Полный справочник по Java™

Java SE™ 6 Edition

7-е издание

Java™: The Complete Reference

Java SE™ 6 Edition

Seventh Edition

Herbert Schildt

McGraw-Hill/Osborne

New York, Chicago, San Francisco
Lisbon, London, Madrid, Mexico City
Milan, New Delhi, San Juan
Seoul, Singapore, Sydney Toronto

Полный справочник по Java™

Java SE™ 6 Edition

7-е издание

Герберт Шилдт

Издательский дом “Вильямс”
Москва ♦ Санкт-Петербург ♦ Киев
2009

ББК 32.973.26-018.2.75
Ш57
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского *Д.Я. Иваненко, Ю.И. Корниенко, Н.А. Мухина*

Под редакцией *Ю.Н. Артеменко*

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

Шилдт, Герберт.

Ш57 Полный справочник по Java, 7-е издание. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2009. — 1040 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1168-1 (рус.)

Книга известного гуру в области программирования посвящена новой версии одного из наиболее популярных и совершенных языков — Java. Построенная в виде учебного и справочного пособия, она является превосходным источником исчерпывающей информации по последней версии платформы Java, Java SE 6, и позволяет практически с нуля научиться разрабатывать приложения и аплеты производственного качества. Помимо синтаксиса самого языка и фундаментальных принципов программирования, в книге подробно рассматриваются такие сложные вопросы, как ключевые библиотеки Java API, каркас коллекций, создание аплетов и сервлетов, AWT, Swing и Java Beans. Немалое внимание уделяется вводу-выводу, работе в сети, регулярным выражениям и обработке строк. Изобилие реальных примеров, доступных также и на Web-сайте издательства, существенно упрощает усвоение материала.

Книга ориентирована на программистов и разработчиков различной квалификации, а также будет полезна студентам и преподавателям соответствующих специальностей.

ББК 32.973.26-018.2.75

ISBN 978-5-8459-1168-1 (рус.)
ISBN 0-07-226385-7 (англ.)

Оглавление

Содержание

Об авторе

Герберт Шилдт (Herbert Schildt) — известный во всем мире автор множества бестселлеров, посвященных программированию на языках Java, C, C++ и C#. Популярность его книг доказана продажей свыше 3,5 миллионов экземпляров и переводом их на большинство языков. Среди успешных книг Герберта: *The Art of Java*, *Java: A Beginner's Guide*, *Swing: A Beginner's Guide*, *C++: The Complete Reference*, *C++: A Beginner's Guide*, *C#: The Complete Reference* и *C#: A Beginner's Guide*. Он получил диплом о высшем образовании и ученую степень получил в университете Иллинойса. Дополнительную информацию об авторе можно получить на его Web-сайте по адресу `www.HerbSchildt.com`.

Предисловие

Когда я приступил к написанию этой книги, история развития языка Java лишь только перевалила во второе десятилетие. В отличие от многих других языков программирования, влияние которых с годами начинает ослабевать, Java со временем только крепнет. С момента своего появления язык Java оказался на переднем рубеже программирования для Internet. Каждая последующая версия еще больше укрепляла его позиции. На сегодняшний день Java остается основным и лучшим средством разработки Web-приложений.

Одна из причин успеха Java — его быстрая изменчивость. Этот язык быстро адаптируется к изменениям в среде программирования и в подходах к программированию. И самое главное — он не просто следует тенденциям, *а помогает их создавать*. В отличие от некоторых других языков, цикл пересмотра которых составляет около 10 лет, новые версии Java появляются в среднем через каждые полтора года. Способность приспособления Java к высокой скорости изменений в компьютерном мире — основная составляющая причины того, что он остается на передовых рубежах проектирования компьютерных языков. С выходом версии Java SE 6 ведущие позиции Java остаются непоколебимыми. Если вы создаете программы для Internet, выбор этого языка совершенно правилен. Java был и продолжает оставаться наиболее предпочтительным языком программирования для Internet.

Как известно многим читателям, это — седьмое издание данной книги, которая впервые была опубликована в 1996 г. Настоящее издание полностью учитывает Java SE 6. Кроме того, в книгу включено несколько важных тем. Теперь книга содержит вдвое больше материала по Swing и в ней подробнее рассмотрены пакеты ресурсов. Книга содержит множество других добавлений и уточнений. Одним словом, в нее добавлены десятки страниц с новым материалом.

Книга для всех программистов

Эта книга предназначена для всех программистов — как новичков, так и для опытных профессионалов. Начинающий программист найдет в ней подробные пошаговые описания и множество чрезвычайно полезных примеров. А углубленное рассмотрение более сложных функций и библиотек Java должно удовлетворить ожидания профессиональных программистов. Для обеих категорий читателей в книге приведены указания действующих ресурсов и полезных ссылок.

Что внутри

Эта книга представляет собой всеобъемлющее руководство по языку Java, описывающее его синтаксис, ключевые слова и фундаментальные принципы программирования. В ней рассмотрена также значительная часть библиотеки Java API. Книга разделена на

четыре части, каждая из которых посвящена отдельному аспекту среды программирования Java.

Часть I представляет собой подробный учебник по языку Java. Она начинается с рассмотрения основных понятий, таких как типы данных, управляющие операторы и классы. В ней рассмотрены также механизм обработки исключений Java, подсистема многопоточной обработки, пакеты и интерфейсы. Естественно, также подробно описаны новые свойства Java, такие как обобщения, аннотации, перечисления и средства автоматической упаковки.

В части II рассмотрены основные аспекты стандартной библиотеки интерфейса прикладного программирования Java. В ней раскрыты такие темы как строки, ввод-вывод, сетевая обработка, стандартные утилиты, каркас коллекций Collections Framework, аплеты, элементы управления графического интерфейса пользователя (GUI), средства работы с изображениями и параллельной обработки.

В части III рассмотрены три важных технологии Java: Java Beans, Swing и сервлеты.

Часть IV содержит две главы с примерами реального использования Java. Первая глава посвящена разработке нескольких аплетов, которые выполняют популярные финансовые вычисления, такие как вычисление выплаты процентов по ссуде или размера минимального вклада необходимого для получения желаемого ежемесячного дохода. В этой главе также показано, как преобразовать эти аплеты в сервлеты. Другая глава этой части посвящена разработке утилиты загрузки, управляющей загрузкой файлов. К числу выполняемых им функций относятся запуск, останов и возобновление передачи данных. Обе главы заимствованы из моей книги *The Art of Java* (*Искусство программирования на Java*. И.Д. "Вильямс", 2005), которую я написал в соавторстве с Джеймсом Холмсом (James Holmes).

Коды примеров доступны в Web

Помните что исходные коды всех примеров, приведенных в этой книге, доступны на Web-сайте издательства.

Особые благодарности

Выражаю свою особую благодарность Патрику Нотону (Patrick Naughton). Он был одним из создателей языка Java. Он также помог мне в написании первого издания этой книги. Например, значительная часть материала глав 19, 20 и 25 была предоставлена Патриком. Его проницательность, опыт и энергия в огромной степени способствовали успеху этой книги.

Я благодарю также Джо О'Нила (Joe O'Neil), который предоставил исходные черновые материалы для глав 27, 28, 30 и 31. Джо помогал мне при написании нескольких книг и, как всегда, я высоко ценю его вклад.

И, наконец, я горячо благодарю Джеймса Холмса (James Holms) за подготовку материалов главы 32. Джеймс — выдающийся программист и автор. Он был моим соавтором книги *The Art of Java*, является автором книги *Struts: The Complete Reference* и соавтором книги *JSF: The Complete Reference*.

Герберт Шилдт
8 ноября 2006 г.

Для дальнейшего изучения

Эта книга открывает серию книг по программированию, написанных Гербертом Шилдтом. Ниже перечислены другие книги этого автора, которые, несомненно, вас заинтересуют.

Если вы хотите больше узнать о программировании на Java, рекомендуем прочесть следующие книги:

- *Java: руководство для начинающих*. И.Д. "Вильямс". Плановая дата выхода — 3 кв. 2007 года.
- *Swing: руководство для начинающих*. И.Д. "Вильямс".
- *Искусство программирования на Java*. И.Д. "Вильямс".

Тем, кто желает изучить язык C++, особенно полезными будут следующие книги:

- *Полный справочник по C++*. И.Д. "Вильямс".
- *C++: руководство для начинающих*. И.Д. "Вильямс".
- *The Art of C++*. McGraw-Hill/Osborne.
- *C++: базовый курс*. И.Д. "Вильямс".
- *STL Programming From the Ground Up*. McGraw-Hill/Osborne.

Для изучения C# рекомендуем следующие книги Шилдта:

- *Полный справочник по C#*. И.Д. "Вильямс".
- *C#: A Beginner's Guide*. McGraw-Hill/Osborne.

Перечисленные ниже книги помогут в изучении языка C:

- *Полный справочник по C*. И.Д. "Вильямс".
- *Teach Yourself C*. McGraw-Hill/Osborne.

Если вам нужно быстро получить исчерпывающие ответы, обратитесь к книгам Герберта Шилдта — признанного во всем мире авторитета в области программирования.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш Web-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг.

Наши координаты:

E-mail: `info@williamspublishing.com`
WWW: `http://www.williamspublishing.com`

Информация для писем из:

России: 127055, Москва, ул. Лесная, д. 43, стр. 1
Украины: 03150, Киев, а/я 152

I ЧАСТЬ

Язык Java

1

ГЛАВА

История и развитие языка Java

Чтобы досконально изучить язык Java, необходимо понять причины, которые привели к его созданию, факторы, повлиявшие на конечную архитектуру, и унаследованные им особенности. Подобно удачным компьютерным языкам, появившимся до него, Java объединяет в себе лучшие элементы из своего богатого наследия и новаторские концепции, применение которых обусловлено его уникальным положением. В то время как остальные главы этой книги посвящены практическим аспектам Java — в том числе его синтаксису, библиотекам и приложениям — в настоящей главе описано, как и почему был разработан этот язык, что делает его столь важным, и как он развился за годы своего существования.

Хотя язык Java неразрывно связан с сетевой средой Internet, важно помнить, что, прежде всего, это язык программирования. Модернизация и разработка компьютерных языков обусловлены двумя основными причинами:

- необходимостью адаптации к изменяющимся средам и областям применения;
- необходимостью реализации 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 языка Java

Язык Java тесно связан с языком C++, который, в свою очередь, является прямым наследником языка C. Многие из особенностей 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 Java был перенесен — или разработан в ответ на возникшие потребности — из его предшественников. Более того, создание Java своими корнями глубоко уходит в процесс усовершенствования и адаптации, который происходит в языках компьютерного программирования на протяжении нескольких последних десятилетий. Поэтому в данном разделе мы рассмотрим последовательность событий и факторы, которые привели к появлению

языка Java. Как вы убедитесь, каждое новшество в архитектуре языка было обусловлено необходимостью решения той или иной фундаментальной проблемы, которую не могли решить существовавшие до этого языки. И Java — не исключение в этом отношении.

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 компьютерного языка часто приходится находить компромиссы вроде следующих:

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

До появления языка C программистам, как правило, приходилось выбирать между языками, которые позволяли оптимизировать тот или иной набор характеристик. Например, хотя FORTRAN можно было использовать для написания достаточно эффективных программ для научных приложений, он не слишком подходил для создания системного кода. Аналогично, в то время как язык BASIC был очень прост в изучении, он предоставлял не очень-то много функциональных возможностей, а его недостаточная структурированность ставила под сомнение его полезность при создании крупных программ. Язык ассемблера можно использовать для создания очень эффективных программ, но его трудно изучать или эффективно использовать. Более того, отладка ассемблерного кода может оказаться весьма сложной задачей.

Еще одной сыгравшей свою роль проблемой была та, что ранние языки программирования, такие как BASIC, COBOL и FORTRAN, были спроектированы без учета принципов структурирования. Вместо этого в них основными средствами управления программой были операторы безусловного перехода GOTO. В результате программы, созданные с применением этих языков, тяготели к созданию так называемого "макаронного кода" — множества запутанных переходов и условных ветвей, которые делали программу буквально недоступной для понимания. Хотя языки вроде Pascal и структурированы, они не были предназначены для создания максимально эффективных программ и были лишены ряда важных функций, необходимых для применения этих языков к написанию широкого круга программ. (В частности, учитывая существование нескольких стандартных диалектов Pascal, было нецелесообразно применять этот язык для создания кода системного уровня.)

Таким образом, непосредственно накануне изобретения языка C, несмотря на затраченные усилия, ни одному языку не удалось решить существующие конфликты. Вместе с тем потребность в таком языке становилась все более насущной. В начале 70-х гг. началась компьютерная революция, и потребность в программном обеспечении быстро превысила возможности программистов по его созданию. В академических кругах большие усилия были приложены к созданию более совершенного языка программирования. Однако, и это наиболее важно, все больше стало ощущаться влияние второго фактора. Компьютеры, наконец, получили достаточно широкое распространение, чтобы была достигнута "критическая масса". Компьютеры больше не стояли за запертыми дверями. Впервые программисты получили буквально неограниченный доступ к своим компьютерам. Это дало свободу экспериментам. Программисты смогли также приступить к созданию своих соб-

ственных программных средств. Накануне создания языка C произошел качественный скачок в области компьютерных языков.

Изобретенный и впервые реализованный Деннисом Ритчи на компьютере DEC PDP-11, работающем под управлением операционной системы UNIX, язык C явился результатом процесса разработки, начавшегося с предшествующего языка BCPL, разработанного Мартином Ричардсом. BCPL повлиял на язык, получивший название B, который был изобретен Кеном Томпсоном, и который в начале 70-х гг. привел к появлению языка C. В течение долгих лет фактическим стандартом языка C была его версия, поставлявшаяся с операционной системой UNIX и описанная в книге *Язык программирования C*, написанной Брайаном Керниганом и Деннисом Ритчи (2-е издание, ИД "Вилямс", 2007 г.). Язык C был формально стандартизован в декабре 1989 г., когда Национальный институт стандартизации США (American National Standards Institute — ANSI) принял стандарт C.

Многие считают создание языка C началом современного этапа развития компьютерных языков. Он успешно объединил конфликтующие компоненты, которые доставляли столько неприятностей в предшествующих языках. Результатом явился мощный, эффективный, структурированный язык, изучение которого было сравнительно простым. Кроме того, ему была присуща еще одна, почти непостижимая особенность: он был языком программиста. До появления C языки программирования проектировались в основном либо в качестве академических упражнений, либо бюрократическими организациями. Язык C –иной. Он был спроектирован, реализован и разработан действительно работающими программистами и отражал их подход к программированию. Его функции были отлажены, проверены и многократно переработаны людьми, которые действительно использовали этот язык. В результате появился язык, который нравилось использовать программистам. Действительно, C быстро приобрел много приверженцев, которые почти молились на него. Поэтому C получил быстрое и широкое признание в программистском сообществе. Короче говоря, C — это язык, разработанный программистами и для программистов. Как вы вскоре убедитесь, Java унаследовал эту особенность.

Следующий шаг: язык C++

В конце 70-х — начале 80-х гг. C стал превалирующим компьютерным языком программирования, и он продолжает широко применяться и в настоящее время. Поскольку C — удачный и удобный язык, может возникнуть вопрос, чем обусловлена потребность в чем-либо еще. Ответ: растущей *сложностью*. На протяжении всей истории развития программирования все возрастающая сложность программ порождала потребность в более совершенных способах преодоления этой 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. Например, когда компьютеры только появились, программирование выполнялось путем ручного изменения двоичных машинных инструкций с передней панели компьютера. До тех пор, пока длина программ не превышала нескольких сотен инструкций, этот подход был вполне приемлем. С увеличением программ был изобретен язык ассемблера, который позволил программисту работать с большими, все более сложными программами, используя при этом символьные представления машинных инструкций. По мере того, как программы продолжали увеличиваться в объеме, появились языки высокого уровня, которые предоставили программисту дополнительные средства преодоления сложности программ.

Первым языком программирования, который получил широкое распространение, был, конечно же, FORTRAN. Хотя этот язык и явился первым ВПЕЧАТЛЯЮЩИМ шагом, его вряд ли можно считать языком, который способствует созданию четких и легких для понимания программ. 60-е гг. знаменовались рождением *структурного программирования*. Этот метод программирования наиболее ярко проявился в таких языках, как C. Использование структурированных языков впервые предоставило программистам возможность достаточно легко создавать программы средней сложности. Однако даже при использовании методов структурного программирования по достижении проектом определенного размера его сложность начинала превышать ту, с которой программист мог справиться. К началу 80-х гг. сложность многих проектов начала превышать такую, с которой можно было справиться с использованием структурного подхода. Для решения этой проблемы был изобретен новый способ программирования, получивший название *объектно-ориентированного программирования* (ООП). Объектно-ориентированное программирование подробно рассматривается в последующих главах этой книги, но мы все же приведем краткое определение: ООП — это методология программирования, которая помогает организовывать сложные программы за счет использования наследования, инкапсуляции и полиморфизма.

