указывает на то, что в сервлете произошла ошибка
<code>UnavailableException</code>	Указывает на то, что сервлет недоступен

Интерфейс `Servlet`

Все сервлеты должны реализовать интерфейс `Servlet`. В нем объявляются методы `init()`, `service()` и `destroy()`, которые вызываются сервером в течение жизненного цикла сервлета. Кроме них предоставляется также метод, позволяющий сервлету получать любые параметры инициализации. Методы, определяемые в интерфейсе `Servlet`, перечислены в табл. 38.3.

Таблица 38.3. Методы из интерфейса `Servlet`

Метод	Описание
<code>void destroy()</code>	Вызывается при выгрузке сервлета
<code>ServletConfig getServletConfig()</code>	Возвращает объект типа <code>ServletConfig</code> , содержащий любые параметры инициализации
<code>String getServletInfo()</code>	Возвращает символьную строку, описывающую сервлет
<code>void init(ServletConfig sc) throws ServletException</code>	Вызывается во время инициализации сервлета. Параметры инициализации для сервлета могут быть получены из параметра <code>sc</code> . Если сервлет нельзя инициализировать, то генерируется исключение типа <code>ServletException</code>
<code>void service(ServletRequest запрос, ServletResponse ответ) throws ServletException, IOException</code>	Вызывается для обработки запроса клиента. Запрос клиента можно ввести из объекта, обозначаемого параметром <code>запрос</code> , а ответ клиенту — вывести в объект, обозначаемый параметром <code>ответ</code> . Если при выполнении сервлета или вводе-выводе возникают ошибки, то генерируется исключение

Методы `init()`, `service()` и `destroy()` определяют жизненный цикл сервлета. Они вызываются сервером. Метод `getServletConfig()` вызывается сервлетом для получения параметров инициализации. Разработчик сервлета переопределяет метод `getServletInfo()`, чтобы предоставить строку с полезной информацией (например, Ф.И.О. автора, номер версии, дата выпуска, авторские права). Этот метод также вызывается сервером.

Интерфейс `ServletConfig`

Интерфейс `ServletConfig` позволяет получать в сервлете данные конфигурации сервлета во время его загрузки. В табл. 38.4 перечислены методы, объявляемые в этом интерфейсе.

Таблица 38.4. Методы из интерфейса `ServletConfig`

Метод	Описание
<code>ServletContext getServletContext()</code>	Возвращает содержимое данного сервлета
<code>String getInitParameter(String параметр)</code>	Возвращает значение параметра инициализации
<code>Enumeration getInitParameterNames()</code>	Возвращает перечень имен параметров инициализации
<code>String getServletName()</code>	Возвращает имя вызывающего сервлета

Интерфейс `ServletContext`

Интерфейс `ServletContext` позволяет получать в сервлете сведения о среде исполнения сервлетов. Некоторые его методы перечислены в табл. 38.5.

Таблица 38.5. Методы из интерфейса `ServletContext`

Метод	Описание
<code>Object getAttribute(String атрибут)</code>	Возвращает значение указанного атрибута сервера
<code>String getMimeType(String файл)</code>	Возвращает тип MIME указанного файла
<code>String getRealPath(String виртуальный_путь)</code>	Возвращает реальный путь, соответствующий указанному виртуальному_пути
<code>String getServerInfo()</code>	Возвращает сведения о сервере
<code>void log(String s)</code>	Записывает указанную строку s в журнал сервлета
<code>void log(String s, Throwable e)</code>	Записывает указанную строку s и трассировку стека для заданного исключения e в журнал сервлета
<code>void setAttribute(String атрибут, Object значение)</code>	Присваивает заданному атрибуту указанное значение

Интерфейс `ServletRequest`

Интерфейс `ServletRequest` позволяет получать в сервлете сведения о запросе клиента. Некоторые его методы перечислены в табл. 38.6.

Таблица 38.6. Методы из интерфейса `ServletRequest`

Метод	Описание
<code>Object getAttribute(String атрибут)</code>	Возвращает значение указанного атрибута
<code>String getCharacterEncoding()</code>	Возвращает кодировку символов в запросе
<code>Int getContentLength()</code>	Возвращает длину запроса. Если длину запроса нельзя определить, возвращается значение -1
<code>String getContentType()</code>	Возвращает тип запроса. Если тип запроса нельзя определить, возвращается пустое значение null

Метод	Описание
ServletInputStream getInputStream() throws IOException	Возвращает поток ввода типа ServletInputStream , который можно использовать для чтения двоичных данных из запроса. Если метод getReader() уже вызывался для этого запроса, то генерируется исключение типа IllegalStateException
String getParameter(String имя_параметра)	Возвращает значение указанного параметра имя_параметра
Enumeration<String> getParameterNames()	Возвращает перечисление имен параметров по данному запросу
String[] getParameterValues(String имя)	Возвращает массив, состоящий из значений, связанных с параметром имя
String getProtocol()	Возвращает описание протокола
BufferedReader getReader() throws IOException	Возвращает буферизированный поток чтения, который можно использовать для ввода текста из запроса. Если метод getInputStream() уже вызывался по данному запросу, то генерируется исключение типа IllegalStateException
String getRemoteAddr()	Возвращает строковый эквивалент IP-адреса клиента
String getRemoteHost()	Возвращает строковый эквивалент имени хоста клиента
String getScheme()	Возвращает схему передачи URL, которая используется для запроса (например, "http", "ftp" и т.д.)
String getServerName()	Возвращает имя сервера
int getServerPort()	Возвращает номер порта

Интерфейс **ServletResponse**

Интерфейс **ServletResponse** позволяет сформировать в сервлете ответ клиенту. Некоторые его методы из этого интерфейса перечислены в табл. 38.7.

Таблица 38.7. Методы из интерфейса **ServletResponse**

Метод	Описание
String getCharacterEncoding()	Возвращает кодировку символов в ответе
ServletOutputStream getOutputStream() throws IOException	Возвращает поток вывода типа ServletOutputStream , который можно использовать для записи двоичных данных из сервлета в ответ. Если метод getWriter() уже вызывался по данному запросу, то генерируется исключение типа IllegalStateException
PrintWriter getWriter() throws IOException	Возвращает поток записи типа PrintWriter , который можно использовать для вывода символьных данных в ответ. Если метод getOutputStream() уже вызывался по данному запросу, то генерируется исключение типа IllegalStateException

Окончание табл. 38.7

Метод	Описание
<code>void setContentLength(int <i>длина</i>)</code>	Задаёт указанную <i>длину</i> содержимого ответа
<code>void.setContentType(String <i>тип</i>)</code>	Задаёт тип содержимого ответа

Класс GenericServlet

Класс `GenericServlet` предоставляет реализации основных методов жизненного цикла сервлета. Этот класс реализует интерфейсы `Servlet` и `ServletConfig`, а также предоставляет метод для записи символьной строки в журнал сервера. Ниже приведены общие формы этого метода, где параметр *s* обозначает записываемую в журнал символьную строку, а параметр *e* — исключение, генерируемое при возникновении ошибки.

```
void log(String s)
void log(String s, Throwable e)
```

Класс ServletInputStream

Класс `ServletInputStream` расширяет класс `InputStream`. Он реализуется контейнером сервлетов и предоставляет поток ввода, который разработчик сервлета может использовать для чтения данных из запроса клиента. В этом классе определяется конструктор по умолчанию, а также метод для чтения байтов из потока ввода. Ниже приведена общая форма этого метода.

```
int readLine(byte[] буфер, int смещение, int размер) throws IOException
```

Здесь параметр *буфер* обозначает массив, в котором хранится определенное количество (*размер*) байтов, начиная с указанного *смещения*. Этот метод возвращает фактическое количество прочитанных байтов или значение `-1`, если будет достигнуто условие окончания потока ввода.

Класс ServletOutputStream

Класс `ServletOutputStream` расширяет класс `OutputStream`. Он реализуется контейнером сервлетов и предоставляет поток вывода, который разработчик сервлета может применять для записи данных в ответ клиенту. В этом классе определяется конструктор по умолчанию, а также методы `print()` и `println()`, предназначенные для вывода данных в поток.

Класс ServletException

В пакете `javax.servlet` определяются два класса исключений. Первый класс представляет исключение типа `ServletException`, которое извещает об ошибке при выполнении сервлета, а второй класс — исключение типа `UnavailableException`, класс которого расширяет класс `ServletException`. Это исключение извещает о том, что сервлет недоступен.

Ввод параметров сервлета

В интерфейс `ServletRequest` входят методы, позволяющие вводить имена и значения параметров, включаемых в запрос клиента. В приведенном далее примере разрабатывается сервлет, демонстрирующий их применение. Этот пример состоит из двух файлов. Веб-страница определяется в файле `PostParameters.html`, а сам сервлет — в файле `PostParametersServlet.java`.

Ниже приведен исходный код из HTML-файла `PostParameters.html`. В этом коде определяется таблица, состоящая из двух меток и двух текстовых полей. Одна из меток называется **Employee** (Сотрудник), другая — **Phone** (Телефон). Имеется также кнопка подтверждения. Обратите внимание на то, что параметр `action` скриптора `<form>` определяет URL. Этот URL обозначает сервлет, который будет выполнять обработку HTTP-запроса типа POST.

```
<html>
<body>
<center>
<form name="Form1"
      method="post"
      action="http://localhost:8080/examples/servlets/
            servlet/PostParametersServlet">

<table>
<tr>
  <td><B>Employee</td>
  <td><input type="text" name="e" size="25" value=""></td>
</tr>
<tr>
  <td><B>Phone</td>
  <td><input type="text" name="p" size="25" value=""></td>
</tr>
</table>
<input type="submit" value="Submit">
</body>
</html>
```

Ниже приведен исходный код из файла `PostParametersServlet.java`. Метод `service()` переопределяется для обработки запросов клиента, а метод `getParameterNames()` возвращает перечисление имен параметров. Их обработка осуществляется в цикле. Как видите, для клиента выводится имя параметра и его значение. Значение параметра получается методом `getParameter()`.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;

public class PostParametersServlet
extends GenericServlet {

    public void service(ServletRequest request,
                       ServletResponse response)
    throws ServletException, IOException {

        // получить поток записи типа PrintWriter
        PrintWriter pw = response.getWriter();

        // получить перечисление имен параметров
```

```
Enumeration e = request.getParameterNames();

// вывести имена параметров и их значения
while(e.hasMoreElements()) {
    String pname = (String)e.nextElement();
    pw.print(pname + " = ");
    String pvalue = request.getParameter(pname);
    pw.println(pvalue);
}
pw.close();
}
```

Скомпилируйте сервлет. Затем скопируйте его в соответствующий каталог и обновите файл `web.xml`, как упоминалось ранее. После этого выполните следующие действия для проверки сервлета из данного примера.

1. Запустите на выполнение контейнер сервлетов Tomcat, если это еще не сделано.
2. Воспроизведите веб-страницу в окне браузера.
3. Введите в текстовых полях фамилию служащего и номер его телефона.
4. Передайте веб-страницу клиенту.

Таким образом, в окне браузера будет воспроизведен ответ, динамически сформированный сервлетом.

Пакет `javax.servlet.http`

Для демонстрации основных функциональных возможностей сервлетов в приведенных ранее примерах применялись такие классы и интерфейсы из пакета `javax.servlet`, как `ServletRequest`, `ServletResponse` и `GenericServlet`. Но для работы с сетевым протоколом HTTP обычно применяются интерфейсы и классы из пакета `javax.servlet.http`. Как станет ясно в дальнейшем, его функциональные возможности упрощают создание сервлетов, обрабатывающих запросы и ответы по сетевому протоколу HTTP. В табл. 38.8 перечислены основные интерфейсы, предоставляемые в данном пакете.

В табл. 38.9 описаны основные классы, предоставляемые в данном пакете. Наиболее важным из них является класс `HttpServlet`. Разработчики сервлетов обычно расширяют этот класс для обработки HTTP-запросов.

Таблица 38.8. Основные интерфейсы из пакета `javax.servlet.http`

Интерфейс	Описание
<code>HttpServletRequest</code>	Позволяет вводить данные из HTTP-запроса в сервлет
<code>HttpServletResponse</code>	Позволяет выводить данные из сервлета в HTTP-ответ
<code>HttpSession</code>	Позволяет вводить и выводить данные сеанса связи

Таблица 38.9. Основные классы из пакета `javax.servlet.http`

Класс	Описание
<code>Cookie</code>	Позволяет хранить данные состояния на машине клиента
<code>HttpServlet</code>	Предоставляет методы для обработки запросов и ответов по сетевому протоколу HTTP

Интерфейс `HttpServletRequest`

Интерфейс `HttpServletRequest` реализуется контейнером сервлетов. Он позволяет получать из сервлета сведения о запросе клиента. В табл. 38.10 перечислены некоторые методы из этого класса.

Таблица 38.10. Методы из интерфейса `HttpServletRequest`

Метод	Описание
<code>String getAuthType()</code>	Возвращает схему аутентификации
<code>Cookie[] getCookies()</code>	Возвращает массив, содержащий cookie-файл в данном запросе
<code>long getDateHeader(String поле)</code>	Возвращает значение из указанного <i>поля</i> заголовка даты
<code>String getHeader(String поле)</code>	Возвращает значение из указанного <i>поля</i> заголовка
<code>Enumeration<String> getHeaderNames()</code>	Возвращает перечисление имен заголовков
<code>int getIntHeader(String поле)</code>	Возвращает целочисленный (<code>int</code>) эквивалент поля заголовка
<code>String getMethod()</code>	Возвращает метод для HTTP-запроса
<code>String getPathInfo()</code>	Возвращает любые сведения о пути, который следует после пути к сервлету и перед строкой запроса в URL
<code>String getPathTranslated()</code>	Возвращает любые сведения о пути, который следует после пути к сервлету и перед строкой запроса в URL после ее преобразования в настоящий путь
<code>String getQueryString()</code>	Возвращает любую строку запроса в URL
<code>String getRemoteUser()</code>	Возвращает имя пользователя, который составил данный запрос
<code>String getRequestedSessionId()</code>	Возвращает идентификатор сеанса связи
<code>String getRequestURI()</code>	Возвращает URL
<code>StringBuffer getRequestURL()</code>	Возвращает URL
<code>String getServletPath()</code>	Возвращает ту часть URL, которая обозначает сервлет
<code>HttpSession getSession()</code>	Возвращает сеанс связи по данному запросу. Если сеанс связи не существует, он создается, а затем возвращается
<code>HttpSession getSession(Boolean номин)</code>	Если параметр <i>номин</i> принимает логическое значение <code>true</code> и сеанс связи не существует, то сеанс связи создается и возвращается по данному запросу. В противном случае возвращается существующий сеанс связи по данному запросу

Окончание табл. 38.10

Метод	Описание
boolean isRequestedSession- IdFromCookie()	Возвращает логическое значение true , если cookie-файл содержит идентификатор сеанса связи, а иначе — логическое значение false
boolean isRequestedSession- IdFromURL()	Возвращает логическое значение true , если URL содержит идентификатор сеанса связи, а иначе — логическое значение false
boolean isRequestedSession- IdValid()	Возвращает логическое значение true , если запрошенный идентификатор сеанса является действительным в текущем содержимом сеанса связи

Интерфейс HttpServletResponse

Интерфейс `HttpServletResponse` позволяет сформировать в сервлете HTTP-ответ для клиента. В нем определяется ряд констант, соответствующих кодам различных состояний, которые можно присвоить HTTP-ответу. Например, значение константы `SC_OK` обозначает, что HTTP-ответ достиг цели, а значение константы `SC_NOT_FOUND` — запрошенный ресурс недоступен. В табл. 38.11 перечислены некоторые методы из этого интерфейса.

Таблица 38.11. Методы из интерфейса HttpServletResponse

Метод	Описание
void addCookie(Cookie cookie)	Вводит указанный <i>cookie</i> -файл в HTTP-ответ
boolean containsHeader(String поле)	Возвращает логическое значение true , если в заголовке HTTP-ответа содержится заданное <i>поле</i>
String encodeURL(String url)	Определяет, должен ли идентификатор сеанса связи быть закодированным в указанном URL. Если должен, то возвращается измененная версия указанного URL, а иначе — сам указанный URL. Этим методом должны обрабатываться все URL, сформированные сервлетом
String encodeRedirectURL(String url)	Определяет, должен ли идентификатор сеанса связи быть закодированным в указанном URL. Если должен, то возвращается измененная версия указанного URL, а иначе — сам указанный URL. Этим методом должны обрабатываться все URL, передаваемые методу sendRedirect()
void sendError(int c) throws IOException	Посылает клиенту код ошибки, обозначаемый параметром <i>c</i>
void sendError(int c, String s) throws IOException	Посылает клиенту код ошибки, обозначаемый параметром <i>c</i> , а также строку сообщения, определяемую параметром <i>s</i>
void sendRedirect(String url) throws IOException	Переадресовывает клиента по указанному URL

Метод	Описание
<code>void setDateHeader(String поле, long <i>миллисекунд</i>)</code>	Вводит указанное поле в заголовок со значением даты, вычисляемой в миллисекундах . Отсчет времени ведется в миллисекундах, начиная с полуночи 1 января 1970 года (GMT — среднее время по Гринвичу)
<code>void setHeader(String поле, String <i>значение</i>)</code>	Вводит указанное поле в заголовок со значением, обозначаемым параметром значение
<code>void setIntHeader(String поле, int <i>значение</i>)</code>	Вводит указанное поле в заголовок со значением, обозначаемым параметром значение
<code>void setStatus(int код)</code>	Устанавливает указанный код состояния данного ответа

Интерфейс HttpSession

Интерфейс HttpSession позволяет читать и записывать в сесслете данные состояния, связанные с сеансом связи по сетевому протоколу HTTP. Некоторые из методов этого интерфейса перечислены в табл. 38.12. Каждый из них генерирует исключение типа `IllegalStateException`, если сеанс связи уже недействительный.

Таблица 38.12. Методы из интерфейса HttpSession

Метод	Описание
<code>Object getAttribute(String <i>атрибут</i>)</code>	Возвращает значение, связанное с именем, передаваемым в качестве параметра атрибут . Возвращает пустое значение <code>null</code> , если указанный атрибут не найден
<code>Enumeration<String> getAttributeNames()</code>	Возвращает перечисление имен атрибутов, связанных с сеансом связи
<code>long getCreationTime()</code>	Возвращает время, прошедшее с момента создания сеанса связи. Отсчет времени ведется в миллисекундах, начиная с полуночи 1 января 1970 года (GMT — среднее время по Гринвичу)
<code>String getId()</code>	Возвращает идентификатор сеанса связи
<code>long getLastAccessedTime()</code>	Возвращает время, прошедшее с того момента, когда клиент сделал последний запрос в вызывающем сеансе связи. Отсчет времени ведется в миллисекундах, начиная с полуночи 1 января 1970 года (GMT — среднее время по Гринвичу)
<code>void invalidate()</code>	Отменяет данный сеанс связи и удаляет его из контекста
<code>boolean isNew()</code>	Возвращает логическое значение <code>true</code> , если сервер создал сеанс связи, еще не доступный клиенту
<code>void removeAttribute(String <i>атрибут</i>)</code>	Удаляет заданный атрибут из сеанса связи
<code>void setAttribute(String <i>атрибут</i>, Object <i>значение</i>)</code>	Связывает указанное значение с именем заданного атрибута

Класс Cookie

Класс `Cookie` инкапсулирует *cookie-файл*, который хранится на стороне клиента и содержит данные состояния. Пользоваться *cookie-файлами* удобно для отслеживания активности пользователей. Допустим, пользователь посещает интернет-магазин. В *cookie-файле* можно сохранить имя пользователя, адрес и прочие сведения о нем. В этом случае пользователю не нужно вводить эти данные всякий раз, когда он посещает этот интернет-магазин.

Сервлет может сохранить *cookie-файл* на машине пользователя с помощью метода `addCookie()` из интерфейса `HttpServletResponse`. Данные из этого файла затем включаются в заголовок HTTP-ответа, который отправляется браузеру.

Имена и значения из *cookie-файлов* хранятся на машине пользователя. Часть сведений, сохраняемых из каждого *cookie-файла*, включает следующее:

- имя *cookie-файла*;
- значение из *cookie-файла*;
- срок действия *cookie-файла*;
- домен и путь к *cookie-файлу*.

Срок действия определяет, когда данный *cookie-файл* будет удален из машины пользователя. Если дата окончания срока действия *cookie-файла* не назначена явным образом, *cookie-файл* удаляется по завершении текущего сеанса связи с браузером.

Домен и путь к *cookie-файлу* определяют, когда он будет включен в заголовок HTTP-запроса. Если пользователь вводит URL, где домен и путь совпадают с этими значениями, то *cookie-файл* предоставляется веб-серверу, в противном случае — не предоставляется.

У класса `Cookie` имеется единственный конструктор. Ниже приведена его общая форма.

`Cookie(String имя, String значение)`

Здесь параметры *имя* и *значение* обозначают соответственно имя и значение, которые передаются конструктору из *cookie-файла*. Методы из класса `Cookie` перечислены в табл. 38.13.

Табл. 38.13. Методы из класса `Cookie`

Метод	Описание
<code>Object clone()</code>	Возвращает копию данного объекта
<code>String getComment()</code>	Возвращает комментарий
<code>String getDomain()</code>	Возвращает домен
<code>int getMaxAge()</code>	Возвращает максимальный срок действия <i>cookie-файла</i> (в секундах)
<code>String getName()</code>	Возвращает имя
<code>String getPath()</code>	Возвращает путь
<code>boolean getSecure()</code>	Возвращает логическое значение <code>true</code> , если <i>cookie-файл</i> безопасный, а иначе — логическое значение <code>false</code>
<code>String getValue()</code>	Возвращает значение

Метод	Описание
<code>int getVersion()</code>	Возвращает версию
<code>boolean isHttpOnly()</code>	Возвращает логическое значение true , если cookie-файл содержит атрибут HttpOnly
<code>void setComment(String c)</code>	Присваивает заданный комментарий c
<code>void setDomain(String d)</code>	Присваивает заданный домен d
<code>void setComment(String c)</code> <code>void setHttpOnly(boolean только_Http)</code>	Если параметр только_Http принимает логическое значение true , то атрибут HttpOnly вводится в cookie-файл. А если параметр только_Http принимает логическое значение false , то атрибут HttpOnly удаляется
<code>void setMaxAge(int секунд)</code>	Устанавливает максимальный срок действия cookie-файла в секундах. Этот срок определяется количеством секунд, по истечении которых cookie-файл удаляется
<code>void setPath(String p)</code>	Присваивает заданный путь p
<code>void setSecure(Boolean безопасно)</code>	Устанавливает признак безопасности, определяемый параметром безопасно
<code>void setValue(String v)</code>	Устанавливает заданное значение v
<code>void setVersion(int v)</code>	Присваивает заданную версию v

Класс HttpServlet

Класс `HttpServlet` расширяет класс `GenericServlet` и обычно применяется при разработке сервлетов, получающих и обрабатывающих HTTP-запросы. Методы из класса `HttpServlet` перечислены в табл. 38.14.

Таблица 38.14. Методы из класса HttpServlet

Метод	Описание
<code>void doDelete(HttpServletRequestRequest запрос, HttpServletResponse ответ)</code> <code>throws IOException, ServletException</code>	Обрабатывает HTTP-запрос типа DELETE
<code>void doGet(HttpServletRequestRequest запрос, HttpServletResponse ответ)</code> <code>throws IOException, ServletException</code>	Обрабатывает HTTP-запрос типа GET
<code>void doOptions(HttpServletRequestRequest запрос, HttpServletResponse ответ)</code> <code>throws IOException, ServletException</code>	Обрабатывает HTTP-запрос типа OPTIONS
<code>void doPost(HttpServletRequestRequest запрос, HttpServletResponse ответ)</code> <code>throws IOException, ServletException</code>	Обрабатывает HTTP-запрос типа POST
<code>void doPut(HttpServletRequestRequest запрос, HttpServletResponse ответ)</code> <code>throws IOException, ServletException</code>	Обрабатывает HTTP-запрос типа PUT
<code>void doTrace(HttpServletRequestRequest запрос, HttpServletResponse ответ)</code> <code>throws IOException, ServletException</code>	Обрабатывает HTTP-запрос типа TRACE

Окончание табл. 38.14

Метод	Описание
Long getLastModified(<i>HttpServletRequest</i> <i>запрос</i>)	Возвращает момент времени, измеряемого в миллисекундах после полуночи 1 января 1970 года (GMT – среднее время по Гринвичу), когда в последний раз был изменен запрошенный ресурс
void service(<i>HttpServletRequest</i> <i>запрос</i>, <i>HttpServletResponse</i> <i>ответ</i>) throws <i>IOException</i>, <i>ServletException</i>	Вызывается сервером, когда для данного сервлета поступает HTTP-запрос. Параметры <i>запрос</i> и <i>ответ</i> предоставляют доступ к HTTP-запросу и ответу соответственно

Обработка HTTP-запросов и ответов HTTP

В классе `HttpServletRequest` предоставляются специальные методы для обработки разнотипных HTTP-запросов. Разработчики сервлетов обычно переопределяют один из этих методов. К их числу относятся методы `doDelete()`, `doGet()`, `doHead()`, `doOptions()`, `doPost()`, `doPut()` и `doTrace()`. Привести здесь полное описание различных типов HTTP-запросов не представляется возможным. Но чаще всего при обращении с заполняемыми формами применяются HTTP-запросы типа GET и POST, и поэтому далее приводятся примеры из обработки в сервлетах.

Обработка HTTP-запросов типа GET

В этом разделе представлен пример разработки сервлета, обрабатывающего запрос HTTP-запрос типа GET. Этот сервлет вызывается, когда передается форма, заполненная на веб-странице. Данный пример состоит из двух файлов. В частности, разметка веб-страницы определяется в HTML-файле `ColorGet.html`, а сервлет – в файле `ColorGetServlet.java`. Ниже приведен исходный код разметки из HTML-файла `ColorGet.html`. В этом коде определяется форма, содержащая элемент выбора и кнопку для передачи формы. Обратите внимание на то, что параметр `action` дескриптора `<form>` определяет URL. Этот URL обозначает сервлет для обработки HTTP-запроса типа GET.

```
<html>
<body>
<center>
<form name="Form1"
      action="http://localhost:8080/examples/servlets
              /servlet/ColorGetServlet">
<B>Color:</B>
<select name="color" size="1">
<option value="Red">Red</option>
<option value="Green">Green</option>
<option value="Blue">Blue</option>
</select>
<br><br>
<input type="submit" value="Submit">
</form>
</body>
</html>
```

Ниже приведен исходный код сервлета из файла `ColorGetServlet.java`. Метод `doGet()` переопределяется для обработки любых HTTP-запросов типа GET, посылаемых данному сервлету. Чтобы получить результаты выбора, сделанного пользователем, в сервлете вызывается метод `getParameter()` из интерфейса `HttpServletRequest`. После этого формируется ответ по данному запросу.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ColorGetServlet extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {

        String color = request.getParameter("color");
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>The selected color is: ");
        pw.println(color);
        pw.close();
    }
}
```

Сначала скомпилируйте сервлет, а затем скопируйте его в соответствующий каталог и обновите файл `web.xml`, как пояснялось ранее. Далее выполните следующие действия для проверки сервлета из данного примера.

1. Запустите на выполнение контейнер сервлетов Tomcat, если он еще не действует.
2. Выведите веб-страницу в окне браузера.
3. Выберите нужный цвет.
4. Передайте форму, заполненную на веб-странице.

После этого браузер выведет результат, динамически сформированный сервлетом. Следует также иметь в виду, что параметры для HTTP-запроса типа GET включены как составная часть в URL, посылаемый веб-серверу. Допустим, пользователь выбрал красный цвет и передал заполненную форму. URL, посылаемый серверу из браузера, будет выглядеть так, как показано ниже. Символы, стоящие справа от вопросительного знака, составляют *строку запроса*.

```
http://localhost:8080/examples/servlets/servlet/ColorGetServlet?color=Red
```

Обработка HTTP-запросов типа POST

А теперь перейдем к примеру разработки сервлета, обрабатывающего HTTP-запрос типа POST. Этот сервлет вызывается при передаче формы, заполненной на веб-странице. Данный пример состоит из двух файлов. В частности, разметка веб-страницы определяется в HTML-файле `ColorPost.html`, а сервлет — в файле `ColorPostServlet.java`.