Поведем итоги сказанному. Хотя C — один из основных мировых языков программирования, существует предел его способности справляться со сложностью программ. Как только размеры программы превышают определенное значение, она становится слишком сложной, чтобы ее можно было охватить как единое целое. Хотя точное значение этого предела зависит как от структуры самой программы, так и от подходов, используемых программистом, начиная с определенного момента, любая программа становится слишком сложной для понимания и внесения изменений. Язык C++ предоставил возможности, которые позволили программисту преодолевать этот барьер и понимать и управлять крупными по размерам программами.

Язык C++ был изобретен Бьярном Страуструпом в 1979 г. во время его работы в компании Bell Laboratories в городе Мюррей-Хилл, шт. Нью-Джерси. Вначале Страуструп назвал новый язык “C with Classes” (“C с классами”). Однако в 1983 г. это название было изменено на C++. Язык C++ расширяет функциональные возможности C, добавляя в него объектно-ориентированные свойства. Поскольку C++ построен на основе C, он поддерживает все его возможности, атрибуты и преимущества. Это обстоятельство явилось главной причиной успешного распространения C++ в качестве языка программирования. Изобретение C++ не было попыткой создания совершенно нового языка программирования. Напротив, все усилия были направлены на усовершенствование уже существующего очень удачного языка.

Предпосылки к созданию языка Java

К концу 80-х — началу 90-х гг. объектно-ориентированное программирование с применением языка C++ стало основным методом программирования. Действительно, в течение некоторого непродолжительного времени казалось, что программисты, наконец, изобрели идеальный язык. Поскольку C++ сочетал в себе высокую эффективность и стилистические элементы C с объектно-ориентированным подходом, этот язык можно было использовать для создания самого широкого круга программ. Однако, как и в прошлом, уже вызревали факторы, которые должны были, в который раз, стимулировать развитие компьютерных языков. Пройдет еще несколько лет и World Wide Web и Internet достигнут критической массы. Это приведет к еще одной революции в программировании.

Создание языка Java

Начало разработке Java было положено в 1991 г. Джеймсом Гослингом (James Gosling), Патриком Нотоном (Patrick Naughton), Крисом Вартом (Chris Warth), Эдом Франком (Ed Frank) и Майком Шериданом (Mike Sheridan), работавшими в компании Sun Microsystems, Inc. Разработка первой работающей 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. множество других людей приняло участие в проектировании и развитие этого языка. 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.

Как ни странно, первоначальной побудительной причиной к созданию Java послужила вовсе не сеть Internet! Основной причиной была потребность в независящем от платформы (т.е. архитектурно нейтральном) языке, который можно было бы использовать для создания программного обеспечения, встраиваемого в различные бытовые электронные устройства, такие как СВЧ-печи и устройства дистанционного управления. Как не трудно догадаться, в качестве контроллеров используется множество различных типов процессоров. Проблема применения языков C и C++ (как и большинства других языков) состоит в том, что написанные на них программы должны компилироваться для конкретной платформы. Хотя программы C++ могут быть скомпилированы практически для любого типа процессора, для этого требуется наличие полного компилятора C++, предназначенного для данного процессора. Проблема в том, что создание компиляторов обходится дорого и требует значительного времени. Поэтому требовалось более простое — и экономически выгодное — решение. Пытаясь найти такое решение, Гослинг и другие начали работу над переносимым, не зависящим от платформы языком, который можно было бы использовать для создания кода, пригодного для выполнения на различных процессорах в различных средах. Вскоре эти усилия привели к созданию языка Java.

Примерно в то же время, когда определялись основные характеристики Java, на сцену выступил второй, несомненно, более важный фактор, который должен был сыграть решающую роль в судьбе этого языка. Конечно же, этим вторым фактором была World Wide Web. Если бы формирование Web не происходило почти одновременно с реализацией Java, этот язык мог бы остаться полезным, но оставшимся незамеченным языком программирования бытовых электронных устройств. Но с появлением World Wide Web Java вышел на передний рубеж проектирования компьютерных языков, поскольку Web также нуждалась в переносимых программах.

Еще на заре своей карьеры большинство программистов твердо усвоили, что переносимые программы столь же недостижимы, сколь и желанны. В то время как потребность в средстве создания эффективных, переносимых (не зависящих от платформы) программ почти столь же стара, как и сама отрасль программирования, она отодвигалась на задний план другими, более насущными проблемами. Более того, поскольку большая часть самого мира компьютеров была разделена на три конкурирующих лагеря Intel, Microsoft и UNIX, большинство программистов оставалось запертым в своих аппаратно-программных "твердынях", что несколько снижало потребность в переносимом коде. Тем не менее, с появлением Internet и Web старая проблема переносимости снова возникла с еще большей актуальностью. В конце концов, Internet представляет собой разнообразную и распределенную вселенную, заполненную множеством различных типов компьютеров, операционных систем и процессоров. Несмотря на то что к Internet подключено множество

типов платформ, пользователям желательно, чтобы все они могли выполнять одинаковые программы. То, что в начале было неприятной, но не слишком насущной проблемой, превратилось в потребность первостепенной важности.

К 1993 г. членам группы проектирования Java стало очевидным, что проблемы переносимости, часто возникающие при создании кода, предназначенного для встраивания в контроллеры, возникают также и при попытках создания кода для Internet. Фактически, та же проблема, для решения которой в малом масштабе предназначался язык Java, в большем масштабе была актуальна и в среде Internet. Понимание этого обстоятельства вынудило разработчиков языка Java перенести свое внимание с бытовой электроники на программирование для Internet. Таким образом, хотя потребность в архитектурно нейтральном языке программирования послужило своего рода "начальной искрой", Internet обеспечила крупномасштабный успех Java.

Как уже упоминалось, Java наследует многие из своих характеристик от языков C и C++. Это сделано намеренно. Разработчики Java знали, что использование знакомого синтаксиса C и повторение объектно-ориентированных свойств C++ должно было сделать их язык привлекательным для миллионов опытных программистов на C/C++. Помимо внешнего сходства, Java использует ряд других атрибутов, которые способствовали успеху языков C и C++. Во-первых, язык Java был спроектирован, проверен и усовершенствован настоящими работающими программистами. Этот язык построен с учетом потребностей и опыта людей, которые его создали. Таким образом, Java — это язык программистов. Во-вторых, Java целостен и логически непротиворечив. В-третьих, если не учитывать ограничения, накладываемые средой Internet, Java предоставляет программисту полный контроль над программой. Если программирование выполняется правильно, это непосредственно отражается в программах. В равной степени справедливо и обратное. Иначе говоря, Java не является языком тренажера. Это язык профессиональных программистов.

Из-за сходства характеристик Java и C, кое-кто склонен считать Java просто "Internet-версией C++". Однако это серьезное заблуждение. Языку Java присущи значительные практические и концептуальные отличия. Хотя и верно, что C++ оказал влияние на характеристики языка Java, последний не является усовершенствованной версией C++. Например, Java не обладает совместимостью с C++ ни по восходящей, ни по нисходящей. Конечно, сходство с языком C++ значительно, и в программе Java программист C++ будет чувствовать себя как дома. Вместе с тем, Java не предназначен служить заменой C++. Java был предназначен для решения одного набора проблем, а C++ — для решения другого. Еще длительное время оба эти языка неизбежно будут сосуществовать.

Как уже было отмечено в начале этой главы, развитие компьютерных языков обусловлено двумя причинами: необходимостью адаптации к изменениям в среде и необходимостью реализации новых идей в области программирования. Изменением среды, которое обусловило потребность в языке, подобном Java, была потребность в не зависящих от платформы программах, предназначенных для распространения в Internet. Однако Java изменяет также подход к написанию программ. В частности, Java углубил и усовершенствовал объектно-ориентированный подход, использованный в C++, добавил в него поддержку многопоточной обработки и предоставил библиотеку, которая упростила доступ к Internet. Однако столь поразительный успех Java обусловлен не теми или иными его отдельными особенностями, а их совокупностью как языка в целом. Он явился прекрасным ответом на потребности в то время лишь зарождающейся среды в высшей степени распределенных компьютерных систем. Для программирования Internet-программ Java стал тем, чем язык C был для системного программирования: революционной силой, которая изменила мир.

Связь с языком C#

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

Вероятно, наиболее наглядным примером влияния Java на программирование служит язык C#. Созданный в компании Microsoft для поддержки каркаса .NET Framework, язык C# тесно связан с Java. Например, оба эти языка используют одинаковый общий синтаксис, поддерживают распределенное программирование и работают с одной и той же объектной моделью. Конечно, между Java и C# существует ряд различий, но в целом эти языки "выглядят" очень похожими. На сегодняшний день это "перекрестное опыление" между Java и C# — наилучшее доказательство того, что Java коренным образом изменил представление о компьютерных языках и их применении.

Как язык Java изменил Internet

Сеть Internet способствовала выдвижению языка Java на передовые рубежи программирования, а Java, в свою очередь, оказал сильнейшее влияние на Internet. Кроме того, что Java упростил создание Internet-программ в целом, он привел к появлению нового типа предназначенных для работы в сетях программ, получивших название аплетов, которые изменили понятие содержимого для сетевой среды. Кроме того, Java позволил решить две наиболее острых проблемы программирования, связанные с Internet: переносимость и безопасность. Рассмотрим каждую из этих проблем.

Аплеты Java

Аплет — это особый вид программы Java, предназначенный для передачи по Internet и автоматического выполнения Java-совместимым Web-браузером. Более того, аплет загружается по требованию, не требуя дальнейшего взаимодействия с пользователем. Если пользователь щелкает на ссылке, которая содержит аплет, аплет автоматически загружается и запускается в браузере. Аплеты создаются в виде небольших программ. Как правило, они используются для отображения данных, предоставляемых сервером, обработки действий пользователя или выполнения простых функций, таких как вычисление процентов по кредитам, которые выполняются локально, а не на сервере. По сути, аплет позволяет перенести ряд функций с сервера к клиенту.

Появление аплетов изменило программирование Internet-приложений, поскольку они расширили совокупность объектов, которые можно свободно перемещать по киберпространству. Если говорить в целом, между сервером и клиентом передаются две большие категории объектов: пассивная информация и динамические, активные программы. Например, чтение сообщений электронной почты подразумевает просмотр пассивных данных. Даже при загрузке программы ее код остается пассивными данными вплоть до момента выполнения. И напротив, аплет представляет собой динамическую, автоматически выполняющуюся программу. Такая программа является активным агентом на клиентском компьютере, хотя она и инициируется сервером.

Насколько желательно, чтобы программы были динамическими, как это имеет место при использовании сетевых программ, настолько же они представляют серьезные проблемы с точки зрения безопасности и переносимости. Очевидно, что компьютер клиента необходимо обезопасить от нанесения ему ущерба программой, которая загружается в него,

а затем автоматически выполняется. Кроме того, такая программа должна быть способна выполняться в различных аппаратных средах и под управлением различных операционных систем. Как читатели вскоре убедятся, язык Java решает эти проблемы эффективно и элегантно. Рассмотрим их подробнее.

Безопасность

Как, вероятно, известно читателям, каждая загрузка "обычной" программы сопряжена с риском, поскольку загружаемый код может содержать вирус, "троянского коня" или вредоносный код. Корень проблемы в том, что вредоносный код может выполнить свое "черное" дело, поскольку получает несанкционированный доступ к системным ресурсам. Например, просматривая содержимое локальной файловой системы компьютера, программа вируса может собирать конфиденциальную информацию, такую как номера кредитных карточек, сведения о состоянии банковских счетов и пароли. Для обеспечения безопасности загрузки и выполнения Java-аплетов на компьютере клиента было необходимо воспрепятствовать аплетам предпринимать подобные атаки.

Java обеспечивает эту защиту, заключая аплет в среду выполнения Java и не предоставляя ему доступ к другим частям операционной системы компьютера. (Способы достижения этого рассматриваются в последующих разделах.) Возможность загрузки аплетов с сохранением при этом уверенности в невозможности нанесения вреда системе и нарушения системы безопасности многие эксперты и пользователи считают наиболее новаторским аспектом Java.

Переносимость

Переносимость — основная особенность Internet, поскольку эта глобальная сеть соединяет множество различных типов компьютеров и операционных систем. Чтобы Java-программа могла выполняться буквально на любом компьютере, подключенном к Internet, требовался метод обеспечения выполнения этой программы в различных системах. Например, применительно к аплету это означает, что один и тот же аплет должен иметь возможность загружаться и выполняться на широком множестве центральных процессоров, операционных систем и браузеров, подключенных к Internet. Создание различных версий аплетов для различных компьютеров совершенно нерационально. *Один и тот же* код должен работать на *всех* компьютерах. Поэтому требовался какой-то механизм для генерирования переносимого выполняемого кода. Как вы вскоре убедитесь, тот же механизм, который способствует обеспечению безопасности, способствует также созданию переносимых программ.

Магия Java: байт-код

Основная особенность, которая позволяет языку Java решать обе описанные проблемы обеспечения безопасности и переносимости программ состоит в том, что вывод компилятора Java не является исполняемым кодом. Скорее он представляет собой так называемый байт-код. *Байт-код* — это в высшей степени оптимизированный набор инструкций, предназначенных для исполнения системой времени выполнения Java, называемой *виртуальной машиной Java* (Java Virtual Machine — JVM). Собственно говоря, первоначальная версия JVM разрабатывалась в качестве *интерпретатора байт-кода*. Это может вызывать определенное удивление, поскольку по соображениям обеспечения максимальной производительности многие современные языки призваны создавать исполняемый код.

Однако то, что Java-программа интерпретируется машиной JVM, помогает решать основные проблемы, связанные с программами, предназначенными для Web. И вот почему.

Трансляция Java-программы в байт-код значительно упрощает ее выполнение в широком множестве сред, поскольку на каждой платформе необходимо реализовать только JVM. Как только в данной системе появляется пакет времени выполнения, в ней можно исполнять любую Java-программу. Следует помнить, что хотя на различных платформах особенности реализации машины JVM могут быть различными, все они могут выполнять обработку одного и того же байт-кода. Если бы Java-программа компилировалась во внутренний код, для каждого типа процессоров, подключенных к Internet, должны были бы существовать отдельные версии одной и той же программы. Понятно, что такое решение неприемлемо. Таким образом, выполнение байт-кода машиной JVM — простейший способ создания действительно переносимых программ.

То, что Java-программа выполняется машиной JVM, способствует также повышению ее безопасности. Поскольку машина JVM управляет выполнением программы, она может изолировать программу и воспрепятствовать порождению ею побочных эффектов вне данной системы. Как вы убедитесь, ряд ограничений, существующих в языке Java, также способствует повышению безопасности.

В общем случае, когда программа компилируется в промежуточную форму, а затем интерпретируется виртуальной машиной, она выполняется медленнее, чем если бы она была скомпилирована в исполняемый код. Однако при использовании языка Java различие в производительности не слишком велико. Поскольку байт-код в высокой степени оптимизирован, его применение позволяет машине JVM выполнять программы значительно быстрее, чем можно было ожидать.

Хотя язык Java был задуман в качестве интерпретируемого языка, ничто не препятствует Java выполнять компиляцию байт-кода во внутренний код “на лету” для повышения производительности. Поэтому вскоре после появления Java компания Sun начала поставлять свою технологию HotSpot. Эта технология предоставляет оперативный (Just-In-Time — JIT) компилятор байт-кода. Когда JIT-компилятор является составной частью машины JVM, избранные фрагменты байт-кода один за другим компилируются в исполняемый код в реальном времени, по соответствующим запросам. Важно понимать, что одновременная компиляция всей Java-программы в исполняемый код нецелесообразна, поскольку Java выполняет различные проверки, которые могут быть осуществлены только во время выполнения. Вместо этого во время выполнения JIT-компилятор компилирует код по мере необходимости. Более того, компилируются не все фрагменты байт-кода, а только те, которым компиляция принесет выгоду. Остальной код просто интерпретируется. Однако JIT-подход все же обеспечивает значительное повышение производительности. Даже в случае применения к байт-коду динамической компиляции, характеристики переносимости и безопасности сохраняются, поскольку машина JVM по-прежнему отвечает за целостность среды выполнения.

Сервлеты: серверные Java-программы

Как ни полезны аплеты, они — всего лишь половина системы клиент/сервер. Вскоре после появления языка Java стало очевидно, что он может пригодиться и на серверах. В результате появились *сервлеты* (servlet). Сервлет — это небольшая программа, выполняемая на сервере. Подобно тому как аплеты динамически расширяют функциональные возможности Web-браузера, сервлеты динамически расширяют функциональные возможности Web-сервера. Таким образом, с появлением сервлетов язык Java распространился на оба конца соединения клиент/сервер.