Ниже приведен исходный код разметки веб-страницы из HTML-файла `ColorPost.html`. Он похож на код из HTML-файла `ColorGet.html`, за исключением того, что параметр `method` дескриптора `<form>` явным образом определяет,

что следует использовать метод POST, а в параметре action того же дескриптора указан другой сервлет.

```
<html>
<body>
<center>
<form name="Form1"
      method="post"
      action="http://localhost:8080/examples/servlets
            /servlet/ColorPostServlet">

<B>Color:</B>
<select name="color" size="1">
<option value="Red">Red</option>
<option value="Green">Green</option>
<option value="Blue">Blue</option>
</select>
<br><br>
<input type="submit" value="Submit">
</form>
</body>
</html>
```

Ниже приведен исходный код сервлета из файла ColorPostServlet.java. Метод doPost() заменяется для обработки любых HTTP-запросов типа POST, отправляемых данному сервлету. Для получения результатов выбора, сделанного пользователем, в данном сервлете вызывается метод getParameter() из интерфейса HttpServletRequest. После этого формируется ответ на данный запрос.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ColorPostServlet extends HttpServlet {

    public void doPost(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {

        String color = request.getParameter("color");
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>The selected color is: ");
        pw.println(color);
        pw.close();
    }
}
```

Скомпилируйте данный сервлет. Чтобы проверить его, выполните те же действия, что и в предыдущем разделе.

На заметку! Параметры HTTP-запроса типа POST не включаются в состав URL, отправляемый веб-серверу. В данном примере браузер посылает серверу следующий URL: `http://localhost:8080/examples/servlets/servlet/ColorPostServlet`. Имена и значения параметров отправляются в теле HTTP-запроса.

Применение cookie-файлов

А теперь рассмотрим пример разработки сервлета, чтобы продемонстрировать применение cookie-файлов. Этот сервлет вызывается при передаче формы,

заполненной на веб-странице. Данный пример состоит из трех файлов, перечисленных в табл. 38.15.

Таблица 38.15. Файлы, составляющие пример применения cookie-файлов

Файл	Описание
AddCookie.html	Дает пользователю возможность определить значение для cookie-файла, называемого MyCookie
AddCookieServlet.java	Обрабатывает передачу формы из файла AddCookie.html
GetCookiesServlet.java	Отображает значения из cookie-файла

Ниже приведен исходный код разметки веб-страницы из HTML-файла AddCookie.html. Эта страница содержит текстовое поле для ввода значения, а также кнопку для передачи заполненной формы. Если щелкнуть на этой кнопке, значение в текстовом поле будет передано сервлету AddCookieServlet по HTTP-запросу типа POST.

```
<html>
<body>
<center>
<form name="Form1"
      method="post"
      action="http://localhost:8080/examples/servlets
            /servlet/AddCookieServlet">
<B>Enter a value for MyCookie:</B>
<input type="text" name="data" size=25 value="">
<input type="submit" value="Submit">
</form>
</body>
</html>
```

Ниже приведен исходный код сервлета из файла AddCookieServlet.java. Он получает сначала значение параметра по имени data, а затем создает объект MyCookie типа Cookie, который содержит значение параметра data. После этого в заголовок HTTP-ответа вводится cookie-файл методом addCookie(). И наконец, браузер получает ответное сообщение.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class AddCookieServlet extends HttpServlet {

    public void doPost(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {

        // получить параметр из HTTP-запроса
        String data = request.getParameter("data");

        // создать cookie-файл
        Cookie cookie = new Cookie("MyCookie", data);

        // ввести cookie-файл в HTTP-ответ
        response.addCookie(cookie);

        // вывести результат в окне браузера
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>MyCookie has been set to");
        pw.println(data);
    }
}
```

```

        pw.close();
    }
}

```

Ниже приведен исходный код одного сервлета из файла `GetCookieServlet.java`. Он вызывает метод `getCookie()` для чтения любых cookie-файлов, включенных в HTTP-запрос типа GET. Имена и значения этих cookie-файлов включаются в HTTP-ответ. Обратите внимание на то, что для получения этих данных вызываются методы `getName()` и `getValue()`.

```

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class GetCookiesServlet extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {

        // получить cookie-файлы из заголовка HTTP-запроса
        Cookie[] cookies = request.getCookies();

        // вывести все cookie-файлы
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.println("<B>");
        for(int i = 0; i < cookies.length; i++) {
            String name = cookies[i].getName();
            String value = cookies[i].getValue();
            pw.println("name = " + name +
                      "; value = " + value);
        }
        pw.close();
    }
}

```

Скомпилируйте эти сервлеты. Затем скопируйте их в соответствующий каталог и обновите файл `web.xml`, как пояснялось ранее. После этого выполните следующие действия, чтобы проверить данный пример.

1. Запустите контейнер сервлетов Tomcat на выполнение, если он еще не действует.
2. Откройте HTML-файл `AddCookie.html` веб-страницы в окне браузера.
3. Введите значение для cookie-файла `MyCookie`.
4. Передайте форму, заполненную на веб-странице.

Выполнив эти действия, вы увидите, что в окне браузера отображается ответное сообщение. Введите в поле адреса, находящемся в верхней части окна браузера, следующий URL:

`http://localhost:8080/examples/servlets/servlet/GetCookiesServlet`

В окне браузера должны быть отображены имя и значение из cookie-файла. В данном примере не применяется метод `setMaxAge()` из класса `Cookie` для явного назначения срока действия cookie-файлов. Поэтому срок действия cookie-файлов истекает по завершении сеанса связи с браузером. Если же воспользоваться методом `setMaxAge()`, то можно обнаружить, что cookie-файл будет сохранен на диске клиентской машины.

Отслеживание сеансов связи

Сетевой протокол HTTP действует без сохранения состояния. Каждый последующий запрос не зависит от предыдущего. Но в некоторых приложениях иногда требуется сохранять данные состояния, чтобы накапливать их в результате нескольких взаимодействий браузера и сервера. Такой механизм обеспечивают сеансы связи.

Сеанс связи можно создать с помощью метода `getSession()` из интерфейса `HttpServletRequest`. Он возвращает объект типа `HttpSession`. Этот объект способен сохранять ряд привязок имен к объектам. Этими привязками управляют методы `setAttribute()`, `getAttribute()`, `getAttributeNames()` и `removeAttribute()` из интерфейса `HttpSession`. Состояние сеанса связи совместно используется всеми сервлетами, связанными с определенным клиентом.

В приведенном ниже примере сервлета демонстрируется отслеживание состояния сеанса связи. Метод `getSession()` получает текущий сеанс связи. Если сеанс связи не существует, то создается новый сеанс связи. Метод `getAttribute()` вызывается для получения объекта, привязанного к имени `date`. Это объект типа `Date`, инкапсулирующий дату и время последнего доступа к данной веб-странице. (Разумеется, такая привязка отсутствует при первом доступе к странице.) Затем создается объект типа `Date`, инкапсулирующий текущие дату и время. Метод `setAttribute()` вызывается для привязки имени `date` к этому объекту.

```
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class DateServlet extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {

        // получить объект типа HttpSession
        HttpSession hs = request.getSession(true);

        // получить поток записи типа PrintWriter
        response.setContentType("text/html");
        PrintWriter pw = response.getWriter();
        pw.print("<B>");

        // вывести дату и время последнего доступа к странице
        Date date = (Date)hs.getAttribute("date");
        if(date != null) {
            pw.print("Last access: " + date + "<br>");
        }

        // вывести текущие дату и время
        date = new Date();
        hs.setAttribute("date", date);
        pw.println("Current date: " + date);
    }
}
```

При первом запросе этого сервлета в окне браузера выводится одна строка с информацией о текущих дате и времени. А при последующем вызове выводятся две строки. В первой строке указываются дата и время последнего доступа к сервлету, а во второй строке — текущие дата и время.



Применение документирующих комментариев в Java

Как пояснялось в части I данной книги, в языке Java поддерживаются три вида комментариев. Двумя первыми являются комментарии `//` и `/* */`. А третий вид называется *документирующим комментарием*. Такой комментарий начинается с последовательности символов `/**` и завершается последовательностью символов `*/`. Документирующие комментарии позволяют ввести в программу сведения о ней самой, а затем извлечь их с помощью утилиты `javadoc`, входящей в состав JDK, и разместить в HTML-файле. Документирующие комментарии упрощают процесс написания документации к разрабатываемым программам. Вам, должно быть, уже встречалась документация, составленная с помощью утилиты `javadoc`, поскольку именно она использована для документирования прикладного программного интерфейса Java API.

Дескрипторы утилиты `javadoc`

Утилита `javadoc` распознает дескрипторы, перечисленные в табл. А.1.

Таблица А.1. Дескрипторы утилиты `javadoc`

Дескриптор	Назначение
<code>@author</code>	Идентифицирует автора
<code>{@code}</code>	Отображает информацию в исходном виде, т.е. без обработки стилей HTML-разметки, используя шрифт кода
<code>@deprecated</code>	Обозначает, что класс или его член не рекомендуется для применения
<code>{@docRoot}</code>	Обозначает путь к корневому каталогу текущей документации
<code>@exception</code>	Обозначает исключение, генерируемое методом или конструктором
<code>{@inheritDoc}</code>	Наследует комментарий от непосредственного суперкласса
<code>{@link}</code>	Вставляет встроенную ссылку на другую тему
<code>{@linkplain}</code>	Вставляет встроенную ссылку на другую тему, причем ссылка отображается обычным шрифтом
<code>{@literal}</code>	Отображает информацию в исходном виде, т.е. без обработки стилей HTML-разметки

Дескриптор	Назначение
<code>@param</code>	Документирует параметр, передаваемый методу
<code>@return</code>	Документирует значение, возвращаемое методом
<code>@see</code>	Обозначает ссылку на другую тему
<code>@serial</code>	Документирует поле, сериализуемое по умолчанию
<code>@serialData</code>	Документирует данные, записываемые методами <code>writeObject()</code> и <code>writeExternal()</code>
<code>@serialField</code>	Документирует компонент <code>ObjectStreamField</code>
<code>@since</code>	Обозначает выпуск, в котором было введено определенное изменение
<code>@throws</code>	То же, что и дескриптор <code>@exception</code>
<code>{@value}</code>	Отображает значение константы, которая должна быть статическим (<code>static</code>) полем
<code>@version</code>	Определяет версию класса

Дескрипторы утилиты **javadoc**, начинающиеся со знака `@`, называются *автономными* (или *дескрипторами блоков*) и должны использоваться в отдельной строке. А дескрипторы, начинающиеся с фигурной скобки, например `{@code}`, называются *встроенными* и могут применяться в более крупном описании. В документирующих комментариях можно использовать и другие стандартные дескрипторы HTML-разметки. Но некоторые дескрипторы (например, заголовков) нельзя использовать, потому что они нарушают внешний вид HTML-файла, сформированный утилитой **javadoc**.

Документирующие комментарии можно применять для документирования классов, интерфейсов, полей, конструкторов и методов. Но в любом случае документирующий комментарий должен стоять перед документируемым элементом. Одни дескрипторы, например `@see`, `@since` и `@deprecated`, применяются для документирования любого элемента, а другие — только для соответствующих элементов. Каждый из этих дескрипторов рассматривается далее в отдельности.

На заметку! Комментарии могут быть также использованы для документирования пакета и подготовки краткого обзора, но эти процедуры отличаются от используемых для документирования исходного кода. Подробнее об этом можно узнать из документации на утилиту **javadoc**.

Дескриптор `$author`

Документирует сведения об авторе класса и имеет следующий синтаксис:

`@author описание`

где параметр *описание* обычно обозначает Ф.И.О. того, кто разработал класс. Выполняя утилиту `javadoc`, следует указать параметр `-author`, чтобы включить в HTML-документацию поле дескриптора `@author`.

Дескриптор `{@code}`

Позволяет встраивать в комментарий текст (например, фрагмент кода). Этот текст будет отображаться шрифтом кода без последующей обработки (например, без воспроизведения в формате HTML). Это дескриптор имеет следующий синтаксис:

```
{@code фрагмент_кода}
```

Дескриптор `@deprecated`

Определяет устаревший и не рекомендованный к употреблению элемент программы. Чтобы уведомить программиста об имеющихся альтернативных вариантах, рекомендуется включать в исходный код программы дескрипторы `@see` или `@link`. Синтаксис этого дескриптора выглядит следующим образом:

```
@deprecated описание
```

где параметр *описание* обозначает сообщение, описывающее исключение. Дескриптор `@deprecated` можно использовать для документирования полей, методов, конструкторов и классов.

Дескриптор `{@docRoot}`

Дескриптор `{@docRoot}` определяет путь к корневому каталогу текущей документации.

Дескриптор `@exception`

Дескриптор `@exception` описывает исключение в данном методе. Он имеет следующий синтаксис:

```
@exception имя_исключения пояснение
```

где параметр *имя_исключения* обозначает полное имя исключения, а параметр *пояснение* — символьную строку, описывающую причины, по которым может возникнуть данное исключение. Дескриптор `@exception` можно использовать только для документирования методов или конструкторов.

Дескриптор `{@inheritDoc}`

Наследует комментарий от непосредственного суперкласса.

Дескриптор **{@link}**

Предоставляет встроенную ссылку на дополнительную информацию. Он имеет следующий синтаксис:

```
{@link пакет.класс#член текст}
```

где параметр *пакет.класс#член* обозначает имя класса или метода, на который вводится ссылка, а параметр *текст* — отображаемую символьную строку.

Дескриптор **{@linkplain}**

Вставляет встроенную ссылку на другую тему. Ссылка отображается обычным шрифтом. А в остальном этот дескриптор аналогичен дескриптору **{@link}**.

Дескриптор **{@literal}**

Позволяет встраивать текст в комментарий. Этот текст отображается в исходном виде, т.е. без последующей обработки (например, без воспроизведения в формате HTML). Ниже приведен синтаксис этого дескриптора, где параметр *описание* обозначает встраиваемый текст.

```
{@literal описание}
```

Дескриптор **@param**

Документирует параметр и имеет следующий синтаксис:

```
@param имя_параметра пояснение
```

где параметр *имя_параметра* обозначает имя документируемого параметра. Назначение этого параметра раскрывает предоставляемое *пояснение*. Дескриптор **@param** можно использовать только для документирования метода, конструктора, обобщенного класса или интерфейса.

Дескриптор **@return**

Описывает значение, возвращаемое методом, и имеет следующий синтаксис:

```
@return пояснение
```

где параметр *пояснение* описывает тип и смысл значения, возвращаемого методом. Дескриптор **@return** можно использовать только для документирования методов.

Дескриптор **@see**

Предоставляет ссылку на дополнительную информацию. Ниже приведены наиболее употребительные формы этого дескриптора.