Сервлеты служат для создания динамически генерируемого содержимого, которое затем обслуживает клиента. Например, интерактивный склад может использовать сервлет для поиска стоимости товара в базе данных. Затем информация о цене используется для динамической генерации Web-страницы, отправляемой браузеру. Хотя динамически генерируемое содержимое доступно также посредством таких механизмов, как CGI (Common Gateway Interface — общий шлюзовой интерфейс), сервлет обеспечивает ряд преимуществ, в том числе — повышение производительности.

Поскольку сервлеты (подобно всем Java-программам) компилируются в байт-код и выполняются машиной JVM, они в высшей степени переносимы. Следовательно, один и тот же сервлет может применяться в различных серверных средах. Единственные необходимые условия для этого — поддержка сервером машины JVM и контейнера сервлета.

Терминология, связанная с Java

Рассмотрение истории создания и развития языка Java было бы неполным без рассмотрения специфичной терминологии Java. Хотя основные факторы, обусловившие изобретение Java — необходимость обеспечения переносимости и безопасности, другие факторы также сыграли свою роль в формировании окончательной версии языка. Группа разработки Java обобщила основные понятия в следующем перечне терминов:

- простота;
- безопасность;
- переносимость;
- объектная ориентированность;
- устойчивость;
- многопоточность;
- архитектурная нейтральность;
- интерпретируемость;
- высокая производительность;
- распределенность;
- динамический характер.

Два из этих терминов мы уже рассмотрели: безопасность и переносимость. Рассмотрим значения остальных терминов.

Простота

Java был задуман в качестве простого в изучении и эффективного в использовании профессиональными программистами языка. Для тех, кто обладает определенным опытом программирования, овладение языком Java не представит особой сложности. Если же вы уже знакомы с базовыми концепциями объектно-ориентированного программирования, изучение Java будет еще проще. А для опытного программиста на C++ переход к Java вообще потребует минимум усилий. Поскольку Java наследует синтаксис C/C++ и многие объектно-ориентированные свойства C++, для большинства программистов изучение Java не представит сложности.

Объектная ориентированность

Хотя предшественники языка Java и оказали влияние на его архитектуру и синтаксис, при его проектировании задача совместимости по исходному коду с каким-либо другим языком не ставилась. Это позволило группе разработки Java выполнять проектирование, что называется, с чистого листа. Одним из следствий этого явился четкий, практичный, прагматичный подход к объектам. Притом что Java позаимствовал свойства многих удачных объектно-программных сред, разработанных на протяжении нескольких последних десятилетий, в нем удалось достичь баланса между строгим соблюдением концепции "все компоненты программы — объекты" и более прагматичной моделью "прочь с дороги". Объектная модель Java проста и легко расширяема. В то же время примитивные типы, такие как целые числа, сохраняются в виде высокопроизводительных компонентов, не являющихся объектами.

Устойчивость

Многоплатформенная среда Web предъявляет к программам повышенные требования, поскольку они должны надежно выполняться в разнообразных системах. Поэтому способность создавать устойчивые программы была одним из главных приоритетов при проектировании Java. Для обеспечения надежности Java накладывает ряд ограничений в нескольких наиболее важных областях, что вынуждает программиста выявлять ошибки на ранних этапах разработки программы. В то же время Java избавляет от беспокойства по поводу многих наиболее часто встречающихся ошибок программирования. Поскольку Java — строго типизированный язык, проверка кода выполняется во время компиляции. Однако проверка кода осуществляется и во время выполнения. В результате многие трудно обнаруживаемые программные ошибки, которые часто ведут к возникновению трудновоспроизводимых ситуаций времени выполнения, в Java-программе попросту невозможны. Предсказуемость кода в различных ситуациях — одна из основных особенностей Java.

Чтобы понять, чем достигается устойчивость Java-программ, рассмотрим две основных причины программных сбоев: ошибки управления памятью и неправильная обработка исключений (т.е. ошибки времени выполнения). 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 проблем, поскольку программисты либо забывают освободить ранее зарезервированную память, либо, что еще хуже, пытаются освободить участок памяти, все еще используемый другой частью кода. Java полностью исключает такие ситуации, автоматически управляя резервированием и освобождением памяти. (Фактически, освобождение выполняется полностью автоматически, поскольку Java предоставляет функцию сборки мусора в отношении неиспользуемых объектов.) В традиционных средах условия исключений часто возникают в таких ситуациях, как деление на ноль или "файл не найден", и управление ими должно осуществляться с помощью громоздких и трудных для понимания конструкций. Java облегчает выполнение этой задачи, предлагая объектно-ориентированный механизм обработки исключений. В хорошо написанной Java-программе все ошибки времени выполнения могут — и должны — управляться самой программой.

Многопоточность

Язык Java был разработан в ответ на потребность создания интерактивных сетевых программ. Для достижения этой цели Java поддерживает написание многопоточных

программ, которые могут одновременно выполнять много действий. Система времени выполнения Java содержит изящное, но вместе с тем сложное решение задачи синхронизации множества процессов, которое позволяет создавать действующие без перебоев интерактивные системы. Простой в применении подход к организации многопоточной обработки, реализованный в Java, позволяет программисту сосредоточивать свое 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. Одна из главных проблем, стоящих перед программистами — отсутствие гарантий того, что код, созданный сегодня, будет успешно выполняться завтра, даже на том же самом компьютере. Операционные системы и процессоры модернизируются, а все изменения в основных системных ресурсах могут приводить к неработоспособности программ. Пытаясь изменить эту ситуацию, проектировщики приняли ряд жестких решений в языке Java и виртуальной машине Java. Они поставили перед собой цель, чтобы "программы создавались лишь однажды, в любой среде, в любое время и навсегда". В значительной степени эта цель была достигнута.

Интерпретируемость и высокая производительность

Как уже говорилось, выполняя компиляцию программ в промежуточное представление, называемое байт-кодом, Java позволяет создавать многоплатформенные программы. Этот код может выполняться в любой системе, которая реализует виртуальную машину Java. Самые первые попытки получения многоплатформенных решений добивались поставленной цели за счет снижения производительности. Как пояснялось ранее, байт-код Java был тщательно спроектирован так, чтобы посредством использования JIT-компиляции его можно было с высокой производительностью легко преобразовывать во внутренний машинный код. Системы времени выполнения Java, которые предоставляют эту функцию, сохраняют все преимущества кода, не зависящего от платформы.

Распределенность

Язык Java предназначен для распределенной среды Internet, поскольку он поддерживает протоколы семейства TCP/IP. Фактически обращение к ресурсу через URL-адрес не очень отличается от обращения к файлу. Java поддерживает также *удаленный вызов методов* (Remote Method Invocation — RMI). Это свойство позволяет программам вызывать методы по сети.

Динамический характер

Программы Java содержат значительный объем информации времени выполнения, которая используется для проверки полномочий и предоставления доступа к объектам во время выполнения. Это позволяет выполнять безопасную и технически оправданную динамическое связывание кода. Это обстоятельство исключительно важно для устойчивости среды Java, в которой небольшие фрагменты байт-кода могут динамически обновляться в действующей системе.

Эволюция языка Java

Первоначальная версия Java не содержала никаких особо революционных решений, но она не ознаменовала собой завершение эры быстрого совершенствования этого языка.

В отличие от большинства других систем программирования, совершенствование которых происходило небольшими, последовательными шагами, язык Java продолжает стремительно развиваться. Уже вскоре после выпуска версии Java 1.0, разработчики создали версию Java 1.1. Добавленные в эту версию функциональные возможности значительно превосходили те, которые можно было ожидать, судя по изменению подномера версии. Разработчики добавили много новых библиотечных элементов, переопределили способ обработки событий и изменили конфигурацию многих свойств библиотеки версии 1.0. Кроме того, они отказались от нескольких свойств (признанных устаревшими), которые первоначально были определены в Java 1.0. Таким образом, в версии Java 1.1 были как добавлены новые атрибуты, так и удалены некоторые атрибуты, определенные в первоначальной спецификации.

Следующей базовой версией Java стала версия Java 2, где "2" означает "второе поколение". Создание Java 2 явилось знаменательным событием, означавшим начало "современной эры" Java. Первой версии Java 2 был присвоен номер 1.2. Это может казаться 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 стала выпускать программное обеспечение Java в виде пакета J2SE (Java 2 Platform Standard Edition — Стандартная версия платформы Java 2), и теперь номера версий применяются к этому продукту.

В Java 2 была добавлена поддержка ряда новых функций, таких как Swing и Collections Framework. Кроме того, были усовершенствованы виртуальная машина Java и различные средства программирования. Из Java 2 был исключен также ряд свойств. Наибольшие изменения претерпел класс потока `Thread`, в котором методы `suspend()`, `resume()` и `stop()` были представлены как устаревшие.

Версия J2SE 1.3 была первой серьезной модернизацией первоначальной версии Java J2SE. В основном модернизация свелась к расширению существующих функциональных возможностей и "уплотнению" среды разработки. В общем случае программы, написанные для версий 1.2 и 1.3, совместимы по исходному коду. Хотя версия 1.3 содержала меньший набор изменений, чем три предшествующих базовых версии, это не снижало ее важности.

Версия J2SE 1.4 продолжила совершенствование языка Java. Эта версия содержала несколько важных модернизаций, усовершенствований и добавлений. Например, в нее было добавлено новое ключевое слово `assert`, цепочки исключений и подсистема ввода-вывода на основе каналов. Изменения были внесены и в каркас Collections Framework и сетевые классы. Эта версия содержала также множество небольших изменений. Несмотря на значительное количество новых функциональных возможностей, версия 1.4 сохранила почти стопроцентную совместимость по исходному коду с предшествующими версиями.

Следующей версией Java стала версия J2SE 5, в которой был внесен ряд революционных изменений. В отличие от большинства предшествующих модернизаций Java, которые предоставляли важные, но постепенные усовершенствования, J2SE 5 коренным образом расширяет область применения, возможности и диапазон языка. Чтобы оценить объем изменений, внесенных в язык Java в версии J2SE 5, ознакомьтесь с перечнем основных новых функциональных возможностей:

- общие функции;
- аннотации;
- автоматическое помещение в контейнер и автоматическое извлечение из контейнера;
- перечисления;
- усовершенствованный, поддерживающий стиль `for-each`, цикл `for`;
- 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 представляет значительное добавление в языке Java. Некоторые из них вроде общих функций, усовершенствованного цикла `for` и аргументов переменной длины представляют новые синтаксические элементы. Другие, такие как автоматическое помещение в контейнер и автоматическое извлечение из контейнера, изменяют семантику языка. Аннотации вносят в программирование совершенно новое измерение. В любом случае влияние всех этих добавлений вышло за рамки их прямого эффекта. Они полностью изменили сам характер языка Java.

Важность новых функциональных возможностей нашла отражение в примененном номере версии — "5". Если следовать привычной логике, следующим номером версии Java должен был быть 1.5. Однако новые свойства столь значительны, что переход от версии 1.4 к версии 1.5 не отражал масштаб внесенных изменений. Поэтому, чтобы подчеркнуть значимость этого события, в компании Sun решили присвоить новой версии номер 5. Поэтому версия продукта была названа J2SE 5, а комплект разработчика — JDK 5. Тем не менее, для сохранения единообразия в компании Sun решили использовать номер 1.5 в качестве внутреннего номера версии, называемого также номером *версии разработки*. Цифру 5 в обозначении версии называют номером *версии продукта*.

Java SE 6

Новейшая (на момент подготовки этого издания к печати) версия Java получила название SE 6, что нашло свое отражение в материалах книги. С выходом Java SE 6 компания Sun решила в очередной раз изменить название платформы Java. Во-первых, цифра 2 в названии была опущена. Таким образом, теперь платформа называется *Java SE*, а официальное название продукта — *Java Platform, Standard Edition 6* (Платформа Java, стандартная версия 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 построена на основе версии и добавляет к ней ряд дальнейших усовершенствований. Она не содержит дополнений к числу основных функций языка Java, но расширяет библиотеки API, добавляет несколько новых пакетов и предоставляет ряд усовершенствований времени выполнения. Что касается настоящей книги, то наиболее значительными являются изменения в описании ядра API. Во многие пакеты были добавлены новые классы, а во многие классы — новые методы. Все эти изменения нашли свое отражение в разделах книги. В целом версия Java SE 6 призвана закрепить достижения, полученные в J2SE 5.

Культура инновации

С самого начала язык Java оказался в центре культуры инновации. Его первоначальная версия изменила подход к программированию для Internet. Виртуальная машина Java (JVM) и байт-код изменили представления о безопасности и переносимости. Аплет (а затем и сервлет) вдохнули жизнь в Web. Процесс Java Community Process (JCP) изменил способ ассимиляции новых идей в языке. Никогда мир языка Java не оставался неизменным в течение длительного времени. На момент подготовки этого издания версия Java SE 6 являлась самой новой в непрекращающемся динамичном развитии языка Java.

2 ГЛАВА

Обзор языка Java

Как и во всех компьютерных языках, элементы Java существуют не сами по себе. Скорее они работают совместно, образуя язык в целом. Однако эта взаимосвязанность может затруднять описание какого-то одного аспекта Java, не касаясь при этом нескольких других аспектов. Часто для понимания одного свойства требуется наличия знаний о другом. Поэтому в настоящей главе представлен краткий обзор нескольких основных свойств языка Java. Приведенный здесь материал послужит отправной точкой, которая позволит создавать и понимать простые программы. Большинство рассмотренных в этой главе тем будет подробнее рассмотрено в остальных главах части I.

Объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) — это основная характеристика языка Java. Фактически все Java-программы являются, по меньшей мере, частично, объектно-ориентированными. Язык Java связан с ООП настолько тесно, что прежде чем приступать к написанию на нем даже простейших программ, следует вначале ознакомиться с базовыми принципами ООП. Поэтому ознакомление с материалом этой главы мы начнем с рассмотрения теоретических аспектов ООП.

Две концепции

Все компьютерные программы состоят из двух элементов: кода и данных. Более того, концептуально программа может быть организована вокруг своего кода или вокруг своих данных. То есть организация некоторых программ определяется тем "что происходит", а других — тем, "на что оказывается влияние". Существуют две концепции создания программы. Первый способ называют *моделью, ориентированной на процессы*. Этот подход характеризует программу в виде последовательностей линейных шагов (т.е. кода). Модель, ориентированную на процессы, можно рассматривать в качестве *кода, воздействующего на данные*. Процедурные языки вроде C достаточно успешно используют эту модель. Однако, как было отмечено в главе 1, этот подход порождает ряд проблем с увеличением размеров и сложности программ.

Для преодоления увеличивающейся сложности была начата разработка подхода, названного *объектно-ориентированным программированием*. Объектно-ориентированное программирование организует программу вокруг ее данных (т.е. объектов) и набора подробно определенных интерфейсов к этим данным. Объектно-ориентированную программу можно характеризовать *как данные, управляющие доступом к коду*. Как будет показано в дальнейшем, передавая функции управления данным, можно получить несколько организационных преимуществ.

Абстракция

Важный элемент объектно-ориентированного программирования — *абстракция*. Человеку свойственно представлять сложные явления и объекты посредством абстракции. Например, люди представляют себе автомобиль не в виде набора десятков тысяч отдельных деталей, а в виде совершенно определенного объекта, имеющего собственное уникальное поведение. Эта абстракция позволяет не задумываться о сложности деталей, образующих автомобиль, используя его, скажем, для поездки в магазин. Мы можем не обращать внимания на подробности работы двигателя, коробки передач и тормозной системы. Вместо этого объект можно использовать как единое целое.

Мощное средство применения абстракции — применение иерархических классификаций. Это позволяет упрощать семантику сложных систем, разбивая их на более пригодные для управления фрагменты. 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 устройства воспроизведения аудиокассет. Суть сказанного в том, что сложную структуру автомобиля (или любой другой сложной системы) можно описать с помощью иерархических абстракций.

Иерархические абстракции сложных систем можно применять и к компьютерным программам. Посредством абстракции данные традиционной ориентированной на процессы программы можно преобразовать в составляющие ее объекты. При этом последовательность шагов процесса может быть преобразована в коллекцию сообщений, передаваемых между этими объектами. Таким образом, каждый и этих объектов описывает свое уникальное поведение. Эти объекты можно считать конкретными элементами, которые отвечают на сообщения, указывающие им о необходимости *выполнить что-либо*. Сказанное — суть объектно-ориентированного программирования.

Концепции объектной ориентированности лежат в основе языка Java, точно так же, как они лежат в основе восприятия мира человеком. Важно понимать, как эти концепции реализуются в программах. Как вы увидите, объектно-ориентированное программирование — мощная и естественная концепция создания программ, которые способны пережить неизбежные изменения, сопровождающие жизненный цикл любого крупного программного проекта, включая создание концепции, рост и старение. Например, при наличии тщательно определенных объектов и четких, надежных интерфейсов к этим объектам можно безбоязненно и без особых проблем извлекать или заменять части старой системы.