@see *привязка*
@see *пакет.класс#член текст*

В первой форме параметр *привязка* обозначает ссылку на абсолютный или относительный URL. Во второй форме параметр *пакет.класс#член* обозначает имя элемента, а параметр *текст* — текст, отображаемый для данного элемента. Параметр *текст* является необязательным, и если он не указан, то отображается элемент, обозначаемый параметром *пакет.класс#член*. Имя члена также является необязательным. Таким образом, можно определить ссылку на пакет, класс или интерфейс, помимо ссылки на конкретный метод или поле. Имя может быть определено полностью или частично. Но точку, стоящую перед именем члена (если таковой существует), необходимо заменить на знак #.

Дескриптор @serial

Определяет комментарий к полю, которое сериализуется по умолчанию. Ниже приведен его синтаксис, где параметр *описание* обозначает комментарий к данному полю:

@serial *описание*

Дескриптор @serialData

Документирует данные, записываемые с помощью методов `writeObject()` и `writeExternal()`. Ниже приведен его синтаксис, где параметр *описание* обозначает комментарий к этим данным.

@serialData *описание*

Дескриптор @serialField

Для класса, реализующего интерфейс `Serializable`, дескриптор `@serialField` предоставляет комментарии к компоненту `ObjectStreamField`. Ниже приведен его синтаксис, где параметр *имя* обозначает имя поля; параметр *тип* — конкретный тип поля; параметр *описание* — комментарий к данному полю.

@serialField *имя тип описание*

Дескриптор @since

Указывает на то, что класс или элемент был впервые внедрен в конкретном выпуске. Ниже приведен его синтаксис, где параметр *выпуск* обозначает символьную строку, в которой указывается выпуск или версия, в которых данное средство стало доступным.

@since *выпуск*

Дескриптор @throws

Имеет то же назначение, что и дескриптор @exception.

Дескриптор {@value}

Имеет две формы. В первой форме отображается значение константы, которая предшествует этому дескриптору и должна быть статическим (**static**) полем. Эта форма приведена ниже.

```
{@value}
```

Во второй форме отображается значение конкретного статического поля. Эта форма приведена ниже, где параметр `пакет.класс#поле` обозначает имя статического поля.

```
{@value пакет.класс#поле}
```

Дескриптор @version

Определяет версию класса или интерфейса. Он имеет следующий синтаксис:

```
@version информация
```

где параметр *информация* обозначает символьную строку, содержащую сведения о версии комментируемого элемента программы (как правило, номер версии, например 2.2). Запуская утилиту **javadoc** на выполнение, следует указать параметр **-version**, чтобы включить поле @version в HTML-документацию.

Общая форма документирующих комментариев

После начальной последовательности символов **/**** первая строка (или несколько строк) становится главным описанием комментируемого класса, интерфейса, поля, конструктора или метода. После нее можно ввести один или несколько различных дескрипторов @. Каждый дескриптор @ должен располагаться в начале новой строки или следовать за одним или несколькими знаками звездочки (*), с которых начинается строка. Несколько дескрипторов одного и того же типа нужно сгруппировать вместе. Так, если имеются три дескриптора @see, их следует расположить один за другим. Встроенные дескрипторы, начинающиеся с фигурной скобки, можно разместить в пределах любого описания.

Ниже приведен пример документирующего комментария к классу.

```
/**
 * Этот класс строит столбиковую диаграмму
 * @author Герберт Шилдт
 * @version 3.2
 */
```

Результаты, выводимые утилитой `javadoc`

В качестве выводимых данных утилита `javadoc` получает файл с исходным кодом прикладной программы на Java и выводит несколько HTML-файлов, содержащих документацию на эту программу. Сведения о каждом классе будут содержаться в его собственном HTML-файле. Утилита `javadoc` выводит также индексное и иерархическое деревья. Могут быть сформированы и другие HTML-файлы.

Пример применения документирующих комментариев

Ниже в качестве примера приведена простая программа, в которой применяются документирующие комментарии. Обратите внимание на то, что каждый комментарий располагается непосредственно перед тем элементом, который он описывает. После обработки утилитой `javadoc` документирующих комментариев к классу `SquareNum` их можно найти в HTML-файле `SquareNum.html`.

```
import java.io.*;
/**
 * В этом классе демонстрируется применение
 * документирующих комментариев
 * @author Герберт Шилдт
 * @version 1.2
 */
public class SquareNum {
    /**
     * Этот метод возвращает квадрат числа.
     * Это многострочное описание. В нем можно ввести
     * столько строк, сколько потребуется.
     * @param num Значение, которое требуется возвести в квадрат
     * @return num Значение, возведенное в квадрат
     */
    public double square(double num) {
        return num * num;
    }

    /**
     * Этот метод вводит число, заданное пользователем
     * @return Введенное значение типа double
     * @exception Если при вводе возникает ошибка, то
     *             генерируется исключение типа IOException
     * @see IOException
     */
    public double getNumber() throws IOException {
        // создать буферизированный поток чтения
        // типа BufferedReader, используя стандартный
        // поток ввода System.in
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader inData = new BufferedReader(isr);
        String str;
        str = inData.readLine();
        return (new Double(str)).doubleValue();
    }
}
```



```
* Этот метод демонстрирует применение метода square()
* @param args Не используется
* @exception Если при вводе возникает ошибка, то
*             генерируется исключение типа IOException
* @see IOException
*/
public static void main(String args[])
throws IOException
{
    SquareNum ob = new SquareNum();
    double val;
    System.out.println(
        "Введите значение для возведения в квадрат: ");
    val = ob.getNumber();
    val = ob.square(val);
    System.out.println("Квадрат значения равен " + val);
}
}
```

Предметный указатель

С

Collections Framework

- алгоритмы, 565
- изменения в версии JDK 5, 566
- назначение, 564
- состав, 565

F

Fork/Join Framework

- дескрипторы, назначение, 1058
- задачи
 - асинхронное выполнение, 1055
 - влияние уровня параллелизма на выполнение, 1049
 - возвращающие результаты, выполнение, 1053
 - организация очереди, 1045
 - отмена, 1056
 - перезапуск, 1056
 - перехват работы, принцип выполнения, 1045
 - состояние завершения, определение, 1056
- назначение, 49; 1040
- основные классы
 - взаимосвязь, 1041
 - разновидности, 1041
- поддержка параллельного программирования, 50; 1007
- рекомендации по применению, 1059
- стратегия "разделяй и властвуй"
 - назначение, 1045
 - оптимальное пороговое значение, выбор, 1046
 - преимущества, 1046

A

Автоматическое управление ресурсами, преимущества, 371

Автораспаковка

- в выражениях, 330
- в методах, 329
- данных типа boolean и char, 332
- назначение, 328
- предупреждение против злоупотребления, 334

Автоупаковка

- в выражениях, 330
- в методах, 329
- данных типа boolean и char, 332
- предотвращение ошибок, 333

Аннотации

- все, получение, 339
- встроенные, назначение, 345; 347
- маркеры, назначение, 343
- механизм создания, 334
- назначение, 334
- объявление, 334
- одночленные, применение, 343
- повторяющиеся, применение, 352
- получение с помощью рефлексии, 336
- правила удержания, 335
- применение, 335
- типовые
 - применение, 347
 - целевые константы, 348

Аплеты

- архитектура, 837
- безопасность и переносимость, 41
- воспроизведение
 - методы, 841
 - повторное, 843
- графический контекст, 375
- заглушки, назначение, 837
- запуск, способы, 375
- консольный вывод, организация, 854
- назначение, 40; 374
- на основе библиотеки AWT, создание, 833
- Swing, создание, 1135
- обязательное подписание, 376; 835
- основные методы
 - переопределение, 838
 - порядок вызова, 839
- передача параметров, 848
- принцип действия, 837
- развертывание, способы, 834
- разработка
 - особенности, 377
 - стадии, 376
- строка состояния, применение, 846

типы, 833
 Аргументы
 командной строки
 назначение, 202
 порядок передачи, 203
 передача
 по значению, 184
 по ссылке, 185
 переменной длины
 и неоднозначность, 207
 применение, 203
 Архитектура
 MVC, составляющие, 1124
 компонентов Swing, особенности, 1124

Б

Библиотеки

AWT

актуальность, 885
 диалоговые окна, создание
 и применение, 968
 изображения, обработка, 975
 классы, разновидности, 886
 кнопки-переключатели, создание
 и применение, 933
 кнопки, создание и применение, 926
 компоненты, применение, 924
 меню, создание и применение, 962
 метки, создание и применение, 925
 назначение, 374
 ограниченность, 1122
 окна выбора файлов, диалоговые,
 создание и применение, 972
 переопределение метода paint(), 973
 поддержка графики, 900
 полосы прокрутки, создание
 и применение, 939
 применение, 886; 898
 раскрывающиеся списки, создание
 и применение, 934
 списки, создание и применение, 936
 строки и пункты меню, создание
 и применение, 962
 текстовые области, создание
 и применение, 945
 текстовые поля, создание
 и применение, 943
 тяжеловесные компоненты, 1122
 флажки, создание и применение, 930
 цветовая система, 904

JavaFX

ввод изображений в экранные кнопки,
 1238
 внедрение, 51; 1211
 воспроизведение изображений на месте
 меток, 1236
 графы сцены, 1213
 деревья, создание и применение, 1269
 дополнительные средства, 1308
 запуск приложений, 1214

кнопки-переключатели, создание
 и применение, 1243
 компиляция и выполнение
 приложений, 1219
 метки, создание и применение, 1220
 неопределенное состояние флажков,
 разрешение, 1251
 обработка событий, 1222
 одновременный выбор из списка,
 активизация, 1258
 отключение элементов управления, 1280
 панели компоновки, разновидности, 1213
 переключатели, создание
 и применение, 1240
 подмости и сцены, 1213
 подсказки всплывающие,
 применение, 1279
 представления списков, создание
 и применение, 1254
 преобразования, выполнение, 1275
 прокручиваемые панели, создание
 и применение, 1265
 разрабо ка, стадии, 1212
 рисование в режиме удержания, 1227
 списки прокручиваемые, создание
 и применение, 1258
 структура приложений, 1215
 текстовые поля, создание
 и применение, 1263
 узлы, разновидности, 1213
 фильтры событий, реализация, 1223
 флажки, создание и применение, 1251
 цепочка диспетчеризации событий, 1223
 экранные кнопки, создание
 и применение, 1223
 элементы управления, классы, 1233
 эффекты, применение, 1274
 Swing
 главные особенности, 1122
 действия, применение, 1196
 другие компоненты, назначение, 1208
 классы компонентов, 1143
 кнопки, классы, 1147
 компоненты и контейнеры,
 назначение, 1125
 лектовесные компоненты, 1123
 меню, система, 1175
 обработка событий, механизм, 1132
 подключаемые стили оформления, 1123
 построение на основе AWT, 1122
 представитель пользовательского
 интерфейса, 1124
 приложения и апплеты, 1127
 происхождение, 1121
 структура приложений, 1127
 динамически подключаемые, создание, 382
 классов, встроены, 73
 Блоки кода
 назначение, 69
 обозначение, 69

В

Ввод-вывод

- адресаты вывода, определение, 717
- ввод, данных с консоли, организация, 359
- возврат в поток ввода, механизм, 740
- вывод, данных на консоль, организация, 362
- источники ввода, определение, 717
- консольный
 - применение, 64
- поддержка в Java
- организация в Java
 - новая система NIO, 769
 - потокковая система, 717
- поточковый
 - в системе NIO, организация, 793
 - традиционный, организация, 717
 - через сеть, 813
- система NIO
 - буферы, назначение и свойства, 770
 - каналы, назначение и получение, 772
 - копирование файлов, 792
 - наборы символов, кодеры и декодеры, 774
 - новый канальный ввод-вывод, 783
 - операции в файловой системе, 796
 - поточковый ввод-вывод, 793
 - прежний канальный ввод-вывод, 804
 - применение, 770; 782
 - селекторы, назначение, 774
 - усовершенствования в JDK 7, 774

Веб

- cookie-файлы
 - назначение, 826; 1339
 - применение, 1343
 - содержимое, 1339
- URI, назначение, 826
- URL
 - назначение, 820
 - формат и составляющие, 820
- назначение, 819

Векторы

- емкость, определение, 636
- инкремент, задание, 636
- применение, 636
- создание, 636

Взаимная блокировка

- возникновение, 309
- трудности отладки, 309

Всплывающие меню

- в JavaFX
 - активизация, 1297
 - построение, 1296
- в Swing
 - запуск, 1191
 - построение, 1190
- контекстные, назначение, 1296
- назначение, 1190

Г

Графика

- графический контекст, способы получения, 900

- изменение размеров, 902
- установка цвета, 905

Д

Дейтаграммы

- обработка, пример, 830
- реализация в Java, 828

Дескрипторы

APPLET

- атрибуты, описание, 847
- полная форма, 847
- OBJECT, назначение, 834
- утилиты javadoc
 - автономные, 1348
 - встроенные, 1348
 - описание, 1348; 1352
 - разновидности, 1347

Динамический полиморфизм, механизм, 227

Диспетчеры компоновки

- вставки, применение, 952
- границной, реализация, 950
- карточной, реализация, 954
- назначение, 923
- поточной, реализация, 948
- принцип действия, 948
- сеточной, реализация, 953
- сеточно-контейнерной, реализация, 957

Древоидные множества

- извлечение элементов, 588
- применение, 587
- создание, 587

И

Идентификаторы, назначение, 71

Изображения

- воспроизведение, 977; 981
- встраивание в гипертекст, 977
- динамическое подключение фильтров, 993
- загрузка, 977
- наблюдатели, назначение, 977
- обработка, основные операции, 976
- создание объектов, 977
- фильтрация, 991
- форматы файлов, 975
- формирование, 975

Интерфейсы

Action

- методы и свойства действий, 1197
- назначение и реализация, 1196
- ActionListener, назначение и методы, 871; 1155
- AdjustmentListener, назначение и методы, 871
- AnnotatedElement, назначение и методы, 341; 353
- Annotation, расширение и методы, 334
- Appendable, реализация и методы, 559
- AppletContext, назначение и методы, 852
- AppletStub, назначение, 854
- AudioClip, назначение и методы, 854
- AutoCloseable, назначение и реализация, 371; 560; 725