Три принципа ООП

Все языки объектно-ориентированного программирования предоставляют механизмы, которые облегчают реализацию объектно-ориентированной модели. Этими механизмами являются инкапсуляция, наследование и полиморфизм. Рассмотрим эти концепции.

Инкапсуляция

Инкапсуляция — механизм, который связывает код и данные, которыми он манипулирует, защищая оба эти компонента от внешнего вмешательства и злоупотреблений. Один из возможных способов представления инкапсуляции — представление в виде защитной оболочки, которая предохраняет код и данные от произвольного доступа со стороны другого кода, находящегося снаружи оболочки. Доступ к коду и данным, находящимся внутри оболочки, строго контролируются тщательно определенным интерфейсом. Чтобы провести аналогию с реальным миром, рассмотрим автоматическую коробку передач автомобиля. Она инкапсулирует сотни бит информации об автомобиле, такой как степень ускорения, крутизна поверхности, по которой совершается движение и положение рычага переключения скоростей. Пользователь (водитель) может влиять на эту сложную инкапсуляцию только одним методом: перемещая рычаг переключения скоростей. На коробку передач нельзя влиять, например, посредством индикатора поворота или дворников. Таким образом, рычаг переключения скоростей — строго определенный (а в действительности единственный) интерфейс к коробке передач. Более того, происходящее внутри коробки передач, не влияет на объекты, находящиеся вне ее. Например, переключение передач не включает фары! Поскольку функция автоматического переключения передач инкапсулирована, десятки изготовителей автомобилей могут реализовать ее каким угодно способом. Однако с точки зрения водителя все эти коробки передач работают одинаково. Аналогичную идею можно применять к программированию. Сила инкапсулированного кода в том, что все знают, как к нему можно получить доступ, и, следовательно, могут его использовать независимо от нюансов реализации и не опасаясь неожиданных побочных эффектов.

В языке Java основой инкапсуляции является класс. Хотя подробнее мы рассмотрим классы в последующих главах книги, сейчас полезно ознакомиться со следующим кратким описанием. *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 имеет физическое воплощение.

При создании класса определяют код и данные, которые образуют этот класс. Совокупность этих элементов называют *членами* класса. В частности, определенные классом данные называют *переменными-членами* или *переменными экземпляра*. Код, который выполняет действия по отношению к данным, называют *переменными-методами* или просто *методами*. (То, что программисты на Java называют *методом*, программисты на C/C++ называют *функциями*.) В правильно написанных Java-программах методы определяют способы использования переменных-членов. Это означает, что поведение и интерфейс класса определяются методами, которые выполняют действия по отношению к данным его экземпляра.

Поскольку назначение класса — инкапсуляция сложной структуры программы, существуют механизмы сокрытия сложной структуры реализации внутри класса. Каждый метод или переменная в классе может быть помечена как приватная или общедоступная. *Общедоступный* интерфейс класса представляет все, что должны или могут знать внешние пользователи класса. *Приватные* методы и данные могут быть доступны только для кода, который является членом данного класса. Следовательно, любой другой код, не являющийся членом класса, не может получать доступ к приватному методу или переменной. Поскольку приватные члены класса доступны другим частям программы только посредством общедоступных методов класса, можно быть уверенным в невозможности

выполнения неправомерных действий. Конечно, это означает, что общедоступный интерфейс должен быть тщательно спроектирован, открывая не слишком много нюансов внутренней работы класса (рис. 2.1).

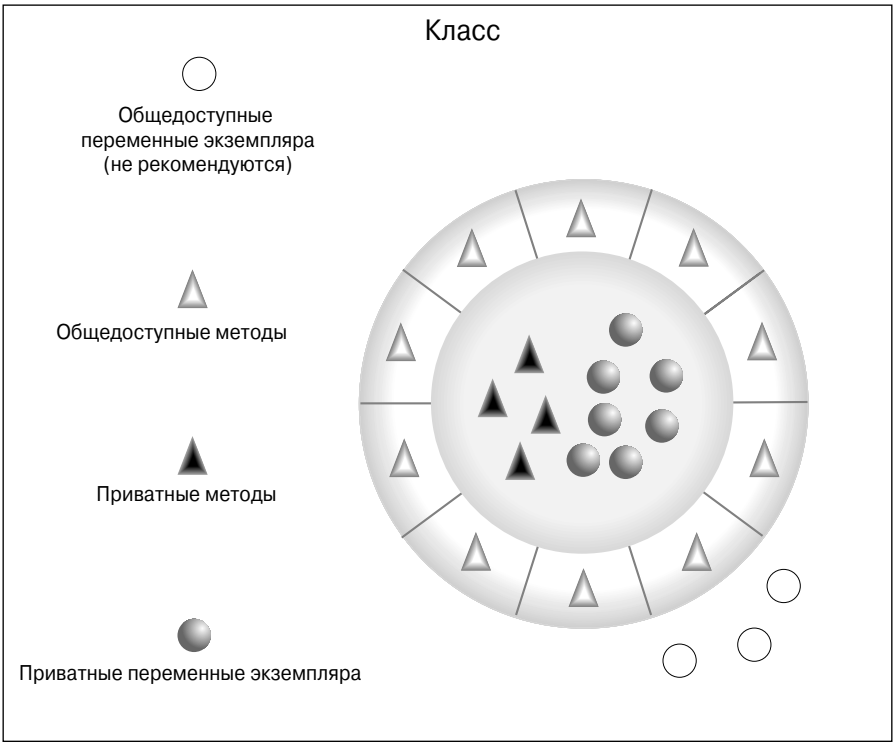

Рис. 2.1. Инкапсуляция: общедоступные методы можно использовать для защиты приватных данных

Наследование

Наследование — процесс, посредством которого один объект получает свойства другого объекта. Это важно, поскольку он поддерживает концепцию иерархической классификации. Как уже отмечалось, большинство сведений становится доступным для понимания посредством иерархических (т.е. нисходящих) классификаций. Например, золотистый ретривер — часть классификации *собака*, которая, в свою очередь, относится к классу *млекопитающие*, а тот — к еще большему классу *животных*. Без использования иерархий каждый объект должен был бы явно определять все свои характеристики. Однако благодаря наследованию объект должен определять только те из них, которые делают его уникальным внутри класса. Объект может наследовать общие атрибуты от своего родительского объекта. Таким образом, механизм наследования обеспечивает возможность того, чтобы один объект был особым экземпляром более общего случая. Рассмотрим этот процесс подробнее.

Как правило, большинство людей воспринимает 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 атрибутами, такими как размеры, уровень интеллекта и тип скелета. Животным присущи также определенные особенности поведения: они едят, дышат и спят. Приведенное описание атрибутов и поведения — определение *класса* животных.

Если бы требовалось описать более конкретный класс животных, например млекопитающих, нужно было бы указать более конкретные атрибуты, такие как тип зубов и молочных желез. Это определение называют *подклассом* животных, которые относятся к *суперклассу* (родительскому классу) млекопитающих.

Поскольку млекопитающие — всего лишь более точно определенные животные, они *наследуют* все атрибуты животных. Подкласс нижнего уровня *иерархии классов* наследует все атрибуты каждого из его родительских классов (рис. 2.2).

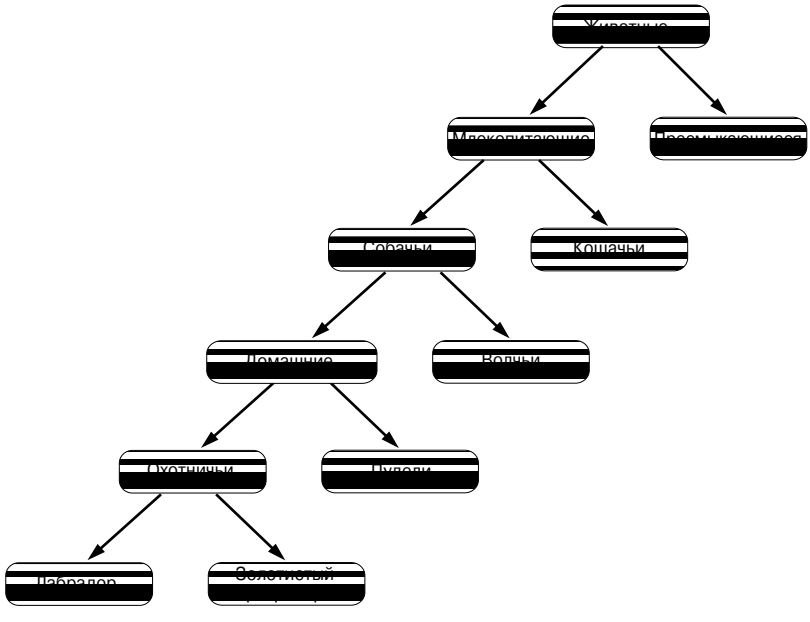

Рис. 2.2. Иерархия классов животных

Наследование связано также с инкапсуляцией. Если данный класс инкапсулирует определенные атрибуты, то любой его подкласс будет иметь эти же атрибуты *плюс* любые дополнительные атрибуты, являющиеся составной частью его специализации (рис. 2.3). Эта ключевая концепция делает возможным возрастание сложности объектно-ориентированных программ в линейной, а не геометрической прогрессии. Новый подкласс наследует все атрибуты всех своих родительских классов. Поэтому он не содержит непредсказуемых взаимодействий с большей частью остального кода системы.

Полиморфизм

Полиморфизм (от греческого слова, означающего “много форм”) — свойство, которое позволяет использовать один и тот же интерфейс для общего класса действий. Конкретное действие определяется конкретным характером ситуации.

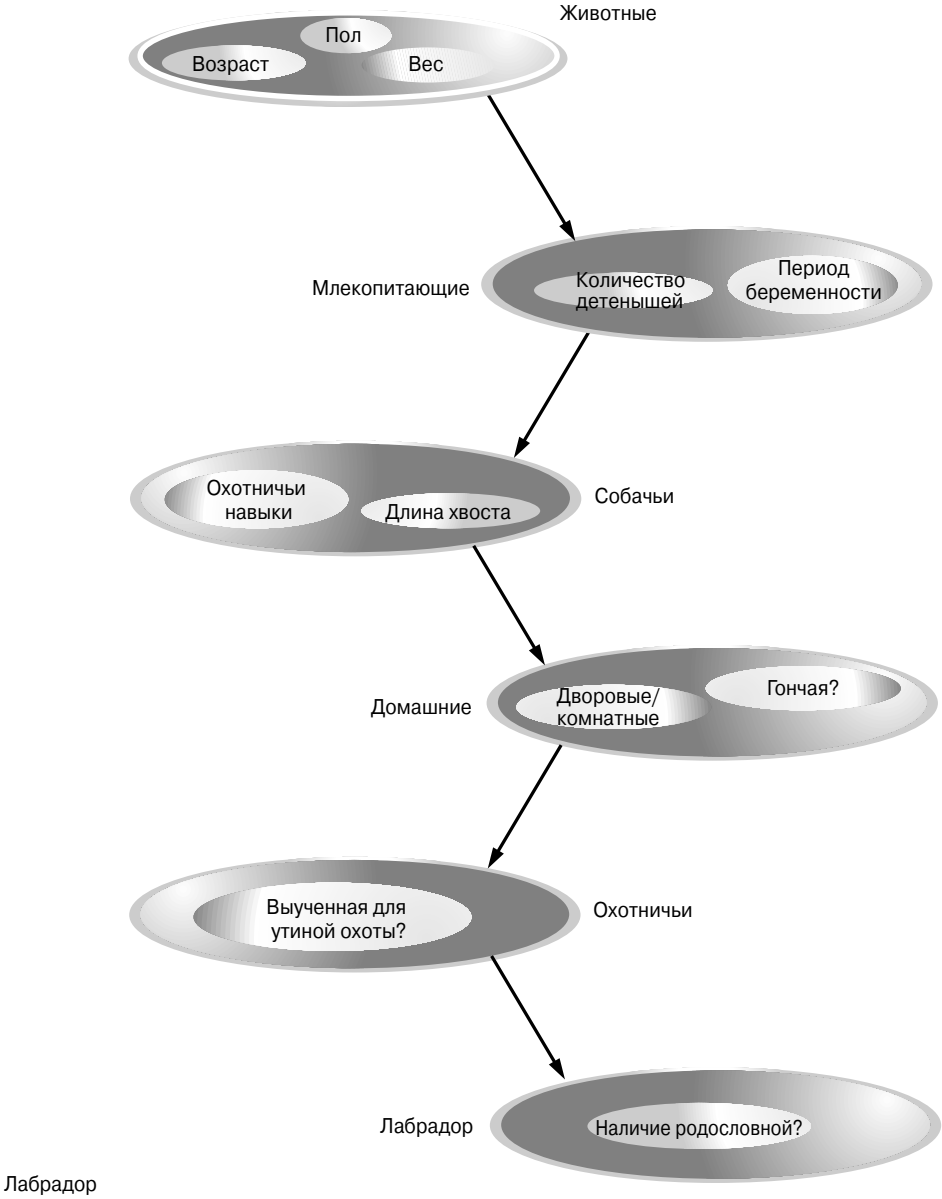

Рис. 2.3. Лабрадор полностью наследует свойства инкапсуляции всех родительских классов

Рассмотрим стек (представляющий собой список типа “последним вошел, первым вышел”). Может существовать программа, которая требует применения трех типов стеков. Один стек используется для целочисленных значений, один — для значений с плавающей точкой и один — для символов. Алгоритм реализации каждого из этих стеков остается неизменным, несмотря на различие хранящихся в них данных. В не объектно-ориентированном языке пришлось бы создавать три различных набора подпрограмм стека, каждый из которых должен был бы иметь отдельное имя. Однако в Java, благодаря полиморфизму, можно определить общий набор подпрограмм стека, использующих одни и те же имена.

В более общем виде концепцию полиморфизма часто выражают фразой “один интерфейс, несколько методов”. Это означает, что можно спроектировать общий интерфейс для группы связанных между собой действий. Этот подход позволяет уменьшить сложность программы, поскольку один и тот же интерфейс используется для указания *общего класса действий*. Выбор же *конкретного действия* (т.е. метода), применимого к каждой ситуации — задача компилятора. Программисту не нужно осуществлять этот выбор вручную. Необходимо лишь помнить об общем интерфейсе и применять его.

Если продолжить аналогию с собаками, можно сказать, что собачье обоняние — полиморфное свойство. Если собака ощутит запах кошки, она залает и погонится за ней. Если собака ощутит запах своего корма, у нее начнется слюноотделение, и она поспешит к своей миске. В обеих ситуациях действует одно и то же чувство обоняния. Различие в том, что издает запах — т.е. в типе данных, воздействующих на нос собаки! Эту же общую концепцию можно реализовать в Java применительно к методам внутри программы.

Совместное использование полиморфизма, инкапсуляции и наследования

При правильном совместном использовании полиморфизма, инкапсуляции и наследования они создают среду программирования, которая поддерживает разработку более устойчивых и масштабируемых программ, чем в случае применения модели, ориентированной на процессы. Тщательно спроектированная иерархия классов — основа многократного использования кода, на разработку и тестирование которого были затрачены время и усилия. Инкапсуляция позволяет возвращаться к ранее созданным реализациям, не разрушая код, зависящий от общедоступного интерфейса применяемых в приложении классов. Полиморфизм позволяет создавать понятный, чувствительный, удобочитаемый и устойчивый код.

Из двух приведенных примеров реального мира пример с автомобилем полнее иллюстрирует возможности объектно-ориентированного проектирования. Пример с собаками хорошо подходит для рассмотрения его с точки зрения наследования, но автомобили имеют больше общего с программами. Садясь за руль различных типов (подклассов) автомобилей, все водители используют наследование. Независимо от того, является ли автомобиль школьным автобусом, легковым мерседесом, порше или семейным микроавтобусом, все водители могут более-менее легко найти и пользоваться рулем, тормозами и педалью акселератора. Немного помучившись с рычагом переключения передач, большинство людей может даже оценить различия между ручной и автоматической коробками передач — это становится возможным благодаря получению четкого представления об общем родительском классе этих объектов — системе передач.

В процессе использования автомобилей люди постоянно взаимодействуют с их инкапсулированными характеристиками. Педали тормоза и газа скрывают невероятную сложность соответствующих объектов за настолько простым интерфейсом, что управлять этими объектами можно простым нажатием ступней педали! Конкретная реализация дви-

гателя, тип тормозов и размер шин не оказывают никакого влияния на способ взаимодействия с определением класса педалей.

Последний атрибут, полиморфизм, четко отражает способность компаний-изготовителей автомобилей предлагать широкое множество вариантов, по сути, одного и того же средства передвижения. Например, 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. В любом случае нужно будет жать на педаль тормоза, чтобы остановиться, вращать руль, чтобы повернуть, и жать педаль акселератора, чтобы автомобиль двигался. Один и тот же интерфейс может применяться для управления множеством различных реализаций.