- BaseStream, назначение и методы, 1062
- BasicFileAttributes, назначение и методы, 780
- BeanInfo
 - назначение и методы, 1315
 - реализация и применение, 1315
- ButtonModel, назначение, 1148
- Callable, назначение и методы, 1031
- CGI
 - назначение, 1324
 - недостатки, 1324
- ChangeListener, назначение, реализация и методы, 1247
- Channel, назначение и реализация, 773
- CharSequence, реализация и методы, 558
- Cloneable, назначение и реализация, 533
- Closeable, назначение и реализация, 371; 725
- Collection
 - методы, 568; 1065
 - назначение, 568
 - статические переменные, 628
- Collector, назначение и методы, 1080
- ComboBoxModel, назначение и реализация, 1166
- Comparable, назначение и реализация, 415; 558
- Comparator
 - методы, 614
 - назначение и реализация, 459; 614
- ComponentListener, назначение и методы, 872
- ContainerListener, назначение и методы, 872
- DataInput
 - методы, 747
 - назначение и реализация, 746
- DataOutput
 - методы, 746
 - назначение и реализация, 746
- Deque
 - методы, 577
 - назначение, 577
- DosFileAttributes, назначение и методы, 781
- Enumeration
 - методы, 635
 - назначение, 635
- EventHandler, назначение, реализация и методы, 1222
- ExecutorService, назначение и методы, 1028
- Executor, назначение и методы, 1028
- Externalizable, назначение и методы, 762
- FileFilter, назначение и реализация, 724
- FilenameFilter, назначение и реализация, 723
- FileVisitor, реализация и методы, 802
- Flushable, назначение и реализация, 725
- FocusListener, назначение и методы, 872
- Future, назначение и методы, 1032
- HttpServletRequest, назначение и методы, 1336
- HttpServletResponse, назначение, константы и методы, 1337
- HttpSession, назначение и методы, 1338
- Icon, назначение и реализация, 1144
- ImageConsumer, назначение и реализация, 988
- ImageObserver
 - назначение и методы, 979
 - реализация, 977
- ImageProducer, назначение и реализация, 986
- ItemListener, назначение и методы, 872
- Iterable, реализация и методы, 559
- Iterator
 - методы, 592; 1083
 - назначение и реализация, 591
 - применение, 593
- JNI, назначение, 381
- KeyListener, назначение и методы, 872
- List
 - методы, 571
 - назначение, 571
- ListIterator
 - методы, 592
 - назначение и реализация, 592
 - применение, 593
- ListModel, назначение, 1162
- ListSelectionListener, назначение, реализация и методы, 1163
- ListSelectionModel, назначение и константы, 1163
- Lock
 - методы, 1036
 - назначение и реализация, 1036
- Map
 - методы, 602
 - назначение, 601
- Map.Entry
 - методы, 608
 - назначение, 608
- MouseListener, назначение и методы, 873; 1191
- MouseMotionListener, назначение и методы, 873
- MouseWheelListener, назначение и методы, 873
- MutableComboBoxModel, назначение и реализация, 1166
- MutableTreeNode, назначение, реализация и методы, 1169
- NavigableMap
 - методы, 606
 - назначение, 606
- NavigableSet
 - методы, 574
 - назначение, 574
- ObjectInput, назначение и методы, 764
- ObjectOutput, назначение и методы, 763
- Observer
 - методы, 673
 - назначение и реализация, 672
 - применение, 674
- OpenOption, назначение и реализация, 779
- Path
 - методы, 775

назначение, 774
 PosixFileAttributes, назначение и методы, 781
 Queue
 методы, 576
 назначение, 576
 RandomAccess, назначение
 и реализация, 601
 Readable, назначение и методы, 560
 ReadWriteLock, назначение
 и реализация, 1038
 Remote, назначение, 1105
 Runnable
 метод `run()`, определение, 545
 назначение, 289
 применение, 289
 реализация, 292; 545
 ScheduledExecutorService, назначение, 1029
 Serializable, назначение и реализация, 762
 ServletConfig, назначение и методы, 1330
 ServletContext, назначение и методы, 1331
 ServletRequest, назначение и методы,
 1331; 1334
 ServletResponse, назначение и методы, 1332
 Servlet, реализация и методы, 1330
 Set, назначение, 573
 SortedMap
 методы, 605
 назначение, 605
 SortedSet
 методы, 573
 назначение, 573
 SplitIterator
 методы, 596; 1085
 назначение и особенности, 596
 полученные интерфейсы,
 назначение, 599
 применение, 597
 Stream, назначение и методы, 1063; 1087
 SwingConstants, назначение
 и константы, 114
 TableColumnModel, назначение, 1172
 TableModel, назначение, 1172
 TextListener, назначение и методы, 873
 Thread.UncaughtExceptionHandler,
 реализация и методы, 560
 Toggle, назначение и реализация, 1240; 1243
 TreeNode, назначение и методы, 1169
 TreeSelectionListener, назначение,
 реализация и методы, 1169
 UnicastRemoteObject, назначение, 1105
 WindowConstants, назначение
 и реализация, 1130
 WindowFocusListener, назначение
 и методы, 873
 WindowListener, назначение и методы, 873
 вложенные, применение, 249
 исполнителей, назначение
 и реализация, 1007
 коллекций, разновидности
 и назначение, 567
 назначение, 245

обобщенные, применение, 417
 объявление, 245
 отображений, разновидности, 601
 приемников событий, разновидности, 870
 применение, 250
 расширение, 255
 реализация, 246
 Исключения
 более точное повторное генерирование, 283
 в операциях ввода-вывода, обработка, 367
 встроенные, 276
 вывод описания, 268
 многократный перехват, 282
 необработываемые, 265
 непроверяемые, 276
 обработка вручную
 в блоках операторов `try/catch`, 266
 преимущества, 266
 определение, 263
 порядок
 генерирования, 263
 обработки, 263
 при вводе-выводе, обработка, 725
 применение, 28
 проверяемые, 277
 разнотипные, перехват, 268
 составление в цепочки и обработка, 280
 типы, 264
 Исходные файлы, именование
 и расширение, 61
 Итераторы
 в потоках данных
 типы, 1083
 назначение, 591
 получение, 593
 разделители
 в потоках данных, применение, 1085
 назначение, 596
 применение, 596
 характеристики, определение, 599

К

Каталоги
 дерево, перечисление, 801
 назначение, 722
 получение содержимого, 798
 просмотр содержимого, 722
 создание, 724
 фильтрация содержимого, способы, 799
 Классы
 AbstractAction, назначение
 и конструкторы, 1198
 AbstractButton, назначение и методы, 1147
 ActionEvent
 конструкторы и методы, 859
 назначение и константы, 859
 AdjustmentEvent
 конструкторы и методы, 860
 назначение и константы, 860
 Applet
 методы, 835; 977

- назначение, 374; 833
- Application, назначение и методы
- жизненного цикла, 1214
- ArrayDeque
 - конструкторы, 590
 - назначение, 589
 - применение, 590
- ArrayList
 - конструкторы, 581
 - назначение, 580
 - применение, 581
- Arrays
 - методы, 629; 633
 - назначение, 629
- AWTEvent, назначение и методы, 858
- BitSet
 - конструкторы и методы, 653
 - назначение, 653
 - применение, 655
- Boolean
 - константы, поля и конструкторы, 519
 - методы, 519
 - назначение, 519
- BorderLayout
 - конструкторы и константы, 950
 - назначение, 950
- BorderPane, назначение и методы, 1289
- Buffer
 - назначение и методы, 770
 - производные классы, 771
- BufferedInputStream, назначение и конструкторы, 738
- BufferedOutputStream, назначение и конструкторы, 739
- BufferedReader, назначение и конструкторы, 359; 755
- BufferedWriter, назначение и конструкторы, 756
- Button
 - методы, 927
 - назначение и конструкторы, 926
- ButtonGroup, назначение и конструкторы, 1155
- ByteArrayInputStream, назначение и конструкторы, 734
- ByteArrayOutputStream, назначение и конструкторы, 736
- Byte, Short, Integer, Long
 - константы и методы, 504
 - назначение и конструкторы, 503
- Calendar
 - константы, 663
 - методы, 661
 - назначение, 661
 - переменные экземпляра, 661
 - применение, 664
- Canvas
 - конструкторы, 1228
 - методы, 1229
 - назначение, 890
- CardLayout
 - методы, 955
 - назначение и конструкторы, 954
- Character
 - константы и методы, 514
 - методы, 518
 - назначение и конструкторы, 514
- CharArrayReader, назначение и конструкторы, 752
- CharArrayWriter, назначение и конструкторы, 754
- Checkbox
 - методы, 931
 - назначение и конструкторы, 930
- CheckboxGroup, назначение и методы, 933
- CheckboxMenuItem
 - методы, 964
 - назначение и конструкторы, 963
- CheckMenuItem, назначение, конструкторы и методы, 1294
- Choice, назначение и методы, 934
- Class
 - методы, 336; 535
 - назначение, 535
- ClassLoader, назначение, 538
- Collections, алгоритмы и методы, 622
- Collectors, назначение и методы, 1080
- Color
 - константы, 841
 - методы, 905
 - назначение и конструкторы, 904
- ComboBox
 - конструкторы, 1260
 - назначение, 1260
- Compiler, назначение, 545
- Component
 - методы, 841; 976
 - назначение, 889
- ComponentEvent
 - конструкторы и методы, 861
 - назначение и константы, 861
- Console
 - методы, 760
 - назначение, 759
 - применение, 761
- Container
 - методы, 924; 952
 - назначение, 889
- ContainerEvent
 - конструкторы и методы, 862
 - назначение и константы, 861
- ContextMenu, назначение и конструкторы, 1296
- Cookie
 - конструкторы и методы, 1339
 - назначение, 1339
- CountDownLatch
 - конструкторы и методы, 1014
 - назначение, 1014
 - применение, 1015
- CropImageFilter, применение, 991
- Currency

- методы, 679
- назначение и применение, 678
- CyclicBarrier
 - конструкторы и методы, 1016
 - назначение, 1016
 - применение, 1017
- DatagramPacket
 - методы, 829
 - назначение и конструкторы, 829
- DatagramSocket
 - методы, 828
 - назначение и конструкторы, 828
- DataInputStream, назначение и конструкторы, 746
- DataOutputStream, назначение и конструкторы, 746
- Date
 - конструкторы и методы, 660
 - назначение, 659
 - применение, 660
- DateFormat
 - константы и методы, 1109
 - назначение, 1109
- DateTimeFormatter
 - назначение и методы, 1115
 - применение, 1115
 - шаблоны форматирования, 1116
- DefaultMutableTreeNode, назначение, конструкторы и методы, 1169
- Dialog, назначение и конструкторы, 968
- Dictionary
 - методы, 641
 - назначение, 641
- Double
 - методы и константы, 499
 - назначение и конструкторы, 498
- Effect, встроенные эффекты, разновидности, 1274
- Enum
 - методы, 323; 557
 - назначение, 323; 556
- EnumMap
 - конструкторы, 613
 - назначение, 613
- EnumSet
 - назначение, 590
 - фабричные методы, 591
- Error, назначение, 264
- Event
 - методы, 1223
 - назначение и подклассы, 1222
- EventObject
 - методы, 858
 - назначение и конструкторы, 858
- EventSetDescriptor, назначение и методы, 1319
- Exception
 - конструкторы, 278
 - назначение, 264
- Exchanger
 - назначение, 1018
- объявление и методы, 1019
- применение, 1019
- File
 - конструкторы, 719
 - методы, 719
 - назначение, 718
- FileChannel, назначение и методы, 773
- FileDialog
 - методы, 972
 - назначение и конструкторы, 972
- FileInputStream
 - конструкторы и методы, 364
 - назначение, 364
- FileInputStream, назначение и конструкторы, 730
- FileOutputStream
 - конструкторы и методы, 364
 - назначение, 364
- FileOutputStream, назначение и конструкторы, 732
- FileReader, назначение и конструкторы, 751
- Files
 - методы, 777; 782
 - назначение, 776
- FileWriter, назначение и конструкторы, 751
- FilterOutputStream, назначение и конструкторы, 737
- Float
 - методы и константы, 499
 - назначение и конструкторы, 498
- FlowLayout
 - конструкторы и константы, 948
 - назначение, 948
- FocusEvent
 - конструкторы и методы, 862
 - назначение и константы, 862
- Font
 - конструкторы, 911
 - методы, 909
 - назначение, 909
 - переменные, 910
- FontMetrics
 - методы, 915
 - назначение, 914
- ForkJoinPool
 - конструкторы, 1044
 - методы, 1044; 1058
 - назначение, 1044
- ForkJoinTask, назначение и методы, 1042; 1057
- Formatter
 - закрытие объектов, способы, 694
 - конструкторы, 680
 - методы, 681
 - назначение, 680
- Frame
 - конструкторы, 890
 - методы, 891
 - назначение, 890
- FXCollections, назначение и методы, 1255

- GenericServlet, назначение и методы, 1328; 1333
- Glow
 - конструкторы и методы, 1274
 - назначение, 1274
- GraphicsContext, назначение и методы, 1227
- GraphicsEnvironment, назначение и методы, 910
- Graphics, назначение и методы, 375; 900; 977
- GregorianCalendar
 - методы, 665
 - назначение, 664
 - поля и конструкторы, 664
 - применение, 665
- GridBagConstraints
 - константы, 959
 - назначение, 958
 - поля ограничений, 958
 - статические поля, 959
- GridBagLayout
 - конструкторы и методы, 958
 - назначение, 957
- GridLayout, назначение и конструкторы, 953
- HashMap
 - конструкторы, 609
 - назначение, 609
 - применение, 610
- HashSet
 - конструкторы, 586
 - назначение, 585
 - применение, 586
- Hashtable
 - конструкторы, 642
 - методы, 643
 - назначение, 642
 - применение, 644
- HttpServlet, назначение и методы, 1340; 1341
- URLConnection
 - методы, 824
 - назначение, 824
 - применение, 825
- IdentityHashMap, назначение, 613
- Image
 - конструкторы, 1234
 - назначение, 1233
- ImageFilter, назначение и подклассы, 991
- ImageIcon, назначение и конструкторы, 1144
- ImageView
 - конструкторы, 1234
 - назначение, 1233
- Image, назначение, 976
- InetAddress и Inet6Address, назначение, 815
- InetAddress
 - методы экземпляра, 815
 - назначение, 813
 - фабричные методы, 814
- InheritableThreadLocal, назначение, 553
- InnerShadow, назначение и конструкторы, 1275
- InputEvent
 - методы, 863
 - назначение и константы, 863; 1186
- InputStreamReader, назначение, 359
- InputStream, назначение и методы, 728
- Insets, назначение и конструкторы, 952
- Inspector, назначение и методы, 1319
- ItemEvent
 - конструкторы и методы, 864
 - назначение и константы, 864
- JApplet, назначение, 1136
- javaafx.collections.ObservableList,
 - назначение, 1255
- javaafx.scene.control.Button
 - конструкторы, 1223
 - назначение, 1223
- javaafx.scene.control.CheckBox,
 - назначение, 1251
- javaafx.scene.control.Label
 - конструкторы и методы, 1220
 - назначение, 1220
- javaafx.scene.control.MenuBar, назначение,
 - конструкторы и методы, 1283
- javaafx.scene.control.MenuItem, назначение,
 - конструкторы и методы, 1285
- javaafx.scene.control.Menu, назначение и конструкторы, 1284
- javaafx.scene.paint.Color, назначение и поля, 1229; 1275
- javaafx.scene.text.Font, назначение и конструкторы, 1229
- JButton
 - конструкторы, 1134; 1148
 - методы, 1134
 - назначение, 1134; 1148
- JCheckBoxMenuItem, назначение и конструкторы, 1188
- JCheckBox, назначение и конструкторы, 1153
- JComboBox
 - методы, 1166
 - назначение и конструкторы, 1166
- JComponent
 - методы, 1140
 - назначение, 1125
- JFrame, назначение и методы, 1129
- JLabel, назначение и конструкторы, 1143
- JList
 - методы, 1163
 - назначение и конструкторы, 1162
- JMenu
 - конструкторы, 1179
 - методы, 1179
 - назначение, 1178
- JMenuBar, назначение и методы, 1177
- JMenuItem
 - конструкторы, 1180
 - назначение, 1180
- JPopupMenu, назначение и конструкторы, 1190
- JRadioButtonMenuItem, назначение и конструкторы, 1188

- JRadioButton, назначение и конструкторы, 1155
- JScrollPane, назначение и конструкторы, 1160
- JTabbedPane, назначение и конструкторы, 1157
- JTable, назначение и конструкторы, 1172
- JTextField, назначение и конструкторы, 1145
- JToggleButton, назначение и конструкторы, 1150
- JToolBar, назначение и конструкторы, 1194
- JTree, назначение и конструкторы, 1168
- KeyCombination, назначение и методы, 1292
- KeyEvent
 - конструкторы и методы, 865
 - назначение и константы, 864; 1185
- Label
 - методы, 925
 - назначение и конструкторы, 925
- LinkedHashMap
 - конструкторы, 612
 - методы, 613
 - назначение, 612
- LinkedHashSet
 - назначение, 587
 - применение, 587
- LinkedList
 - конструкторы, 584
 - назначение, 584
 - применение, 584
- List
 - методы, 937
 - назначение и конструкторы, 936
- ListSelectionEvent, назначение и методы, 1163
- ListView
 - конструкторы, 1255
 - назначение, 1254
- LocalDate
 - назначение и методы, 1113
- LocalDateTime
 - назначение и методы, 1113
 - применение, 1114
- Locale
 - константы и конструкторы, 669
 - методы, 669
 - назначение, 668
- LocalTime
 - назначение и методы, 1113
- Matcher
 - назначение и методы, 1092
 - применение, 1096
- Math
 - методы, 539; 542
 - назначение и константы, 539
- MediaTracker, применение и методы, 983
- MemoryImageSource, назначение и конструкторы, 986
- MenuBar, назначение и методы, 964
- MenuItem
 - методы, 963
- назначение и конструкторы, 963
- Menu, назначение и конструкторы, 962
- MethodDescriptor, назначение и методы, 1319
- Modifier, назначение и методы, 1103
- MouseEvent
 - конструкторы и методы, 865; 1191
 - назначение и константы, 865
- MouseWheelEvent
 - конструкторы и методы, 867
 - назначение и константы, 867
- MultipleSelectionModel, назначение, 1256
- Naming, назначение и методы, 1106
- Node
 - методы, 1223; 1275
 - назначение, 1213
- Number, назначение, методы и подклассы, 498
- Object
 - методы, 233; 474; 532
 - назначение, 233; 532
- ObjectInputStream
 - конструкторы и методы, 765
 - назначение, 765
- ObjectOutputStream
 - конструкторы и методы, 763
 - назначение, 763
- Observable
 - методы, 672
 - назначение, 672
 - применение, 674
- ObservableList
 - методы, 1221
 - назначение, 1220
- Optional
 - методы, 657
 - назначение, 657
 - применение, 658
- OptionalDouble, OptionalInt, OptionalLong, назначение, 659
- OutputStream, назначение и методы, 729
- Package, назначение и методы, 553
- Paint, назначение и подклассы, 1229
- Panel, назначение, 889
- Paths, назначение и методы, 779
- Pattern
 - назначение и методы, 1092
- Phaser
 - конструкторы, 1021
 - методы, 1021
 - назначение, 1021
 - применение, 1021
- PixelGrabber
 - конструкторы и методы, 988
 - назначение, 988
- Platform, назначение и методы, 1291
- PopupMenu, назначение, 968
- PrintStream
 - конструкторы, 743
 - методы, 744
 - назначение, 743

- PrintStream, назначение и методы, 362
- PrintWriter
 - методы, 363; 759
 - назначение и конструкторы, 363; 758
- PriorityQueue
 - конструкторы, 589
 - назначение, 588
 - применение, 589
- ProcessBuilder
 - методы, 526
 - назначение и конструкторы, 525
- ProcessBuilder.Redirect, назначение и константы, 527
- Process, назначение и методы, 520
- Properties
 - конструкторы, 646
 - методы, 646; 649
 - назначение, 645
 - применение, 647
- PropertyDescriptor
 - конструкторы, 1321
 - назначение и методы, 1319
- PushbackInputStream
 - конструкторы и методы, 740
 - назначение, 740
 - применение, 740
- PushbackReader
 - конструкторы и методы, 757
 - назначение, 757
- RadioButton
 - конструкторы, 1243
 - назначение, 1243
- RadioMenuItem, назначение, конструкторы и методы, 1294
- Random
 - конструкторы, 670
 - методы, 670
 - назначение, 670
- RandomAccessFile
 - конструкторы, 748
 - методы, 748
 - назначение, 748
- Reader, назначение и методы, 749
- RecursiveAction
 - назначение и методы, 1043
 - применение, 1047
- RecursiveTask
 - назначение и методы, 1043
 - применение, 1053
- ReentrantLock, назначение, 1037
- ReentrantReadWriteLock, назначение, 1039
- ResourceBundle
 - методы, 707
 - назначение, 706
 - подклассы, назначение, 709
- RGBImageFilter, применение, 992
- Rotate, назначение, конструкторы и методы, 1276
- Runtime
 - выполнение процессов, 524
 - назначение и методы, 521
 - управление памятью, 523
- RuntimeException, назначение, 264
- RuntimePermission, назначение, 555
- Scale, назначение, конструкторы и методы, 1276
- Scanner
 - конструкторы, 695
 - методы, 697; 705
 - назначение, 695
 - применение, 697; 700
- Scene
 - конструкторы, 1218
 - назначение, 1213
- Scrollbar
 - методы, 940
 - назначение и конструкторы, 940
- ScrollPane
 - конструкторы и методы, 1266
 - назначение, 1266
- SecurityManager, назначение, 555
- Semaphore
 - конструкторы и методы, 1009
 - назначение, 1008
- SequenceInputStream, назначение и конструкторы, 741
- ServerSocket
 - методы, 827
 - назначение и конструкторы, 827
- ServletInputStream, назначение, конструкторы и методы, 1333
- ServletOutputStream, назначение, конструкторы и методы, 1333
- SimpleBeanInfo, назначение, 1315
- SimpleDateFormat
 - применение, 1112
- SimpleDateFormat
 - конструкторы, 1111
 - назначение, 1111
 - шаблоны форматирования, 1111
- SimpleTimeZone, назначение и конструкторы, 667
- SingleSelectionModel, назначение и методы, 1157
- Socket
 - методы, 816
 - назначение и конструкторы, 816
 - применение, 817
- SocketAddress, назначение и реализация, 828
- Stack
 - методы, 639
 - назначение, 639
 - применение, 640
- StackTraceElement
 - методы, 556
 - назначение и конструкторы, 555
- Stage
 - главные подмости, 1213
 - назначение, 1213
- StrictMath, назначение и методы, 544
- String
 - конструкторы, 470

- методы, 201; 474; 487
- назначение, 200; 469
- StringBuffer
 - конструкторы, 471; 489
 - методы, 490; 495
 - назначение, 489
- StringBuilder
 - конструкторы, 471
 - назначение, 496
- StringTokenizer
 - конструкторы, 651
 - методы, 652
 - назначение и применение, 651
- SwingUtilities, назначение и методы, 1131
- Syntax
 - методы, 382; 528
 - назначение, 359
 - переменные потоков ввода-вывода, 359; 528
 - применение, примеры, 530
- TextArea
 - методы, 946
 - назначение и конструкторы, 945
- TextEvent, назначение, константы и конструкторы, 868
- TextField
 - конструкторы и методы, 1263
 - методы, 943
 - назначение, 1263
 - назначение и конструкторы, 943
- Text, назначение, 1279
- Thread
 - константы, 545
 - конструкторы, 545
 - методы, 289; 291; 297; 313; 546
 - назначение, 289; 545
 - применение, 289
 - расширение, 294
- ThreadGroup
 - методы, 548
 - назначение и конструкторы, 548
- ThreadLocal, назначение, 553
- Throwable
 - конструкторы, 281
 - методы, 278; 281
 - назначение, 264; 555
 - применение, 272
- Timer
 - конструкторы и методы, 677
 - назначение, 676
 - применение, 678
- TimerTask
 - конструкторы и методы, 676
 - назначение, 676
 - применение, 678
- TimeZone, назначение и методы, 666
- ToggleButton
 - конструкторы, 1241
 - методы, 1244
 - назначение, 1240
- ToggleGroup, назначение, 1244
- ToolBar, назначение, конструкторы и методы, 1300
- ToolTip, назначение и конструкторы, 1279
- Transform, назначение и подклассы, 1275
- TreeItem, назначение, 1269
- TreeMap
 - конструкторы, 611
 - назначение, 611
 - применение, 611
- TreePath, назначение и методы, 1169
- TreeSet
 - конструкторы, 587
 - назначение, 587
 - применение, 588
- TreeView
 - конструкторы, 1269
 - назначение, 1269
- URI, назначение, 826
- URL
 - конструкторы, 820
 - назначение, 820
- URLConnection
 - методы, 822
 - назначение, 821
- Vector
 - конструкторы, 635
 - назначение, 635
 - поля и методы, 636
 - применение, 638
- Void, назначение, 520
- WindowEvent
 - конструкторы и методы, 869
 - назначение и константы, 868
- Window, назначение, 890
- Writer, назначение и методы, 750
- абстрактные
 - объявление, 229
 - примечание, 229
- адаптеров
 - назначение, 880
 - применение, 880
 - разновидности, 880
- вложенные
 - назначение, 197
 - разновидности, 197
- внутренние
 - анонимные, назначение, 883
 - назначение, 197; 882
 - применение, 198; 883
- выходные файлы, порядок именования, 62
- идентификаторы, обозначение, 63
- иерархии, 57
- исключений
 - собственные, создание, 278
 - стандартные, 264
- исполнителей
 - назначение, 10 9
 - разновидности, 1007
- коллекций, разновидности, 579
- назначение, 55; 155
- наследуемые, объявление, 210

обобщенные
 иерархии, применение, 422
 объявление, общая форма, 404
 определение, 396
 переопределение методов, 428
 приведение типов, 428
 с двумя параметрами типа, 403
 создание, 396
 оболочек типов, назначение, 326
 общая форма, 155
 определение, 55
 отображений, разновидности, 609
 потоков ввода-вывода
 байтов, 357; 728
 символов, 358; 749
 синхронизаторов, разновидности, 1007
 событий, назначение, 858
 стандартные, назначение, 73
 члены, разновидности, 55; 156
 экземпляры как объекты, 55
Ключевые слова
 assert
 назначение, 385
 формы, 385
 class, назначение, 62; 156
 default, назначение, 257
 extends, назначение, 209
 final
 назначение, 194
 применение, способы, 231
 interface, назначение, 235
 native, назначение, 381
 private, назначение, 63
 public, назначение, 63
 static, назначение, 63; 193
 super, применение, 215; 218; 260
 this, назначение, 171
Коллекции в Java
 автоматическая упаковка и распаковка, 567
 алгоритмы
 применение, 628
 разновидности, 628
 генерируемые исключения, 568
 естественное упорядочение, 614
 изменяемые и неизменяемые, 568
 итераторы, назначение, 565
 каркас Collections Framework,
 назначение, 564
 обобщенные, 566
 параллельные
 классы, 1007; 1035
 назначение, 1035
 перебор
 в цикле for в стиле for each, 595
 итераторами, 593
 представление, назначение, 565
 произвольный доступ к элементам, 601
 синхронизированные, 627
 сохранение объектов разных классов, 599
 циклический перебор в стиле for each, 567

Комментарии
 документирующие
 назначение, 1347
 обозначение, 1347
 общая форма, 1352
 определение, 72
 применение, 1353
 многострочные, 62
 назначение, 62
 однострочные, 63
Компактные профили
 назначение, 393
 преимущества, 393
Компараторы
 назначение, 614
 применение, 617
 с естественным упорядочением, 615
 с обратным упорядочением, 615
Комплекты ресурсов
 назначение, 706
 обозначение, 706
 по умолчанию, применение, 706
 применение, 710
Компоненты
Java Beans
 архитектура, 1311
 индексированные свойства,
 назначение, 1314
 методы и шаблоны проектирования, 1315
 назначение, 1311
 настройки, применение, 1316
 обработка событий, 1314
 ограниченные свойства, назначение, 1316
 преимущества, 1312
 привязанные свойства, назначение, 1315
 применение, пример, 1320
 простые свойства, назначение, 1313
 самоанализ, механизм, 1312
 свойства, установка и получение, 1313
 сохраняемость, 1316
Swing
 действия, применение, 1199
 деревья типа JTree, создание
 и применение, 1168
 значки типа ImageIcon, создание
 и применение, 1144
 классы, 1143
 кнопки-переключатели типа JRadioButton,
 создание и применение, 1155
 комбинированные списки типа JComboBox,
 создание и применение, 1166
 метки типа JLabel, создание и применение,
 1143
 панели с вкладками типа JTabbedPane,
 создание и применение, 1157
 переключатели типа JToggleButton,
 создание и применение, 1150
 построение на основе AWT, 1125
 прокручиваемые панели типа JScrollPane,
 создание и применение, 1160

- списки типа `JList`, создание и применение, 1162
- таблицы типа `JTable`, создание и применение, 1172
- текстовые поля типа `TextField`, создание и применение, 1146
- флажки типа `CheckBox`, создание и применение, 1153
- экранные кнопки типа `Button`, создание и применение, 1148
- визуальные, составляющие, 1124
- легковесные, 974; 1123
- равноправные, 1122
- тяжеловесные, 974; 1122