Как видите, именно совместное применение инкапсуляции, наследования и полиморфизма позволяет преобразовать отдельные детали в объект, который мы называем автомобилем. Сказанное применимо также к компьютерным программам. За счет применения объектно-ориентированных принципов различные части сложной программы могут быть собраны воедино, образуя надежную, пригодную для обслуживания программу.

Как было отмечено в начале этого раздела, каждая Java-программа является объектно-ориентированной. Или, точнее, каждая Java-программа использует инкапсуляцию, наследование и полиморфизм. Хотя на первый взгляд может показаться, что не все эти свойства используются в приведенных в остальной части этой главы и нескольких последующих главах коротких примерах, тем не менее, они в них присутствуют. Как вы вскоре убедитесь, многие функции, предоставляемые языком Java, являются составной частью его встроенных библиотек классов, в которых интенсивно применяются инкапсуляция, наследование и полиморфизм.

Первый пример простой программы

Теперь, когда мы ознакомились с исходными положениями объектно-ориентированного фундамента Java, рассмотрим несколько реальных программ, написанных на этом языке. Мы начнем с компиляции и запуска представленного в этом разделе короткого примера программы. Вы убедитесь, что эта задача несколько более трудоемкая, чем может казаться.

```
/*
   Это простая программа Java.
   Назовите этот файл "Example.java".
*/
class Example {
   // Программа начинается с обращения к main().
      public static void main(String args[]) {
      System.out.println("Простая Java-программа.");
   }
}
```

На заметку! *Последующее описание соответствует стандарту комплекта разработчика Java SE 6 Developer's Kit (JDK 6), поставляемому компанией Sun Microsystems. При использовании другой среды разработки Java компиляция и выполнение программ могут требовать выполнения другой процедуры. Для получения необходимых сведений обратитесь к документации используемого компилятора.*

Ввод кода программы

Для большинства языков программирования имя файла, который содержит исходный код программы, не имеет значения. Однако в Java это не так. Прежде всего, следует твердо усвоить, что присваиваемое исходному файлу имя очень важно. В данном случае именем исходного файла должно быть `Example.java`. И вот почему.

В Java исходный файл официально называется *модулем компиляции*. Он представляет собой текстовый файл, который содержит определения одного или более классов. Компилятор Java требует, чтобы исходный файл имел расширение `.java`.

Как видно из кода программы, именем определенного программой класса является также `Example`. И это не случайно. В Java весь код должен размещаться внутри класса. В соответствии с принятым соглашением имя этого класса должно совпадать с именем файла, содержащего программу. Необходимо также, чтобы употребление строчных и прописных букв имени файла соответствовало их употреблению в имени класса.

Это обусловлено тем, что Java-код чувствителен к регистру символов. Пока что соглашение о соответствии имен файлов и имен классов может казаться произвольным. Однако оно упрощает поддержку и организацию программ.

Компиляция программы

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

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

Компилятор `javac` создаст файл `Example.class`, содержащий байт-кодовую версию программы. Как мы уже отмечали, байт-код Java — это промежуточное представление программы, содержащее инструкции, которые будет выполнять виртуальная машина Java. Следовательно, результат работы компилятора `javac` не является непосредственно исполняемым кодом.

Чтобы действительно выполнить программу, необходимо воспользоваться программой запуска приложений Java, которая носит имя `java`. При этом ей потребуется передать имя класса `Example` в качестве аргумента командной строки, как показано в следующем примере:

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

При выполнении программы на экране отобразится следующий вывод:

```
Простая Java-программа.
```

В процессе компиляции исходного кода каждый отдельный класс помещается в собственный выходной файл, названный по имени класса и получающий расширение `.class`. Именно поэтому исходным файлам Java целесообразно присваивать имена, совпадающие с именами классов, которые они содержат — имя исходного файла будет совпадать с именем файла с расширением `.class`. При запуске `java` описанным способом в командной строке в действительности указывают имя класса, который нужно выполнить. Программа будет автоматически искать файл с указанным именем и расширением `.class`. Если программа найдет файл, она выполнит код, содержащийся в указанном классе.

Более подробное рассмотрение первого примера программы

Хотя программа `Example.java` достаточно коротка, с ней связано несколько важных особенностей, характерных для всех Java-программ. Давайте рассмотрим каждую часть этой программы более подробно. Программа начинается со следующих срок:

```
/*
  Это простая программа Java.
  Назовите этот файл "Example.java".
*/
```

Этот фрагмент кода — *комментарий*. Подобно большинству других языков программирования, Java позволяет вставлять примечания в исходный файл программы. Компилятор игнорирует содержимое комментариев. Эти комментарии служат описанием или пояснением действий программы для любого, кто просматривает исходный код. В данном случае комментарий описывает программу и напоминает, что исходный файл должен быть назван `Example.java`. Конечно, в реальных приложениях комментарии служат главным образом для пояснения работы отдельных частей программы или действий, выполняемых отдельной функцией.

В Java поддерживаются три стиля комментариев. Комментарий, приведенный в начале программы, называют *многострочным комментарием*. Этот тип комментария должен начинаться с символов `/*` и заканчиваться символами `*/`. Весь текст, помещенный между этими двумя символами комментария, компилятором игнорируется. Как следует из его названия, многострочный комментарий может содержать несколько строк.

Следующая строка программы имеет такой вид:

```
class Example {
```

В этой строке ключевое слово `class` используется для объявления о том, что выполняется определение нового класса. `Example` — это *идентификатор*, являющийся именем класса. Все определение класса, в том числе его членов, будет размещаться между открывающей (`{`) и закрывающей (`}`) фигурными скобками. Пока мы не станем подробно останавливаться на рассмотрении особенностей реализации класса. Отметим только, что в среде Java все действия программы осуществляются внутри класса. В этом состоит одна из причин того, что все Java-программы (по крайней мере, частично) являются объектно-ориентированными.

Следующая строка программы — *однострочный комментарий*:

```
// Программа начинается с обращения к main().
```

Это второй тип комментариев, поддерживаемый языком Java. *Однострочный комментарий* начинается с символов `//` и завершается концом строки. Как правило, программисты используют многострочные комментарии для вставки длинных примечаний, а однострочные комментарии — для помещения коротких, построчных описаний. Третий тип комментариев — *комментарий документации* — будет рассмотрен далее в этой главе в разделе “Комментарии”.

Следующая строка кода выглядит так:

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

Она начинается с метода `main()`. Как видно из предшествующего ей комментария, выполнение программы начинается с этой строки. Выполнение всех Java-приложений начинается с вызова метода `main()`. Пока мы не можем подробно раскрыть смысл каждой части этой строки, поскольку для этого требуется четкое представление о подходе

Java к инкапсуляции. Однако, поскольку эта строка кода присутствует в большей части примеров первой части настоящей книги, давайте кратко рассмотрим значение каждой ее части.

Ключевое слово `public` — *спецификатор доступа*, который позволяет программисту управлять видимостью членов класса. Когда члену класса предшествует ключевое слово `public`, этот член доступен коду, расположенному вне класса, в котором определен данный член. (Противоположное ему по действию ключевое слово — `private`, которое препятствует использованию члена класса кодом, определенным вне его класса.) В данном случае метод `main()` должен быть определен как `public`, поскольку при запуске программы он должен вызываться кодом, определенным вне его класса. Ключевое слово `static` позволяет вызывать метод `main()` без конкретизации экземпляра класса. Это необходимо потому, что метод `main()` вызывается виртуальной машиной Java до создания каких-либо объектов. Ключевое слово `void` просто сообщает компилятору, что метод `main()` не возвращает никаких значений. Как будет показано в дальнейшем, методы могут также возвращать значения. Не беспокойтесь, если все сказанное кажется несколько сложным. Все эти концепции подробно рассматриваются в последующих главах.

Как было отмечено, `main()` — метод, вызываемый при запуске Java-приложений. Необходимо помнить, что язык Java чувствителен к регистру символов. Следовательно, строка `Main` не эквивалентна строке `main`. Важно понимать, что компилятор Java будет выполнять компиляцию классов, не содержащих метод `main()`. Но программа запуска приложений (`java`) не имеет средств запуска таких классов. Поэтому, если бы вместо `main` мы ввели `Main`, компилятор все равно выполнил бы компиляцию программы. Однако программа запуска приложений `java` сообщила бы об ошибке, поскольку не смогла бы найти метод `main()`.

Для передачи любой информации, необходимой методу, служат переменные, указываемые в скобках, которые следуют за именем метода. Эти переменные называют *параметрами*. Если для данного метода никакие параметры не требуются, следует указывать пустые скобки. Метод `main()` содержит только один параметр, но достаточно сложный. Параметр `String args[]` объявляет параметр `args`, который представляет собой массив экземпляров класса `String`. (*Массивы* — это коллекции аналогичных объектов.) Объекты типа `String` хранят символьные строки. В данном случае параметр `args` принимает любые аргументы командной строки, присутствующие во время выполнения программы. В данной программе эта информация не используется, но в других программах, рассмотренных позже, она будет применяться.

Последним символом строки является символ фигурной скобки (`{`). Он обозначает начало тела метода `main()`. Весь код, образующий метод, будет располагаться между открывающей и закрывающей фигурными скобками метода.

Еще один важный момент: метод `main()` служит всего лишь началом программы. Сложная программа может включать десятки классов, только один из которых должен содержать метод `main()`, чтобы выполнение было возможным. Аплеты — Java-программы, внедренные в Web-браузеры, вообще не используют метод `main()`, поскольку в них применяются другие средства запуска выполнения аплетов.

Следующая строка кода приведена ниже. Обратите внимание, что она помещена внутри метода `main()`:

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

Эта строка выводит на экран строку текста `"Простая Java-программа."`, за которой следует новая строка. В действительности вывод выполняется встроенным методом `println()`. В данном случае метод `println()` отображает переданную ему строку. Как

будет показано, этот метод можно применять для отображения также и других типов информации. Строка начинается с идентификатора `System.out`. Хотя он слишком сложен, чтобы его можно было подробно пояснить на данном этапе, если говорить вкратце, `System` — это предопределенный класс, который предоставляет доступ к системе, а `out` — выходной поток, связанный с консолью.

Как легко догадаться, в реальных программах и аплетах Java консольный вывод (и ввод) используются не очень часто. Поскольку большинство современных компьютерных сред по своей природе являются оконными и графическими, в большинстве случаев консольный ввод-вывод применяется в простых служебных и демонстрационных программах. Позднее мы рассмотрим другие способы генерирования вывода с помощью Java. А пока продолжим применять методы консольного ввода-вывода.

Обратите внимание, что оператор `println()` завершается символом точки с запятой. В Java все операторы заканчиваются этим символом. Причина отсутствия символа точки с запятой в конце остальных строк программы в том, что с технической точки зрения они не являются операторами.

Первый символ `}` завершает метод `main()`, а последний — определение класса `Example`.

Второй пример короткой программы

Вероятно, ни одна другая концепция не является для языка программирования столь важной, как концепция переменных. Как вы, вероятно, знаете, *переменная* — это именованная ячейка памяти, которой может быть присвоено значение внутри программы. Во время выполнения программы значение переменной может изменяться. Следующая программа демонстрирует способы объявления переменной и присвоения ей значения. Она иллюстрирует также некоторые новые аспекты консольного вывода. Как следует из комментариев в начале программы, этому файлу нужно присвоить имя `Example2.java`.

```
/*
   Это еще один короткий пример.
   Назовите этот файл "Example2.java".
*/
class Example2 {
   public static void main(String args[]) {
      int num; // эта строка объявляет переменную по имени num

      num = 100; // эта строка присваивает переменной num значение, равное 100

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

      num = num * 2;

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

При запуске этой программы на экране отобразится следующий вывод:

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

Рассмотрим генерацию этого вывода подробнее. Первая, еще не знакомая читателям, строка этой программы:

```
int num; // эта строка объявляет переменную по имени num
```

Она объявляет целочисленную переменную num. Java (подобно большинству других языков) требует, чтобы переменные были объявлены до их использования.

Ниже приведена общая форма объявления переменных:

```
тип имя_переменной;
```

В этом объявлении *тип* указывает тип объявляемой переменной, а *имя_переменной* — имя переменной. Если нужно объявить несколько переменных заданного типа, можно использовать разделенный запятыми список имен переменных. Java определяет несколько типов данных, в том числе целочисленный, символьный и с плавающей точкой. Ключевое слово int указывает целочисленный тип. В приведенном примере программы строка

```
num = 100; // эта строка присваивает переменной num значение, равное 100
```

присваивает переменной num значение 100. В Java символом операции присваивания служит одиночный знак равенства.

Следующая строка кода выводит значение переменной num, которому предшествует текстовая строка "Это переменная num:".

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

В этом операторе знак плюса вызывает дописывание значения переменной num в конец предшествующей строки и вывод результирующей строки. (В действительности значение переменной num вначале преобразуется из целочисленного в строковый эквивалент, а затем объединяется с предшествующей строкой. Подробнее этот процесс описан в последующих разделах книги.) Этот подход можно обобщить. Используя символ операции +, внутри одного оператора println() можно объединять любое необходимое количество строк.

Следующая строка кода присваивает переменной num значение этой переменной, умноженное на 2. Как и в большинстве других языков, в Java символ операции * служит для указания операции умножения. После выполнения этой строки кода переменная num будет содержать значение, равное 200.

Следующие две строки программы выглядят так:

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

В них выполняется несколько новых действий. Во-первых, метод print() используется для отображения строки "Значение переменной num * 2 равно". За этой строкой *не* следует символ новой строки. То есть следующий генерируемый вывод будет отображаться в той же строке. Метод print() полностью подобен методу println(), за исключением того, что после каждого вызова он не генерирует символ новой строки. Теперь рассмотрим вызов метода println(). Обратите внимание, что имя переменной num используется само по себе. И print(), и println() могут применяться для вывода значений любых встроенных типов Java.

Два управляющих оператора

Хотя управляющие операторы подробно рассматриваются в главе 5, в этой главе мы кратко рассмотрим два управляющих оператора, чтобы их можно было использовать в примерах программ, приведенных в главах 3 и 4. Кроме того, они послужат хорошей иллюстрацией важного аспекта Java — блоков кода.

Оператор `if`

Работа оператора `if` в Java во многом аналогична работе оператора `IF` любого другого языка. Более того, его синтаксис полностью идентичен синтаксису операторов `if` в языках C, C++ и C#. Простейшая форма этого оператора выглядит следующим образом:

```
if(условие) оператор;
```

Здесь, *условие* — булевское выражение. Если *условие* истинно, оператор выполняется. Если *условие* ложно, оператор пропускается. Рассмотрим следующий пример:

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

В данном случае, если переменная `num` содержит значение, которое меньше 100, условное выражение истинно, и программа выполнит метод `println()`. Если переменная `num` содержит значение, которое больше или равно 100, программа пропустит метод `println()`.

Как будет показано в главе 4, в Java определен полный набор операций сравнения, которые могут быть использованы в условном выражении. Некоторые из них перечислены в табл. 2.1.

Таблица 2.1. Некоторые операции сравнения

Операция	Значение
<	Меньше
>	Больше
==	Равно

Обратите внимание, что символом проверки равенства служит двойной знак равенства.

Ниже приведен пример программы, иллюстрирующий применение оператора `if`.

```
/*
  Демонстрирует применение оператора if.
  Назовите этот файл "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 равна y");

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

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

Эта программа генерирует следующий вывод:

```
x меньше y
x теперь равна y
x теперь больше y
```

Обратите внимание на еще одну особенность этой программы. Строка

```
int x, y;
```

объявляет две переменных x и y, используя при этом разделенный запятой список.

Цикл for

Как вам, возможно, известно из уже имеющегося опыта программирования, операторы цикла — важная составная часть практически любого языка программирования. Java — не исключение в этом отношении. Фактически, как будет показано в главе 5, Java поддерживает обширный набор конструкций циклов. И, вероятно, наиболее универсальный из них — цикл for. Простейшая форма этого цикла имеет вид

```
for(начальное_значение; условие; приращение) оператор;
```

В этой наиболее часто встречающейся форме параметр *начальное_значение* определяет начальное значение управляющей переменной цикла. *Условие* — это булевское выражение, которое проверяет значение управляющей переменной цикла. Если результат проверки истинен, выполнение цикла for продолжается. Если он ложен, выполнение цикла прекращается. Выражение *приращение* определяет изменение управляющей переменной на каждой итерации цикла. Ниже показан пример короткой программы, иллюстрирующий применение цикла for.

```
/*
  Демонстрирует применение цикла for.

  Назовите этот файл "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 генерирует следующий вывод:

```
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 этом примере x — управляющая переменная цикла. В параметре инициализации цикла ей присвоено начальное значение, равное нулю. В начале каждой итерации (включая первую) выполняется проверка условия x<10. Если результат этой проверки истинен, программа выполняет оператор println(), а затем итерационную часть цикла. Процесс продолжается до тех пор, пока результат проверки условия не станет ложным.

Следует отметить, что в профессионально написанных Java-программах вы почти никогда не встретите итерационную часть цикла в том виде, какой она имеет в приведенном примере. То есть операторы вроде следующего встречаются весьма редко:

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

Это объясняется тем, что Java предоставляет специальную более эффективную операцию инкремента значения. Символом этой операции является ++ (два последовательных символа плюса). Операция инкремента значения увеличивает значение операнда на единицу. Используя эту операцию, предшествующий оператор можно было бы переписать следующим образом:

```
x++;
```

Таким образом, как правило, цикл for предшествующей программы будет иметь примерно такой вид:

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

Можете проверить выполнение этого цикла. Вы убедитесь, что он работает точно так же, как в предшествующем примере.

Java предоставляет также операцию декремента значений, символом которой служит --. Эта операция уменьшает значение операнда на единицу.

Использование блоков кода

Java позволяет группировать два и более оператора в *блоки кода*, называемые также *кодовыми блоками*. Это выполняется путем помещения операторов в фигурные скобки. Сразу после создания блок кода становится логическим модулем, который можно использовать в тех же местах, что и отдельный оператор. Например, блок может служить в качестве цели для операторов if и for. Рассмотрим следующий оператор if:

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

В этом примере, если x меньше y, программа выполнит оба оператора, расположенные внутри блока. Таким образом, оба оператора внутри блока образуют логический модуль, и выполнение одного оператора невозможно без одновременного выполнения и второго. Основная идея этого подхода состоит в том, что во всех случаях, когда требуется логически связать два или более оператора, это делается посредством создания блока.

Рассмотрим еще один пример. В следующей программе блок кода использован в качестве целевого модуля цикла for.

```
/*
   Демонстрирует использование блока кода.

   Назовите этот файл "BlockTest.java"
*/
```

```
class BlockTest {
   public static void main(String args[]) {
      int x, y;

      y = 20;

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

Эта программа генерирует следующий вывод:

```
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 цель цикла `for` — блок кода, а не единственный оператор. Таким образом, при каждой итерации цикла программа будет выполнять три оператора, помещенные внутрь блока. Естественно, об этом свидетельствует и генерируемый программой вывод.

Как будет показано в последующих главах книги, блоки кода обладают дополнительными свойствами и применением. Однако их основное назначение — создание логически неразрывных модулей кода.

Вопросы лексики

Теперь, когда читатели ознакомились с несколькими короткими Java-программами, пора более формально описать основные элементы языка. Программы на Java представляют собой коллекцию пробелов, идентификаторов, констант, комментариев, операций, разделителей и ключевых слов. Операции рассматриваются в следующей главе. А остальные элементы описаны в последующих разделах этой главы.

Пробел

Java — язык свободной формы. Это означает, что при написании программы не нужно следовать никаким специальным правилам в отношении отступов. Например, программу `Example` можно было бы записать в виде одной строки или любым другим способом. Единственное обязательное требование — наличие, по меньшей мере, одного пробела между всеми лексемами, которые еще не разграничены символом операции или разделителем. В Java пробелами являются символы пробела, табуляции или символы новой строки.

Идентификаторы

Идентификаторы используются для идентификации имен классов, методов и переменных. Идентификатором может служить любая последовательность строчных и прописных букв, цифр или символов подчеркивания и символов доллара. Идентификаторы не должны начинаться с цифры, чтобы компилятор не путал их с числовыми константами. Повторим еще раз, что Java чувствителен к регистру символов, и поэтому `VALUE` и `Value` — различные идентификаторы. Ниже приведено несколько примеров допустимых идентификаторов.

```
AvgTemp
count
a4
$test
this_is_ok
```

Следующие имена переменных являются недопустимыми:

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

Константы

В Java постоянное значение создается посредством его *литерального* представления. Например, ниже показано несколько констант:

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

Первая константа указывает целочисленное значение, следующая — значение с плавающей точкой, третья — символьную константу, а последняя — строковое значение. Константу можно использовать везде, где допустимо использование значений данного типа.

Комментарии

Как уже было отмечено, в Java определены три типа комментариев. Два из них мы уже встречали: однострочные и многострочные. Третий тип называется *комментарием документации*. Этот тип комментариев используется для создания HTML-файла документации программы. Комментарий документации начинается с символов `/**` и заканчивается символами `*/`. Подробнее комментарии документации описаны в приложении А.

Разделители

Java допускает применение нескольких символов в качестве разделителей. Чаще всего в качестве разделителя используется точка с запятой. Как вы уже видели, она применяется для завершения строк операторов. Допустимые символы-разделители описаны в табл. 2.2.

Таблица 2.2. Допустимые символы-разделители

Символ	Название	Назначение
( )	Круглые скобки	Используются для передачи списков параметров в определениях и вызовах методов. Их применяют также для определения приоритета в выражениях, указания выражений в управляющих операторах и указания преобразования типов.
{ }	Фигурные скобки	Используются для указания значений автоматически инициализируемых массивов. Их применяют также для определения блоков кода, классов, методов и локальных областей определения.
[ ]	Квадратные скобки	Используются для объявления типов массивов, а также при разыменовании значений массивов.
;	Точка с запятой	Завершает операторы.
,	Запятая	Разделяет последовательные идентификаторы в объявлениях переменных. Этот символ-разделитель используют также для создания цепочек операторов внутри оператора `for`.
.	Точка	Используется для разделения 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). Эти ключевые слова, в сочетании с синтаксисом операций и разделителями, образуют основу языка Java. Их нельзя использовать в качестве 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

Ключевые слова `const` и `goto` зарезервированы, но не используются. На начальном этапе разработки Java еще несколько ключевых слов были зарезервированы для возможного использования в будущем. Однако в настоящее время спецификация языка Java определяет только ключевые слова, перечисленные в табл. 2.3.

Кроме ключевых слов, в Java зарезервированы также слова `true`, `false` и `null`. Они представляют значения, определенные спецификацией Java. Их нельзя использовать в качестве 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 уже говорилось, эти методы являются членами класса `System`, который представляют собой стандартный Java-класс, автоматически включаемый в программы. В более широком смысле среда Java построена на основе нескольких встроенных библиотек классов, содержащих множество встроенных методов, которые предоставляют поддержку выполнения таких задач, как ввод-вывод, обработка строк, сетевая обработка и обработка графики. Стандартные классы предлагают также поддержку оконного вывода. Таким образом, в целом среда Java представляет собой сочетание самого языка Java и его стандартных классов. Как будет показано, библиотеки классов предоставляют большую часть функциональных возможностей, обеспечиваемых средой Java. Действительно, в определенной степени стать программистом на Java означает научиться использовать стандартные классы Java. В главах и разделах первой части этой книги мы приводим описание различных элементов стандартных библиотечных классов и методов по мере необходимости, а во второй части библиотеки классов описаны более подробно.

3
ГЛАВА

Типы данных, переменные и массивы

В этой главе рассмотрены три из наиболее важных элементов Java: типы данных, переменные и массивы. Как и все современные языки программирования, Java поддерживает несколько типов данных. Эти типы можно применять для объявления переменных и создания массивов. Как будет показано, подход к использованию этих компонентов, примененный в Java, прост, эффективен и целостен.

Java — строго типизированный язык

Прежде всего, важно уяснить, что Java — строго типизированный язык. Действительно, в определенной степени безопасность и надежность Java-программ обусловлена именно этим обстоятельством. Давайте разберемся, что это означает. Во-первых, каждая переменная обладает типом, каждое выражение имеет тип, и каждый тип строго определен. Во-вторых, все присваивания, как явные, так и посредством передачи параметров в вызовах методов, проверяются на соответствие типов. В Java отсутствуют какие-либо средства автоматического приведения или преобразования конфликтующих типов, как это имеет место в некоторых языках. Компилятор Java проверяет все выражения и параметры на предмет совместимости типов. Любые несоответствия типов являются ошибками, которые должны быть исправлены до завершения компиляции класса.

Элементарные типы

Java определяет восемь *элементарных* типов данных: `byte`, `short`, `int`, `long`, `char`, `float`, `double` и `boolean`.

Часто элементарные типы называют также *простыми* типами, и в этой книге мы будем использовать оба эти термина. Элементарные типы можно разделить на четыре группы.

- Целочисленные. Эта группа включает в себя типы `byte`, `short`, `int` и `long`, которые представляют точные целые числа со знаком.

- Числа с плавающей точкой. Эта группа включает в себя типы `float` и `double`, которые представляют числа, определенные с точностью до определенного десятичного знака.
- Символы. Эта группа включает в себя тип `char`, которая представляет символы символьного набора, такие как буквы и цифры.
- Булевские значения. Эта группа включает в себя тип `boolean` — специальный тип, предназначенный для представления значений типа истинно/ложно.

Эти типы можно использовать в том виде, как они определены, или же для создания собственных типов классов. Таким образом, они служат основой для всех других типов данных, которые могут быть созданы.

Элементарные типы представляют одиночные значения, а не сложные объекты. Хотя во всех других отношениях Java — полностью объектно-ориентированный язык, элементарные типы данных таковыми не являются. Они аналогичны простым типам, которые можно встретить в большинстве других не объектно-ориентированных языков. Эта особенность обусловлена стремлением обеспечить максимальную эффективность. Превращение элементарных типов в объекты привело бы к слишком большому снижению производительности.

Элементарные типы определены так, чтобы они обладали явной областью допустимых значений и математически строгим поведением. Языки вроде C и C++ допускают варьирование размеров целочисленных переменных в зависимости от требований среды выполнения. Однако Java отличается в этом отношении. В связи с требованием переносимости, предъявляемым к Java-программам, все типы данных обладают строго определенной областью допустимых значений. Например, независимо от конкретной платформы, значения типа `int` всегда являются 32-битными. Это позволяет создавать программы, которые гарантированно будут выполняться в любой машинной архитектуре *без специального переноса*. Хотя в некоторых средах строгое указание размера целых чисел может приводить к незначительному снижению производительности, оно абсолютно необходимо для обеспечения переносимости программ.

Рассмотрим каждый и типов данных.

Целочисленные значения

Java определяет четыре целочисленных типа: `byte`, `short`, `int` и `long`. Все эти типы представляют значения со знаком — положительные и отрицательные. Java не поддерживает только положительные целочисленные значения без знака. Многие другие языки программирования поддерживают целочисленные значения как со знаком, так и без знака. Однако разработчики Java посчитали целочисленные значения без знака ненужными. В частности, они решили, что концепция *значений без знака* использовалась, в основном, для указания поведения *старшего бита*, который определяет *знак* целочисленного значения. Как будет показано в главе 4, в Java управление значением старшего бита осуществляется иначе — посредством применения специальной операции “сдвига вправо без учета знака”. Тем самым потребность в целочисленном типе без знака была исключена.

Ширина целочисленного типа представляет не занимаемый объем памяти, а скорее *поведение*, определяемое им для переменных и выражений этого типа. Среда времени выполнения Java может использовать любой размер, до тех пор, пока типы ведут себя объявленным образом. Как показано в табл. 3.1, ширина и область допустимых значений этих целочисленных типов изменяются в широких пределах.

Таблица 3.1. Ширина и область допустимых значений целочисленных типов

Имя	Ширина	Область допустимых значений
long	64	от –9223372036854775808 до 9223372036854775807
int	32	от –2147483648 до 2147483647
short	16	от –32768 до 32767
byte	8	от –128 до 127

Теперь рассмотрим каждый из типов целочисленных значений.

byte

Наименьший по размеру целочисленный тип — byte. Это 8-битный тип со знаком с областью допустимых значений от –128 до 127. Переменные типа byte особенно полезны при работе с потоком данных, поступающих из сети или файла. Они полезны также при манипулировании необработанными двоичными данными, которые могут не быть непосредственно совместимыми с другими встроенными типами Java.

Для объявления переменных типа byte служит ключевое слово byte. Например, в следующей строке объявлены две переменных типа byte, названные b и c:

```
byte b, c;
```

short

short — 16-битный тип со знаком. Он имеет область допустимых значений от –32768 до 32767. Вероятно, этот тип используется в Java наименее часто. Ниже приведено несколько примеров объявления переменных типа short:

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

int

Наиболее часто используемым целочисленным типом является int. Это 32-битный тип со знаком, который имеет область допустимых значений от –2147483648 до 2147483647. Кроме других применений переменные типа int часто применяются для управления циклами и индексирования массивов. Хотя на первый взгляд может показаться, что использование типов byte или short эффективнее использования типа int в ситуациях, когда не требуется более широкий допустимый диапазон значений, предоставляемый последним, в действительности это не всегда так. Это обусловлено тем, что при указании значений типа byte и short в выражениях их тип повышается до int при вычислении выражения. (Повышение типа описано в этой главе позже.) Поэтому часто тип int наиболее подходит для работы с целочисленными значениями.

long

long — 64-битный тип со знаком, удобный в тех ситуациях, когда длина типа int недостаточна для хранения требуемого значения. Область допустимых значений типа long достаточно велика. Это делает его удобным для работы с большими целыми числами.

Например, ниже приведен пример программы, которая вычисляет количество миль, проходимых лучом света за указанное число дней.

```
// Вычисление расстояния, проходимого светом,
// с применением переменных типа long.
class Light {
  public static void main(String args[]) {
    int lightspeed;
    long days;
    long seconds;
    long distance;

    // приблизительная скорость света в милях за секунду
    lightspeed = 186000;

    days = 1000; // указание количества дней

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

    distance = lightspeed * seconds;   // вычисление расстояния

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

Эта программа генерирует следующий вывод:

```
За 1000 дней свет пройдет около 16070400000000 миль.
```

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

Типы с плавающей точкой

Числа с плавающей точкой, называемые также *действительными* числами, используются при вычислении выражений, которые требуют получение результата с точностью до определенного десятичного знака. Например, такие вычисления, как вычисление квадратного корня или трансцендентных функций вроде синуса или косинуса, приводят к результату, который требует применения типа с плавающей точкой. В Java реализован стандартный (в соответствии с IEEE-754) набор типов и операций с плавающей точкой. Существуют два вида типов с плавающей точкой: `float` и `double`, которые соответственно представляют числа одинарной и двойной точности. Их ширина и области допустимых значений описаны в табл. 3.2.

Таблица 3.2. Ширина и область допустимых значений типов с плавающей точкой

Имя	Ширина в битах	Приблизительная область допустимых значений
`double`	64	от 4.9e-324 до 1.8e+308
`float`	32	от 1.4e-045 до 3.4e+038

Рассмотрим каждый из этих типов с плавающей точкой.

float

Тип float определяет *значение одинарной* точности, которое при хранении занимает 32 бита. В некоторых процессорах обработка значений одинарной точности выполняется быстрее и требует в два раза меньше памяти, чем обработка значений двойной точности, но в тех случаях, когда значения либо очень велики, либо очень малы, точность вычислений оказывается недостаточной. Переменные типа float удобны в тех случаях, когда требуется дробная часть значения без особой точности. Например, значение типа float может быть удобно для представления денежных сумм в долларах и центах. Ниже приведен пример объявлений переменных типа float:

```
float hightemp, lowtemp;
```

double

Двойная точность, как следует из ключевого слова double (двойная), требует использования 64 битов для хранения значений. В действительности в некоторых современных процессорах, которые оптимизированы для выполнения математических вычислений с высокой скоростью, обработка значений двойной точности осуществляется быстрее, чем обработка значений одинарной точности. Все трансцендентные математические функции, такие как sin(), cos() и sqrt(), возвращают значения типа double. Применение типа double наиболее рационально, когда требуется сохранение точности множества последовательных вычислений или манипулирование большими числами.

Ниже приведен пример короткой программы, в которой переменные типа double используются для вычисления площади круга.

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

    r = 10.8;       // радиус окружности
    pi = 3.1416;    // pi, приблизительное значение
    a = pi * r * r; // вычисление площади

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

Символы

В Java для хранения символов используется тип данных char. Однако программистам на C/C++ следует помнить, что тип char в Java не эквивалентен типу char в C или C++. В C/C++ char — это целочисленный тип, имеющий ширину 8 битов. В Java это *не так*. Вместо этого в нем для представления символов используется Unicode. *Unicode* определяет международный набор символов, который может представлять все символы, присутствующие во всех известных языках. Он представляет собой унифицированный набор десятков наборов символов, таких как латиница, греческий алфавит, арабский алфавит, кириллица, иврит, японские и тайские иероглифы и множество других. Поэтому для хранения этих символов требуется 16 битов. Таким образом, в Java тип char является 16-битным. Диапазон допустимых значений этого типа — от 0 до 65536. Не существует отрицательных значений типа char. Стандартный набор символов, известный как ASCII,

содержит значения от 0 до 127, а расширенный 8-битный набор символов, ISO-Latin-1 — значения от 0 до 255. Поскольку язык Java предназначен для обеспечения возможности создания программ, применимых во всем мире, использование кодировки Unicode для представления символов вполне обосновано. Конечно, применение Unicode несколько неэффективно для таких языков, как английский, немецкий, испанский или французский, для представления символов которых вполне достаточно 8 битов. Но это та цена, которую приходится платить за переносимость программ во всемирном масштабе.

На заметку! *Более подробную информацию о Unicode можно найти на Web-сайте* `http://www.unicode.org`.

Использование переменных типа `char` демонстрирует следующая программа:

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

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

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

Эта программа отображает следующий вывод:

```
ch1 и ch2: X Y
```

Обратите внимание, что переменной `ch1` присвоено значение 88, являющееся значением ASCII (и Unicode), которое соответствует букве *X*. Как уже было сказано, набор символов ASCII занимает первые 127 значений набора символов Unicode. Поэтому, все "старые трюки", применяемые при работе с символами в других языках, будут работать и в среде Java.

Хотя тип `char` был разработан для хранения символов Unicode, его можно считать также целочисленным типом, пригодным для выполнения арифметических операций. Например, он позволяет выполнять сложение символов или уменьшать значение символьной переменной. Рассмотрим следующую программу:

```
// Символьные переменные ведут себя подобно целочисленным значениям.
class CharDemo2 {
  public static void main(String args[]) {
    char ch1;

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

    ch1++; // увеличение значения ch1 на единицу
    System.out.println("ch1 теперь " + ch1);
  }
}
```

Эта программа генерирует следующий вывод:

```
ch1 содержит X
ch1 теперь Y
```

Вначале программа присваивает переменной ch1 значение *X*. Затем она увеличивает значение переменной ch1 на единицу. В результате хранящееся в переменной значение становится буквой *Y* — следующим символом в последовательности ASCII (и Unicode).

Булевские значения

Java содержит элементарный тип, названный boolean, который предназначен для хранения логических значений. Переменные этого типа могут принимать только одно из двух возможных значений: true (истинно) или false (ложно). Этот тип возвращается всеми операциями сравнения, подобными a < b. Тип boolean обязателен для использования также в условных выражениях, которые управляют такими управляющими операторами, как if и for.

Следующая программа служит примером использования типа boolean:

```
// Демострация использования значений типа boolean.
class BoolTest {
  public static void main(String args[]) {
    boolean b;

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

    // значение типа boolean может управлять оператором if
    if(b) System.out.println("Это выполняется.");

    b = false;
    if(b) System.out.println("Это не выполняется.");

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

Эта программа генерирует следующий вывод:

```
b равна false
b равна true
Это выполняется.
10 > 9 равно true
```

В приведенной программе особый интерес представляют три момента. Во-первых, как видите при выводе значения типа Boolean методом println() на экране отображается строка “true” или “false”. Во-вторых, самого по себе значения переменной типа boolean достаточно для управления оператором if. Вовсе не обязательно записывать оператор if так, как показано ниже:

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

В-третьих, результат выполнения операции сравнения вроде < — значение типа boolean. Именно поэтому выражение 10 > 9 приводит к отображению строки “true”. Более того, выражение 10 > 9 должно быть заключено в дополнительный набор круглых скобок, поскольку операция + обладает более высоким приоритетом, чем операция >.

Более подробное рассмотрение констант

Константы были вскользь упомянуты в главе 2. Теперь, когда встроенные типы формально описаны, рассмотрим константы подробнее.

Целочисленные константы

Целочисленные значения — вероятно наиболее часто используемый тип в типичной программе. Любое целочисленное значение является числовой константой. Примерами могут служить значения 1, 2, 3 и 42. Все они — десятичные значения, описывающие числа с основанием 10. В числовых константах могут использоваться еще два вида представления — *восьмеричное* (с основанием 8) и *шестнадцатеричное* (с основанием 16). В Java восьмеричные значения обозначаются ведущим нулем. Обычные десятичные числа не могут содержать ведущий ноль. Таким образом, внешне вполне допустимое значение 09 приведет к ошибке компиляции, поскольку 9 выходит за пределы диапазона от 0 до 7 допустимых восьмеричных значений. Чаще программисты используют шестнадцатеричное представление чисел, которое явно соответствует словам, размер которых равен 8, 16, 32 и 64 бита, составленным из 8-битных блоков. Шестнадцатеричные значения обозначают ведущим нулем и символом x (`0x` или `0X`). Диапазон допустимых шестнадцатеричных цифр — от 0 до 15, поэтому цифры от 10 до 15 заменяют буквами от *A* до *F* (или от *a* до *f*).

Целочисленные константы создают значение типа `int`, которое в Java является 32-битным целочисленным значением. Поскольку Java — строго типизированный язык, может возникать вопрос, каким образом можно присваивать целочисленную константу одному из других целочисленных типов Java, такому как `byte` или `long`, не вызывая при этом ошибку несоответствия типа. К счастью, с подобными ситуациями легко справиться. Когда значение константы присваивается переменной типа `byte` или `short`, ошибка не генерируется, если значение константы находится в диапазоне допустимых значений целевого типа.

Кроме того, целочисленную константу всегда можно присваивать переменной типа `long`. Однако чтобы указать константу типа `long`, придется явно указать компилятору, что значение константы имеет этот тип. Для этого к константе дописывают строчную или прописную букву *L*. Например, `0x7fffffffffffffL`, или `9223372036854775807L` — наибольшая константа типа `long`. Целочисленное значение можно присваивать типу `char`, если оно лежит в пределах допустимого диапазона этого типа.

Константы с плавающей точкой

Числа с плавающей точкой представляют десятичные значения с дробной частью. Они могут быть выражены в стандартной или научной форме записи. Число в *стандартной форме записи* состоит из целого числа, за которым следуют десятичная точка и дробная часть. Например, `2.0`, `3.14159` и `0.6667` представляют допустимые числа с плавающей точкой в стандартной записи. *Научная форма записи* использует стандартную форму записи числа с плавающей точкой, к которой добавлен суффикс, указывающий степенную функцию числа 10, на которую нужно умножить данное число. Для указания экспоненциальной функции используют символ *E* или *e*, за которым следует десятичное число (положительное или отрицательное). Примерами могут служить `6.022E23`, `314159E-05` и `2e+100`.

По умолчанию в Java константам с плавающей точкой присвоен тип `double`. Чтобы указать константу типа `float`, к ней нужно дописать символ *F* или *f*. Константу типа

double можно также указать явно, дописывая к ней символ *D* или *d*. Но, естественно, это излишне. Используемый по умолчанию тип double занимает в памяти 64 бита, в то время как менее точный тип float требует для хранения только 32 битов.

Булевские константы

Булевские константы очень просты. Значений типа Boolean может существовать только два: true и false. Эти значения не преобразуются ни в какие числовые представления. В Java константа true не равна 1, а константа false не равна 0. В Java эти значения могут быть присвоены только тем переменным, которые объявлены как boolean, или использоваться в выражениях с булевскими операциями.

Символьные константы

В Java символы представляют собой индексы в наборе символов Unicode. Это 16-битные значения, которые могут быть преобразованы в целые значения, и по отношению к которым можно выполнять целочисленные операции, такие как операции сложения и вычитания. Символьные константы указываются внутри пары одинарных кавычек. Все отображаемые символы ASCII можно вводить непосредственно, указывая их в кавычках, например 'a', 'z' и '@'. Для ввода символов, непосредственный ввод которых невозможен, можно использовать несколько управляющих последовательностей, которые позволяют вводить нужные символы, такие как '\'' для символа одинарной кавычки и '\n' для символа новой строки. Существует также механизм для непосредственного ввода значения символа в восьмеричном или шестнадцатеричном виде. Для ввода значения в восьмеричной форме используют символ обратной косой черты, за которым следует трехзначный номер. Например, последовательность '\141' эквивалентна букве 'a'. Для ввода шестнадцатеричного значения применяются символы обратной косой черты и u (\u), за которыми следуют четыре шестнадцатеричные цифры. Например, '\u0061' представляет букву '*a*' из набора символов ISO-Latin-1, поскольку старший байт является нулевым, а 'ua432' — символ японской катаканы. Управляющие последовательности символов перечислены в табл. 3.3.

Таблица 3.3. Управляющие последовательности символов

Управляющая последовательность	Описание
\ddd	Восьмеричный символ (ddd)
\uxxxx	Шестнадцатеричный символ Unicode (xxxx)
\'	Одинарная кавычка
\"	Двойная кавычка
\\	Обратная косая черта
\r	Возврат каретки
\n	Новая строка (этот символ называют также символом перевода строки)
\f	Подача страницы
\t	Табуляция
\b	Возврат на одну позицию (“забой”)

Строковые константы

Указание строковых констант в Java осуществляется так же, как в других языках — путем заключения последовательности символов в пару двойных кавычек. Вот несколько примеров строковых констант:

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

Управляющие символы и восьмеричная/шестнадцатеричная формы записи, определенные для символьных констант, работают точно так же и внутри строковых констант. Важно отметить, что в Java строки должны начинаться и заканчиваться в одной строке. В этом языке отсутствует какой-либо управляющий символ продолжения строки, подобный тем, что имеются в ряде других языков.

На заметку! *Как вы, возможно, знаете, в некоторых языках, включая C/C++, строки реализованы в виде массивов символов. Однако в Java это не так. В действительности строки представляют собой объектные типы. Как будет показано позднее, поскольку в Java строки реализованы в виде объектов, язык предлагает множество мощных и простых в использовании средств их обработки.*

Переменные

Переменная — основной компонент хранения данных в Java-программе. Переменная определяется комбинацией идентификатора, типа и необязательного начального значения. Кроме того, все переменные имеют область определения, которая задает их видимость для других объектов и время существования. Мы рассмотрим эти элементы 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] ...] ;
```

тип — это один из элементарных типов Java либо имя класса или интерфейса. (Типы класса и интерфейса рассмотрены в последующих главах части I этой книги.) *идентификатор* — это имя переменной. Переменной можно присвоить начальное значение (инициализировать ее), указывая знак равенства и значение. Следует помнить, что выражение инициализации должно возвращать значение того же (или совместимого) типа, который указан для переменной. Для объявления более одной переменной указанного типа можно использовать список с разделителями-запятыми.

Несколько примеров объявления переменных различных типов приведено ниже. Обратите внимание, что некоторые объявления осуществляют инициализацию переменных.

```
int a, b, c;          // объявление трех переменных типа int: a, b и c
int d = 3, e, f = 5;  // объявление еще трех переменных типа int c
                      // инициализацией d и f
byte z = 22;          // инициализация переменной z
double pi = 3.14159; // объявление приблизительного значения переменной pi
char x = 'x';         // присваивание значения 'x' переменной x
```

Выбранные имена идентификаторов очень просты и указывают их тип. Java допускает применение любого правильно оформленного идентификатора с любым объявленным типом.

Динамическая инициализация

Хотя в приведенных примерах в качестве начальных значений были использованы только константы, Java допускает динамическую инициализацию переменных посредством любого выражения, допустимого в момент объявления переменной.

Например, ниже приведена короткая программа, которая вычисляет длину гипотенузы прямоугольного треугольника по длинам катетов:

```
// Этот пример демонстрирует динамическую инициализацию.
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);
  }
}
```

Эта программа объявляет три локальные переменные — `a`, `b` и `c`. Две первые, `a` и `b`, инициализируются константами. Однако третья, `c`, инициализируется динамически, принимая значение длины гипотенузы (в соответствии с теоремой Пифагора). Для вычисления квадратного корня аргумента программа использует встроенный метод Java, `sqrt()`, который является членом класса `Math`. В этом примере основной момент состоит в том, что в выражении инициализации можно использовать любые элементы, которые допустимы во время инициализации, в том числе вызовы методов, другие переменные или константы.

Область определения и время существования переменных

До сих пор все использованные в примерах переменные были объявлены в начале метода `main()`. Однако Java допускает объявление переменных внутри любого блока. Как было сказано в главе 2, блок начинается открывающей фигурной скобкой и завершается закрывающей фигурной скобкой. Блок задает *область определения*. Таким образом, при открытии каждого нового блока мы создаем новую область определения. Область определения задает то, какие объекты видимы другим частям программы. Она определяет также время существования этих объектов.

Многие другие языки программирования различают две основных категории областей определения: глобальную и локальную. Однако эти традиционные области определения не очень хорошо вписываются в строгую объектно-ориентированную модель Java. Хотя глобальную область определения и можно задать, в настоящее время такой подход является скорее исключением, нежели правилом. В Java две основных области определения — это определяемая классом и определяемая методом. Но даже это разделение несколько искусственно. Однако поскольку область определения класса обладает несколькими уникальными свойствами и атрибутами, не применимыми к области определения метода, такое разделение имеет определенный смысл. В связи отличиями мы отложим рассмотрение область определения класса (и объявленных внутри нее переменных) до главы 6, в которой описаны классы. А пока рассмотрим только те области определения, которые определяются методом или внутри него.

Определенная методом область определения начинается с его открывающей фигурной скобки. Однако если данный метод обладает параметрами, они также включаются в область определения метода. Хота подробнее параметры рассмотрены в главе 6, пока отметим, что они работают точно так же, как любая другая переменная метода.

Основное правило, которое следует запомнить: переменные, объявленные внутри области определения, не видны (т.е. недоступны) коду, который находится за пределами этой области. Таким образом, объявление переменной внутри области определения ведет к ее локализации и защите от несанкционированного доступа и/или изменений. Действительно, правила обработки области определения — основа инкапсуляции.

Области определения могут быть вложенными. Например, при каждом создании блока кода мы создаем новую, вложенную область определения. В этих случаях внешняя область определения заключает в себя внутреннюю область. Это означает, что объекты, объявленные во внешней области, будут видны коду, определенному во внутренней области. Тем не менее, обратное не верно. Объекты, которые объявлены во внутренней области определения, не будут видны за ее пределами.

Чтобы понять эффект применения вложенных областей определения рассмотрим следующую программу:

```
// Демонстрация области определения блока.
class Scope {
  public static void main(String args[]) {
    int x; // эта переменная известна всему коду внутри метода main

    x = 10;
    if(x == 10) {   // начало новой области определения
      int y = 20;   // известной только этому блоку

      // и x, и y известны в этой области определения.
      System.out.println("x и y: " + x + " " + y);
      x = y * 2;
    }
    // y = 100;     // Ошибка! y не известна в этой области определения

    // переменная x известна и здесь.
    System.out.println("x равна " + x);
  }
}
```

Как видно из комментариев, переменная `x` объявлена в начале области определения метода `main()` и доступна всему последующему коду, находящемуся внутри этого метода. Объявление переменной `y` осуществляется внутри блока `if`. Поскольку блок задает область определения, переменная `y` видна только коду внутри этого блока. Именно поэтому строка `y=100;`, расположенная вне этого блока, помещена в комментарий. Если удалить символ комментария, это приведет к ошибке времени компиляции, поскольку переменная `y` не видна за пределами своего блока. Переменную `x` можно использовать внутри блока `if`, поскольку код внутри блока (т.е. во вложенной области определения) имеет доступ к переменным, которые объявлены внешней областью.

Внутри блока переменные можно объявлять в любом месте, но они становятся допустимыми только после объявления. Таким образом, если переменная объявлена в начале метода, она доступна всему коду внутри этого метода. И наоборот, если переменная объявлена в конце блока, она, по сути, бесполезна, поскольку никакой код не получит к ней доступ. Например, следующий фрагмент кода неправилен, поскольку переменную `count` нельзя использовать до ее объявления:

```
// Этот фрагмент неправилен!
count = 100;     // Стоп! Переменную count нельзя использовать до того,
                 // как она будет объявлена!
int count;
```

Следует запомнить еще один важный нюанс: переменные создаются при входе в их область определения и уничтожаются при выходе из нее. Это означает, что переменная утратит свое значение сразу по выходу из области определения. Следовательно, переменные, которые объявлены внутри метода, не будут хранить свои значения между обращениями к этому методу. Кроме того, переменная, объявленная внутри блока, будет утрачивать свое значение по выходу из блока. Таким образом, время существования переменной ограничено ее областью определения.

Если объявление переменной содержит инициализацию, инициализация переменной будет повторяться при каждом вхождении в блок, в котором она объявлена. Например, рассмотрим приведенную ниже программу.