Константы

- перечислимого типа, 318
- самотипизированные, 318

Конструкторы

- вызов по ссылке `this()`, 391
- инициализация объектов, 168
- назначение, 160
- обобщенные, применение, 417
- параметризованные, применение, 170
- перегрузка, 180
- порядок вызова, 222
- по умолчанию, применение, 160
- суперклассов, вызов, 215

Контейнеры

- `Swing`
 - панели, разновидности, 1126
 - разновидности, 1126
- сервлетов `Tomcat`
 - применение, 1326
 - состав, 1325
 - установка, 1326

- Круглые скобки, применение, 123

Л

Литералы

- двоичные, 83
- классов, 338
- логические, 85
- назначение, 71
- символьные, 85
- с плавающей точкой, формы записи, 84
- строковые, 86; 472
- целочисленные, 82

Лямбда-выражения

- блочные, 444
- внедрение, 50
- захват переменных, 452
- исключения, генерирование, 450
- контекст, разновидности, 440
- лямбда-оператор, назначение, 438
- назначение, 50
- одиночные, 444
- определение, 438
- основные принципы действия, примеры, 440
- передача в качестве аргументов, 447
- преимущества внедрения, 437
- тела, разновидности, 444

М

Массивы

- индексирование, 95
- инициализация, 96
- моды обработки, 629
- многомерные
 - итерация, 145
 - особенности, 97
- назначение, 94
- нерегулярные, 99
- объявление
 - альтернативная форма, 101
 - общая форма, 94
- одномерные, 94
- определение, 94
- параллельная сортировка, 632
- размер, 195
- создание, 95
- списочные
 - емкость, определение, 581
 - назначение, 581
 - получение обычного массива, 582
 - создание, 581

Меню

в `JavaFX`

- главное меню, построение
- изображения, ввод в пункты меню, 1293
- классы ядра системы, 1281
- контекстные, создание и активизация, 1296
- мнемоника, назначение, 1292
- обработка событий, 1283
- оперативные клавиши, назначение, 1292
- основные элементы, 1281
- отмечаемые пункты, применение, 1294
- порядок построения, 1282
- пункты меню, создание и выбор, 1285
- строки меню, построение, 1283
- флажки, применение в пунктах меню, 1294

в `Swing`

- всплывающие, создание и активизация, 1190
- главное меню, построение
- действия, применение, 1198
- изображения, ввод в пункты меню, 1187
- классы ядра системы, 1175
- кнопки-переключатели, применение в пунктах меню, 1189
- мнемоника, назначение в меню, 1185
- обработка событий, 1177
- оперативные клавиши, назначение в меню, 1185
- основные элементы, 1175
- отмечаемые пункты, применение, 1188
- подсказки всплывающие, ввод в пункты меню, 1187
- порядок построения, 1176
- пункты меню, создание и выбор, 1180
- строки меню, построение, 1178
- флажки, применение в пунктах меню, 1188

Методы

- clone(), применение, 533
- compareTo(), применение, 480
- equal () и операция ==, сравнение, 479
- finalize(), определение, 173
- main()
 - вызов, 63
 - определение, 63
 - параметр args, назначение, 64
- printf(), применение, 695; 744
- println(), назначение, 64
- print(), назначение, 66
- super(), назначение, 218
- toString(), реализация
 - и переопределение, 474
- valueOf(), применение, 485
- абстрактные
 - назначение, 229
 - объявление, 229
- аргументы, назначение, 167
- возвращаемые значения, особенности, 165
- вызовы
 - по значению, 184
 - по ссылке, 184
 - удаленный, механизм, 1105
- динамическая диспетчеризация, механизм, 225
- мастовые, применение
- обобщенные, применение, 414
- объявление, общая форма, 162
- определение, 55
- параметры, назначение, 64; 167
- перегрузка, 177
- переопределение
 - механизм, 223
 - назначение, 226
 - применение, 227
- платформенно-ориентированные
 - применение, 381
 - создание, 381
 - трудности применения, 384
- порядок
 - возврата значений, 162
 - вызова, 164
- по умолчанию
 - внедрение, 50; 256
 - назначение, 245
 - объявление, 257
 - применение, 258
- рекурсивные, применение, 187
- с аргументами переменной длины
 - перегрузка, 206
 - применение, 203
- синхронизированные, применение, 301
- статические
 - в классах, ограничения, 193
 - тело, обозначение, 64
- фабричные, назначение, 814
- Мнемоника, назначение, 1185; 1292
- Многозадачность
 - вытесняющая, 288

на основе

потоков, 285

процессов, 285

Многопоточность

особенности применения, 315

поддержка в Java, 286; 1005

преимущества, 286

принцип действия, 286

Модели

делегирования событий

преимущества, 856

применение, 874

принцип действия, 856

объектно-ориентированные, реализация, 54

ориентированные на процессы,

назначение, 53

потоков исполнения в Java, 287

циклов ожидания событий с опросом, 286

Модификаторы доступа

strictfp, применение, 380

synchronized, назначение, 301

transient, применение, 377

volatile, применение, 378

назначение, 63

разновидности, 190

Наследование

интерфейсов, 255

многоуровневые иерархии, построение, 219

множественное, поддержка, 259

назначение, 209

применение, 209

суперклассы и подклассы, 209

О

Области действия

вложенные, 88

назначение, 87

разновидности, 88

Обобщения

аргументы типа, назначение, 398

внедрение, преимущества, 395

выведение типов, 430

главное преимущество, 403

метасимвольные аргументы

ограниченные, 411

применение, 408

обеспечение типовой безопасности, 396; 401

определение, 396

ошибки неоднозначности, 433

параметры типа, назначение, 398

присущие ограничения, 434

реализация в Java, особенности, 431

стирание типов, механизм, 431

Оболочки типов

классы, 326

назначение, 326

примитивных, классы, 498

числовых, классы, 327

Объекты

возврат, 186

как экземпляры класса, 155

- клонирование, 533
- объявление, 159
- передача
 - в качестве параметров, 182
 - по ссылке
- сериализация и десериализация, 762
- синхронизации, применение, 1008
- Окна
 - в виде фреймов, 890
 - верхнего уровня, 890
 - иерархия классов, 888
 - обрамляющие
 - задание размеров, 891
 - закрытие, 891
 - обработка событий, 894
 - порядок создания, 891
 - сокрытие и отображение, 891
 - установка заголовка, 891
 - просмотра
 - задание размеров, 1266
 - назначение, 1266
 - определение, 1160
- Оперативные клавиши, назначение, 1185; 1292
- Операторы
 - catch, применение, 268
 - finally, назначение, 275
 - import static
 - основные формы, 390
 - применение, 388
 - import, назначение, 242
 - instanceof, назначение, 378
 - new, применение, 160
 - package, назначение, 236
 - synchronized, применение, 303
 - throws, назначение, 273
 - throw, применение, 272
 - try
 - вложенные, применение, 270
 - с ресурсами, применение, 371; 727
- ветвления, switch
 - вложенные, 132
 - назначение, 128
 - особенности, 133
 - применение, 129
- перехода
 - break
 - применение, 148
 - с меткой, применение, 150
 - continue, применение, 152
 - return, применение, 153
- пустые, назначение, 134
- ромбовидные, назначение, 430
- управляющие
 - категории, 125
 - назначение, 125
- условные, if
 - вложенные, 127
 - конструкция if-else-if, применение, 127
 - назначение, 125
 - применение, 67
- простейшая форма, 67
- цикла
 - do-while, применение, 135
 - for
 - вложенные, 147
 - простейшая форма, 68
 - разновидности, 140
 - традиционная форма, 138
 - форма в стиле for each, 142
 - while, применение, 133
- Операции
 - арифметические
 - деления по модулю, 105
 - инкремента и декремента, 69; 106
 - основные, 104
 - применение, 103
 - составные с присваиванием, 105
 - логические
 - применение, 119
 - разновидности, 118
 - укороченные, применение, 120
 - отношения
 - применение, 117
 - разновидности, 117
 - поразрядные
 - логические, разновидности, 109
 - применение, 108
 - составные с присваиванием, 116
 - потокковые
 - без сохранения состояния, 1065
 - изменяемого сведения, 1083
 - накопления, ограничения, 1071
 - оконечные и промежуточные, отличие, 1064
 - особого сведения, 1070
 - с сохранением состояния, 1065
 - присваивания
 - обозначение, 120
 - объединение в цепочки, 121
 - сдвига
 - беззнакового вправо, 115
 - влево, 111
 - вправо, 113
 - сравнения, 67
 - 'стрелки', назначение, 438
 - сцепления символьных строк, 473
 - тернарные, применение, 121
 - точки, применение, 157
- Отображения
 - древовидные
 - применение, 611
 - создание, 611
 - ключи и значения, назначение, 601
 - манипулирование подотображениями, 605
 - назначение, 601
 - основные операции, 604
 - отсортированные, 605
 - представления
 - назначение, 605
- Очереди
 - двусторонние

- ввод элементов, 590
- создание, 590
- по приоритетам
 - компараторы, применение, 589
 - создание, 589
 - сортировка элементов, 589

П

Пакеты

- java.awt.event, классы и интерфейсы
 - событий, 858, 859; 880
- java.awt.image, классы и интерфейсы, 975
- java.awt, классы, 886
- java.beans, интерфейсы и классы, 1315–1317
- javafx.application, состав, 1291
- javafx.collections, состав, 1255
- javafx.event, интерфейсы и классы
 - событий, 1222
- javafx.scene.control, состав, 1220–1223; 1233; 1244; 1281
- javafx.scene.effect, состав, 1274
- javafx.scene.image, состав, 1233
- javafx.scene.input, состав, 1292
- javafx.scene.layout, панели компоновки, 1213
- javafx.scene.paint, состав, 1229
- javafx.scene.shape, состав, 1232
- javafx.scene.text, состав, 1229; 1279
- javafx.scene.transform, состав, 1275
- java.io
 - классы и интерфейсы, 718
 - назначение, 356; 717
- java.lang
 - интерфейсы, 497
 - исключения, классы, 276
 - классы, 497
 - основные языковые средства, 243
 - подпакеты, назначение, 561
 - символьные строки, классы, 469
- java.lang.annotation, аннотации, 334
- java.lang.reflect
 - интерфейсы и классы, 1101
 - рефлексия, 336; 1101
- java.net
 - классы и интерфейсы, 813; 826
 - назначение, 811
- java.nio.channels
 - каналы, 772
 - селекторы, 774
- java.nio.charset, наборы символов, кодеры и декодеры, 774
- java.nio.file.attribute, интерфейсы атрибутов файлов, 780
- java.nio.file, классы, 782
- java.nio, буферы, 770
- java.rmi, интерфейсы и классы, 1106
- java.scene.canvas, состав, 1227
- java.text, классы, 1109
- java.time
 - другие классы даты и времени, 1118
 - основные классы даты и времени, 1113
- java.time.forma , состав, 1115

- java.util
 - каркас коллекций Collections Framework, 563
 - классы и интерфейсы, 563; 651; 710
 - подпакеты, назначение, 711
 - унаследованные классы и интерфейсы, 634
- java.util.concurrent
 - каркас Fork/Join Framework, 1040
 - утилиты параллелизма, 1006
- java.util.concurrent.atomic
 - атомарные операции, средства, 1039
 - состав, 1008
- java.util.concurrent.locks, состав, 1008
- java.util.function, функциональные интерфейсы, 465; 712; 1069
- java.util.regex, обработка регулярных выражений, 1092
- java.util.stream, потоковые интерфейсы и классы, 1062; 1080
- javax.imageio, назначение, 1004
- javax.servlet
 - интерфейсы и классы, 1328; 1329
 - классы исключений, 1333
- javax.servlet.http, интерфейсы и классы, 1335
- javax.swing
 - классы компонентов, 1125; 1143
- javax.swing.event, классы и интерфейсы
 - событий, 1132; 1163; 1168
- javax.swing.table, интерфейсы и классы, 1172
- javax.swing.tree, интерфейсы и классы, 1168
- базового API, разновидности, 1089
- библиотеки Swing, разновидности, 1127
- доступность членов классов, категории, 238
- импорт, 242
- каркаса JavaFX, состав, 1212
- многоуровневые, создание, 236
- назначение, 235
- нового API даты и времени, разновидности, 1113
- применение, 235
- системы ввода-вывода NIO, 769

Панели инструментов

- в JavaFX
 - всплывающие подсказки, применение, 1300
 - построение, 1300
- в Swing
 - действия, применение, 1198
 - построение, 1194
 - применение, 1194
 - назначение, 1194
 - отстыкованные, перетаскивание, 1194
 - применение, 1300

Переменные

- в интерфейсах, применение, 253
- действительно конечные, определение, 452
- назначение, 86
- область действия, 89
- объявление, 65; 86
- управления циклом
 - назначение, 138

- объявление, 138
- экземпляра
 - length, применение, 195
 - назначение, 156
 - определение, 55
 - присваивание значений, 158
 - сокращение, 172
- Переход через нуль, 109
- Перечисления
 - ContentDisplay, назначение и константы, 1237
 - Element Type, константы, 345
 - FileVisitResult, константы, 802
 - FormatStyle, назначение и константы, 1115
 - Pos, назначение и константы, 1227
 - StandardOpenOption, константы, 779
 - TimeUnit
 - методы синхронизации, 1035
 - назначение и константы, 1034
 - встроенные методы, применение, 319
 - накладываемые ограничения, 323
 - определение, 317
 - порядковые значения констант, 323
 - реализация в виде классов, 317
 - создание, 317
 - типа классов, возможности, 321
- Подсказки всплывающие, применение, 1187; 1279; 1301
- Полное завершение, механизм, 173
- Потоки
 - ввода-вывода
 - байтов, назначение, 356; 728
 - буферизованные, назначение, 737
 - закрывание, способы, 726
 - определение, 356
 - предопределенные, 359
 - преимущества, 768
 - реализация, 356
 - символов, назначение, 356; 749
 - фильтруемые, назначение, 737
 - данных
 - накопление, 1079
 - особенности, 1061
 - отложенное поведение, механизм, 1064
 - отображение, 1075
 - параллельные, применение, 1073
 - получение, способы, 1065
 - применение, 1066
 - упорядоченность, 1075
 - диспетчеризации событий
 - взаимодействия с компонентами
 - Swing, 1136
 - вызов обработчиков событий, 1135
 - назначение, 1131
 - построение ГПИ, 1131
 - исполнения
 - взаимодействие, 304
 - выбор способа создания, 295
 - главные, назначение, 290
 - группы, назначение, 550
 - завершение, способы определения, 297
 - запускающие, назначение, 1219
 - ложная активизация, 305
 - обмен сообщениями, 289
 - общий пул, назначение, 1044
 - определение, 285
 - переключение контекста, правила, 288
 - получение текущего состояния, 313
 - приложений, назначение, 1219
 - приоритеты, 287; 299
 - приостановка, возобновление
 - и остановка, 311
 - синхронизация, 288; 300
 - создание, 289; 291
 - состояния, 287
 - явно задаваемый пул, назначение, 1029
 - Преобразование типов
 - автоматическое, 90
 - расширяющее, 90
 - сужающее, 91
 - усечение, 91
 - Приведение типов
 - назначение, 90
 - общая форма, 91
 - Пробелы и отступы, назначение, 71
 - Программирование
 - для Интернета, на Java, 38
 - многопоточное
 - потоки исполнения, 285
 - процессы, 285
 - неструктурное, на BASIC, COBOL, FORTRAN и ассемблере, 34
 - объектно-ориентированное
 - абстракция, применение, 54
 - инкапсуляция, принцип, 55
 - на C++, 36
 - назначение, 53
 - на ведение, принцип, 56
 - определение, 36
 - полиморфизм, принцип, 59
 - принципы, совместное применение, 59
 - параллельное
 - задачи, решаемые в Fork/Join Framework, 50
 - назначение, 287
 - определение, 49; 1040
 - поддержка в Fork/Join Framework, 1040
 - структурное, на C, 35
 - этапы развития, 36
 - Продвижение типов
 - автоматическое, 93
 - правила, 93

Р

 - Работа в сети
 - адреса Интернета
 - назначение, 812
 - поиск, 815
 - дейтаграммы, назначение, 828
 - доменные имена, назначение службой DNS, 812
 - порты

- определение, 811
- разновидности, 812
- сокеты, назначение, 811
- Разделители**
 - лексем, применение, 704
 - наборы символов, 651
 - назначение, 651
 - символьное обозначение, 72
- Распаковка, назначение, 328**
- Расширение знака, 114**
- Регулярные выражения**
 - входная последовательность, 1092
- кванторы**
 - обозначение, 1094
 - применение, 1096
- классы символов**
 - задание, 1094
 - применение, 1098
- метасимволы**
 - обозначение, 1094
 - применение, 1097
- обработка, 1092**
- определение, 1092**
- синтаксис, 1094**
- совпадение с шаблоном, строгое и нестрогое, 1097**
- сопоставление с шаблоном, 1094**
- шаблоны, составление, 1092**
- Рекурсия**
 - определение, 187
 - организация, 188
- Рефлексия**
 - назначение, 336
 - определение, 1101
 - применение, способы, 336
 - свойства, применение, 1101
- Рисование**
 - дуг, 901
 - линий, 900; 1228
 - многоугольников, 901
 - прямоугольников, 900; 1228
 - режим, установка, 907
 - средствами
 - AWT, особенности, 1138
 - JavaFX, особенности, 1227
 - Swing, особенности, 1139
 - установка цвета, 906
 - эллипсов и окружностей, 901; 1229

С

- Сборка "мусора", процесс, 172**
- Связные списки**
 - ввод и удаление элементов, 584
 - создание, 584
- Связывание**
 - позднее, 232
 - раннее, 232
- Сеансы связи**
 - назначение, 1346
 - отслеживание состояния, 1346
 - создание, 1346

- Сервлеты**
 - ввод параметров, 1334
 - жизненный цикл, 1324
 - назначение, 43
 - обработка HTTP-запросов типа
 - GET, 1341
 - POST, 1342
 - определение, 1323
 - отслеживание сеансов связи, 1346
 - переносимость, 43
 - порядок
 - проверки, 1335; 1342; 1345
 - создания, 1327
 - преимущества, 1324
 - разработка, варианты, 1325
- Сериализация**
 - и десериализация объектов, применение, 767
 - определение, 761
 - при удаленном вызове методов, применение, 762
- Сетевые протоколы**
 - HTTP, назначение, 812
 - IPv4 и IPv6, назначение, 812
 - IP, назначение, 811
 - JNLP, назначение, 834
 - TCP, назначение, 812
 - UDP, назначение, 812
- Символы**
 - дополнительные
 - определение, 517
 - представление в виде сурроганой пары, 517
 - зеркально отображаемые, определение, 516
 - кодовая точка, определение
 - направленность, определение, 515
 - основная многоязыковая плоскость, 517
- Символьные строки**
 - длина, получение, 472
 - замена, 484
 - извлечение
 - подстрок, способы, 483
 - символов, способы, 475
 - изменение регистра символов, 486
 - как объекты класса String, 101
 - неизменяемость, 200; 469
 - обрезка, 484
 - поиск подстрок, 481
 - преобразование
 - в числа, взаимное, 513
 - данных, 485
 - реализация в Java, 469
 - соединение, 487
 - создание, способы, 200
 - сравнение, 477; 480
 - средства обработки в Java, 469
 - сцепление, 473; 484
- Синтаксический анализ**
 - назначение, 651
 - разбиение на лексемы, 651
 - строк даты и времени, 1117

Синхронизация
 взаимноисключающая блокировка потоков
 исполнения, 300
 механизм мониторов, 288; 300
 потоков исполнения, определение, 300
 степень разрешения, обозначение, 1034

Словари
 ввод и удаление ключей и значений, 641
 создание, 641

События

ввода данных
 в компонентах, разновидности, 863
 в текстовых полях, обработка, 944
 в компонентах
 деревьях типа JTree, обработка, 1168
 кнопках-переключателях типа JRadioButton, обработка, 1155
 комбинированных списках типа JComboBox, обработка, 1166
 переключателях типа JToggleButton, обработка, 1151
 разновидности, 861
 списках типа JList, обработка, 1163
 таблицах типа JTable, обработка, 1172
 текстовых полях типа JTextField, обработка, 1146
 флажках типа JCheckBox, обработка, 1153
 экранных кнопках типа JButton, обработка, 1148
 в контейнере, разновидности, 861
 в меню, обработка, 964; 1177; 1283
 в раскрывающихся списках, обработка, 935
 всплывание, механизм, 1223
 групповая рассылка, 857
 действия
 временная метка, 860
 разновидности, 859
 из библиотеки
 AWT, 858
 JavaFX, 1222
 Swing, 1132
 изменения в группе кнопок-переключателей, обработка, 1247
 источники
 определение, 857
 регистрация приемников, 857
 настройки, разновидности, 860
 обработка, способы, 856
 оконные, разновидности, 868
 определение, 856
 от выбираемых элементов, разновидности, 864
 от клавиатуры
 обработка, 877
 разновидности, 864
 от кнопок, обработка, 927
 от колесика мыши, 867
 от мыши
 обработка, 874
 разновидности, 865
 от полос прокрутки, обработка, 941

от списков, обработка, 938
 от флажков, обработка, 931
 приемники
 определение, 857
 регистрация и снятие с регистрации, 857
 строка с командой действия
 назначение, 927
 получение, 1148
 текстовые, разновидности, 868
 типы, 855
 фокуса ввода, разновидности, 862
 целевая рассылка, 857

Сокеты

определение, 811
 по протоколу TCP/IP
 клиентские, 816
 серверные, 827
 связь по сетевым протоколам, 811

Состояние гонок

возникновение, 302
 исключение, 303

Спецификаторы

минимальной ширины поля,
 применение, 687
 преобразования, обозначение, 682
 точности, применение, 689
 формата
 буквы шаблона, 1116
 определение, 682
 прописные формы, применение, 692
 разновидности, 682

Ссылки

на конструкторы
 применение, 460
 формы создания, 460; 464
 на методы
 назначение, 453
 обобщенные, применение, 458
 применение в коллекциях, 459
 статические, применение, 453
 экземпляра, применение, 454
 на объекты
 особенности, 160
 присваивание, 161

Статический импорт

назначение, 388
 применение, 389

Стек

ввод и удаление элементов, 640
 принцип действия, 639

Стековый фрейм, назначение, 555

Стирание типов, принцип действия, 431

Т

Текст

выводимый в окне
 форматирование
 центровка, 917
 многострочный
 выравнивание, 918
 отображение, 916

Типы данных

- базовые, назначение, 420
- обобщенные, различение по аргументам, 400
- ограниченные, назначение, 405
- примитивные
 - логические значения, 81
 - применение, 76
 - разновидности, 75
 - символы, 80
 - целые числа, 76
 - числа с плавающей точкой, 78
- строковые, назначение, 101

Традиционное управление ресурсами, особенности, 374

Трассировка стека, назначение, 266

У

Удаленный вызов методов

- заглушки
 - назначение, 1107
 - создание, 1107
- запуск
 - клиента, 1108
 - реестра, 1108
 - сервера, 1108
- механизм, 1105

Универсальное скоординированное время, определение, 666

Упаковка, назначение, 328

Управление доступом, механизм

Управляющие последовательности символов, 85

Утверждения

- включение и отключение режима
 - проверки, 388
 - назначение, 385
 - применение, 385

Утилиты параллелизма

- атомарные операции, разновидности, 1039
- барьеры, механизм, 1016

- блокировки
 - механизм, 1036
 - реентерабельные, реализация, 1037

- исполнители
 - механизм, 1028
 - назначение, 1007
 - применение, 1029

и традиционный подход к многозадачности, 1060

назначение, 1005

обмен данными, механизм, 1018

применение, 1006

самоблокировка с обратным отсчетом, механизм, 1014

- семафоры
 - механизм, 1008
 - применение, 1009

синхронизаторы

- назначение, 1007

фаз, механизм, 1021

Ф

Файлы

атрибуты

- доступ, способы, 781
- назначение, 780

доступ, назначение, 721

заккрытие

- автоматическое, 371
- традиционным способом, 365

запись данных через канал

- новые способы, 789
- старые способы, 807

копирование средствами NIO, 792

назначение, 719

открытие для ввода, 365

переименование, 721

сведения, получение, 796

создание, 719

удаление, 721

фильтрация, 723

чтение данных через канал

- новые способы, 783
- старые способы, 804

Форматирование

выводимого текста, 915

выравнивание выводимых данных, 690

данных, механизм, 682

даты и времени, 685; 1109–1115

индексы аргументов, применение, 693

относительные индексы, применение, 693

признаки

- прочие, применение, 691; 692
- формата, обозначение, 689

строк и символов, 684

чисел, 684

Функциональные интерфейсы

BiConsumer, назначение, 1082

BinaryOperator, назначение, 1070

Consumer, назначение, 597; 1069

Function, назначение, 1076

Predicate, назначение, 1069

внедрение, 50

обобщенные, назначение, 446

объявление, 439

определение, 438

предопределенные, разновидности, 465

с единственным абстрактным методом, 438

Х

Хеширование

механизм, 586

преимущество, 586

Хеш-множества

емкость загрузки, определение, 586

порядок сохранения элементов, 586

связные списки, составление, 587

создание, 586

Хеш-отображения

емкость, определение, 609

расположение элементов, 610

связные

создание, 613
создание, 609
Хеш-таблицы
применение, 585
создание, 643
хранение ключей и значений, 642

Ч

Чтение
данных средствами класса Scanner,
механизм, 696
символов из потока ввода, 360
символьных строк с клавиатуры, 361

Ш

Шрифты
доступные, определение, 910
логическое имя, 909
наименование
гарнитуры, 909
семейства, 909
описание, терминология, 914
сведения, получение, 913
создание и выбор, 911

Э

Элементы управления
ввод и удаление, 924
из библиотеки AWT, разновидности, 924
из каркаса JavaFX, разновидности, 1233
обработка событий, 925
определение, 923

Я

Язык Java
архитектурная нейтральность, 45

байт-код
интерпретация, 42
назначение
безопасность, 41
виртуальная машина Java, назначение, 41
внешнее сходство с языком C++, 39
высокая производительность, 46
динамический компилятор, назначение, 42
динамичность, 46
единица компиляции, определение, 61
интерпретируемость, 46
история создания, 37
культура нововведений, 51
многопоточность, 45
наджность, 44
назначение, 28
нововведения в версии
J2SE 5, 47
Java SE 7, 49
Java SE 8, 50
новые языковые средства, 28
переносимость, 41
порядок компиляции программ, 61
предпосылки для создания, 37
причины
разработки, 33
успеха, 28
происхождение, 33
простота, 44
распределенность, 46
связь с языком C#, 39
строго типизированный, 75
терминология, 43
эволюция, 46

Исчерпывающее руководство по программированию на Java



В этом справочном пособии, полностью обновленном с учетом последней версии Java SE 8, поясняется, как разрабатывать, компилировать, отлаживать и выполнять программы на языке программирования Java. Это пособие составлено Гербертом Шилдтом, автором популярных во всем мире книг по языкам программирования, таким образом, чтобы охватить все языковые средства Java, включая синтаксис, ключевые слова, основные принципы объектно-ориентированного программирования, значительную часть прикладного программного интерфейса Java API, библиотеки классов, апплеты и сервлеты, компоненты JavaBeans, библиотеки AWT и Swing, а также продемонстрировать их применение на простых и наглядных примерах. Не обойдены вниманием и новые средства, появившиеся в версии Java SE 8, в том числе лямбда-выражения, стандартные интерфейсные методы, библиотека потоков ввода-вывода, а также технология JavaFX.

В этой книге рассматриваются следующие вопросы:

- Типы данных, переменные, массивы и операции
- Управляющие и условные операторы
- Классы, объекты и методы
- Перегрузка и переопределение методов
- Наследование
- Интерфейсы и пакеты
- Обработка исключений
- Многопоточное программирование
- Перечисления, автоупаковка и автораспаковка
- Потоки ввода-вывода
- Обобщения
- Лямбда-выражения
- Обработка символьных строк
- Каркас коллекций Collection Framework
- Работа в сети
- Обработка событий
- Библиотеки AWT и Swing
- Прикладной программный интерфейс Concurrent API
- Прикладной программный интерфейс Stream API
- Регулярные выражения
- Технология JavaFX
- Компоненты JavaBeans
- Апплеты и сервлеты

Категория: Программирование

Предмет рассмотрения: Язык Java

Уровень: Начальный/промежуточный

Oracle
Press

ISBN 978-5-8459-1918-2



Издательский дом "Вильямс"
<http://www.williamspublishing.com>



McGraw Hill
MHPROFESSIONAL.COM