```
// Демонстрация времени существования переменной.
class LifeTime {
  public static void main(String args[]) {
    int x;

    for(x = 0; x < 3; x++) {
      int y = -1; // y инициализируется при каждом вхождении в блок
      System.out.println("y равна: " + y); // эта строка всегда выводит
                                           // значение -1
      y = 100;
      System.out.println("y теперь равна: " + y);
    }
  }
}
```

Эта программа генерирует следующий вывод:

```
y равна: -1
y теперь равна: 100
y равна: -1
y теперь равна: 100
y равна: -1
y теперь равна: 100
```

Как видите, переменная y повторно инициализируется значением –1 при каждом вхождении во внутренний цикл `for`. Несмотря на то что впоследствии переменной присваивается значение 100, это значение теряется.

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

```
// Компиляция этой программы будет невозможна
class ScopeErr {
  public static void main(String args[]) {
    int bar = 1;
    { // создание новой области определения
      int bar = 2; //Ошибка времени компиляции — переменная bar уже определена!
    }
  }
}
```

Преобразование и приведение типов

Те, кто уже обладает определенным опытом программирования, знают, что достаточно часто программисты присваивают переменной одного типа значение другого. Если оба эти типа совместимы, Java выполнит преобразование автоматически. Например, всегда можно присвоить значение типа `int` переменной типа `long`. Однако не все типы совместимы и, следовательно, не все преобразования типов безоговорочно разрешены. Например, не существует никакого определенного автоматического преобразования `double` в `byte`. К счастью, преобразования между несовместимыми типами выполнять все-таки можно. Для этого необходимо использовать *приведение*, которое выполняет явное преобразование несовместимых типов. Рассмотрим автоматическое преобразование и приведение типов.

Автоматическое преобразование типов в Java

При присваивании типа данных переменной другого типа *автоматическое преобразования типа* выполняется в случае удовлетворения следующих двух условий:

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

При соблюдении этих двух условий выполняется *преобразование с расширением*. Например, тип `int` всегда достаточно велик, чтобы хранить все допустимые значения типа `byte`, поэтому никакие операторы явного приведения типа не требуются.

С точки зрения преобразования с расширением числовые типы, среди которых целочисленный и с плавающей точкой, совместимы друг с другом. Однако не существует автоматических преобразований числовых типов в `char` или `boolean`. Типы `char` и `boolean` также не совместимы и между собой.

Как уже говорилось ранее, Java выполняет автоматическое преобразование типов при сохранении целочисленной константы в переменных типа `byte`, `short`, `long` или `char`.

Приведение несовместимых типов

Хотя автоматическое преобразование типов удобно, оно не в состоянии удовлетворить все потребности. Например, что делать, если нужно присвоить значение типа `int` переменной типа `byte`? Это преобразование не будет выполняться автоматически, поскольку тип `byte` меньше типа `int`. Иногда этот вид преобразования называют *преобразованием с сужением*, поскольку значение явно сужается, чтобы оно могло уместиться в целевом типе.

Чтобы выполнить преобразование между двумя несовместимыми типами, необходимо использовать приведение типов. *Приведение* — это всего лишь явное преобразование типов. Общая форма преобразования имеет вид:

```
(целевой_тип) значение
```

Здесь *целевой_тип* определяет тип, к которому нужно преобразовать указанное значение. Например, следующий фрагмент кода приводит тип `int` к типу `byte`. Если значение целочисленного типа больше допустимого диапазона типа `byte`, оно будет уменьшено до результата деления по модулю (остатка 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 типа: *усечение*. Как вы знаете, целые числа не содержат дробной части. Таким образом, когда значение с плавающей точкой присваивается переменной целочисленного типа, дробная часть отбрасывается. Например, в случае присваивания значения 1,23 целочисленной переменной результирующим значением будет просто 1. Дробная часть — 0,23 — усекается. Конечно, если размер целочисленной части слишком велик, чтобы уместиться в целевом целочисленном типе, значение будет уменьшено до результата деления по модулю на диапазон целевого типа.

Следующая программа демонстрирует ряд преобразований типа, которые требуют приведения:

```
// Демонстрация приведения типов.
class Conversion {
  public static void main(String args[]) {
    byte b;
    int i = 257;
    double d = 323.142;

    System.out.println("\nПреобразование int в byte.");
    b = (byte) i;
    System.out.println("i и b " + i + " " + b);

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

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

Эта программа генерирует следующий вывод:

```
Преобразование int в byte.
i и b 257 1

Преобразование double в int.
d и i 323.142 323

Преобразование double в byte.
d и b 323.142 67
```

Рассмотрим каждое из этих преобразований. Когда значение 257 приводится к типу `byte`, результатом будет остаток от деления 257 на 256 (диапазон допустимых значений типа `byte`), который в данном случае равен 1. Когда значение переменной `d` преобразуется в тип `int`, его дробная часть отбрасывается. 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 повышение типа в выражениях

Кроме операций присваивания определенное преобразование типов может выполняться также в выражениях. Для примера рассмотрим следующую ситуацию. Иногда в выражениях в целях обеспечения необходимой точности промежуточное значение может выходить за пределы допустимого диапазона любого из операндов. Например, рассмотрим следующее выражение:

```
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 решения подобных проблем при вычислении выражений Java автоматически повышает тип каждого операнда `byte` или `short` до `int`. То есть вычисление промежуточного выражения `a*b` выполняется с применением целочисленных значений, а не байтов. Поэтому результат промежуточного выражения 50 * 40, равный 2000, оказывается допустимым, несмотря на то, что и для `a` и для `b` задан тип `byte`.

Хотя автоматическое повышение типа очень удобно, оно может приводить к досадным ошибкам во время компиляции. Например, следующий внешне абсолютно корректный код приводит к возникновению проблемы:

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

Код предпринимает попытку повторного сохранения произведения 50 * 2 — совершенно допустимого значения типа `byte` — в переменной типа `byte`. Однако, поскольку во время вычисления выражения тип операндов был автоматически повышен до `int`, тип результата также был повышен до `int`. Таким образом, теперь результат выражения имеет тип `int`, который не может быть присвоен переменной типа `byte` без приведения типа. Сказанное справедливо даже тогда, когда, как в данном конкретном случае, значение, которое должно быть присвоено, умещается в переменной целевого типа.

В тех случаях, когда последствия переполнения понятны, следует использовать явное приведение типов вроде:

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

которое приводит к правильному значению, равному 100.

Правила повышения типа

В Java определено несколько правил *повышения типа*, применяемых к выражениям. Эти правила следующие: во-первых, тип всех значений `byte`, `short` и `char` повышается до `int`, как было описано в предыдущем разделе. Во-вторых, если один операнд имеет тип `long`, тип всего выражения повышается до `long`. Если один операнд имеет тип `float`, тип всего выражения повышается до `float`. Если любой из операндов имеет тип `double`, типом результата будет `double`.

Следующая программа демонстрирует повышение типа значения одного из операндов к типу второго в каждой операции 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);
  }
}
```

Давайте подробнее рассмотрим повышение типа, выполняемое в следующей строке программы:

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

В первом промежуточном выражении, `f*b`, тип переменной `b` повышается до `float`, и типом результата вычисления этого промежуточного выражения также является `float`. В следующем промежуточном выражении `i/c` тип `c` повышается до `int` и результат вычисления этого выражения — `int`. Затем в выражении `d*s` тип значения `s` повышается до `double`, и все промежуточное выражение получает тип `double`. И, наконец, выполняются операции с этими тремя промежуточными значениями типов `float`, `int` и `double`. Результат сложения значений типов `float` и `int` имеет тип `float`. Затем тип значения разности результирующего значения типа `float` и последнего значения типа `double` повышается до `double`, который и становится окончательным типом результата выражения.

Массивы

Массив — это группа однотипных переменных, ссылка на которые выполняется по общему имени. Java допускает создание массивов любого типа, которые могут иметь одно или больше измерений. Доступ к конкретному элементу массива осуществляется по его индексу. Массивы предлагают удобные средства группирования связанной информации.

На заметку! *Те, кто знаком с языками C/C++, должны быть особенно внимательными. В Java массивы работают иначе, чем в этих языках.*

Одномерные массивы

Одномерные массивы, по сути, представляют собой список однотипных переменных. Чтобы создать массив, вначале необходимо создать переменную массива требуемого типа. Общая форма объявления одномерного массива выглядит следующим образом:

```
тип имя_переменной[];
```

Здесь *тип* задает базовый тип массива. Базовый тип определяет тип данных каждого из элементов, составляющих массив. Таким образом, базовый тип массива определяет

тип данных, которые будет содержать массив. Например, следующий оператор объявляет массив month_days, имеющий тип "массив элементов типа int":

```
int month_days[];
```

Хотя это объявление утверждает, что month_days — массив переменных, в действительности никакого массива еще не существует. Фактически значение массива month_days установлено равным null, которое представляет массив без значений. Чтобы связать month_days с реальным физическим массивом целочисленных значений, необходимо с помощью операции new распределить память и присвоить ее массиву month_days.

Подробнее мы рассмотрим эту операцию в следующей главе, однако она нужна сейчас для выделения памяти под массивы. Общая форма операции new применительно к одномерным массивам выглядит следующим образом:

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

тип определяет тип данных, для которых резервируется память, *размер* указывает количество элементов в массиве, а *переменная_массива* — переменная массива, связанная с массивом. То есть, чтобы использовать операцию new для распределения памяти, потребуется указать тип и количество элементов, для которых нужно зарезервировать память. Элементы массива, для которых память была выделена операцией new, будут автоматически инициализированы нулевыми значениями. Следующий оператор резервирует память для 12-элементного массива целых значений и связывает их с массивом month_days.

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

После выполнения этого оператора month_days будет ссылаться на массив, состоящий из 12 целочисленных значений. При этом все элементы массива будут инициализированы нулевыми значениями.

Подведем итоги: создание массива — двухступенчатый процесс. Во-первых, необходимо объявить переменную нужного типа массива. Во-вторых, с помощью операции new необходимо распределить память для хранения массива и присвоить ее переменной массива. Таким образом, в Java все массивы являются динамически распределяемыми. Если вы еще не знакомы с концепцией динамического распределения памяти, не беспокойтесь. Этот процесс будет подробно описан в последующих главах книги.

Как только массив создан, и память для него распределена, к конкретному элементу массива можно обращаться, указывая его индекс в квадратных скобках. Индексы массива начинаются с нуля. Например, следующий оператор присваивает значение 28 второму элементу массива month_days:

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

Следующая строка кода отображает значение, хранящееся в элементе с индексом, равным 3:

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

Чтобы продемонстрировать весь процесс в целом, приведем программу, которая создает массив числа дней в каждом месяце.

```
// Демонстрация использования 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 количество дней в апреле. Как уже было сказано, в Java индексация элементов массивов начинается с нуля, поэтому количество дней в апреле — `month_days[3]`, или 30.

Объявление переменной массива можно объединять с распределением самого массива, как показано в следующем примере:

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

Именно так обычно и поступают в профессионально написанных Java-программах. Массивы можно инициализировать при их объявлении. Этот процесс во многом аналогичен инициализации простых типов. *Инициализатор массива* — это разделяемый запятыми список выражений, заключенный в фигурные скобки. Запятые разделяют значения элементов массива. Массив будет автоматически создан достаточно большим, чтобы в нем могли уместиться все элементы, указанные в инициализаторе массива. В этом случае использование операции `new` не требуется. Например, чтобы сохранить количество дней каждого месяца, можно использовать следующий код, который создает и инициализирует 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 этой программы она генерирует такой же вывод, как и предыдущая версия.

Система Java выполняет тщательную проверку, чтобы убедиться в том, не была ли случайно предпринята попытка сохранения или ссылки на значения, которые выходят за пределы допустимого диапазона массива. Система времени выполнения Java будет проверять соответствие всех индексов массива допустимому диапазону. Например, система времени выполнения будет проверять соответствие значения каждого индекса допустимому диапазону от 0 до 11 включительно. Попытка обращения к элементам за пределами диапазона массива (указание отрицательных индексов или индексов, которые превышают длину массива) приведет к ошибке времени выполнения.

Приведем еще один пример программы, в которой используется одномерный массив. Эта программа вычисляет среднее значение набора чисел.

```
// Вычисление среднего значения 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 Java *многомерные массивы* представляют собой массивы массивов. Они, как можно догадаться, выглядят и действуют подобно обычным многомерным массивам. Однако, как вы увидите, им присущи и несколько незначительных отличий. При объявлении переменной многомерного массива для указания каждого дополнительного индекса используют отдельный набор квадратных скобок. Например, следующий код объявляет переменную двумерного массива `twoD`.

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

Этот оператор распределяет память для массива размерностью 4×5 и присваивает его переменной `twoD`. Внутренне эта матрица реализована как *массив массивов* значений типа `int`. С точки зрения логической организации этот массив будет выглядеть подобно изображенному на рис. 3.1.

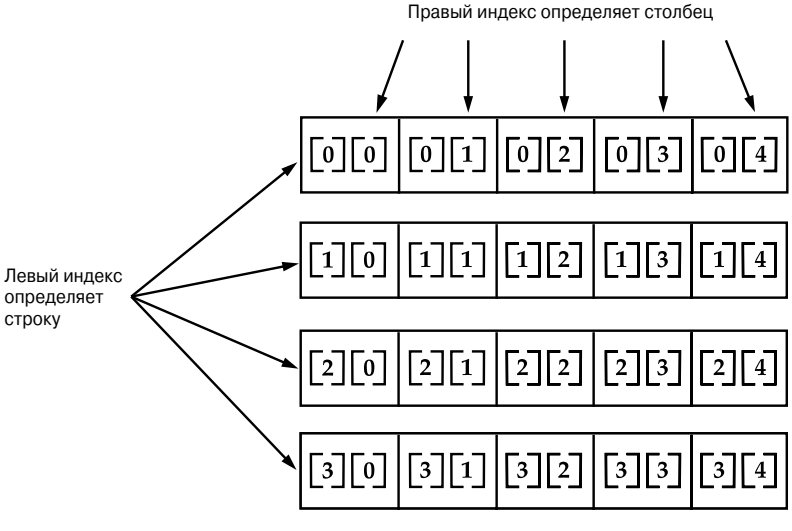

Рис. 3.1. Логическое представление двумерного массива 4×5

Следующая программа нумерует элементы в массиве слева направо, сверху вниз, а затем отображает эти значения.

```
// Демонстрация 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 генерирует следующий вывод:

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

При распределении памяти под многомерный массив необходимо указать память только для первого (левого) измерения. Для каждого из остальных измерений память можно распределять отдельно. Например, следующий код резервирует память для первого измерения массива `twoD` при его объявлении. Распределение 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];
```

Хотя в данной ситуации отдельное распределение памяти для второго измерения массива и не дает никаких преимуществ, в других ситуациях это может быть полезно. Например, при распределении памяти для измерений массива вручную не нужно распределять одинаковое количество элементов для каждого измерения. Как было отмечено ранее, поскольку в действительность многомерные массивы представляют собой массивы массивов, программист полностью управляет длиной каждого массива. Например, следующая программа создает двумерный массив с различными размерами второго измерения.

```
// Резервирование памяти вручную для массива с различными
// размерами 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 генерирует следующий вывод:

```
0
1 2
3 4 5
6 7 8 9
```

Созданный ею массив выглядит, как показано на рис. 3.2.

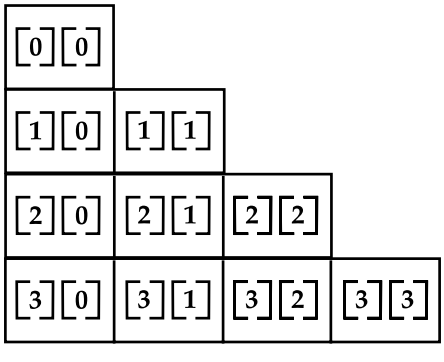

Рис. 3.2. Двумерный массив с различными размерами второго измерения

Использование неоднородных (или нерегулярных) массивов может быть не применимо во многих приложениях, поскольку их поведение отличается от обычного поведения многомерных массивов. Однако в некоторых ситуациях нерегулярные массивы могут оказаться весьма эффективными. Например, нерегулярный массив может быть идеальным решением, если требуется очень большой двумерный 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 набор фигурных скобок. Следующая программа создает матрицу, в которой каждый элемент содержит произведение индексов строки и столбца. Обратите также внимание, что внутри инициализаторов массивов можно применять как значения констант, так и выражения.

```
// Инициализация двумерного массива.
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();
    }
  }
}
```

При выполнении эта программа генерирует следующий вывод:

```
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 использования многомерного массива. Следующая программа создает трехмерный массив размерности 3×4×5. Затем она загружает каждый элемент произведением его индексов. И, наконец, она отображает эти произведения.

```
// Демонстрация 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();
      }
      System.out.println();
    }
  }
}
```

Эта программа генерирует следующий вывод:

```
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 форму синтаксиса:

```
тип[] имя_переменной;
```

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

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

Приведенные ниже два объявления также эквивалентны:

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

Описанная вторая форма объявления удобна для одновременного объявления нескольких массивов. Например, объявление

```
int[] nums, nums2, nums3; // создание трех массивов
```

создает три переменных массивов типа `int`. Оно эквивалентно объявлению

```
int nums[], nums2[], nums3[]; // создание трех массивов
```

Альтернативная форма объявления удобна также при указании массива в качестве возвращаемого типа метода. В этой книге используются обе формы объявлений.

Несколько слов о строках

Как вы, вероятно, заметили, в ходе рассмотрения типов данных и массивов мы не упоминали строки или строковый тип данных. Это связано не с тем, что язык Java не поддерживает этот тип — он его поддерживает. Просто строковый тип данных Java, имеющий имя `String`, не относится к простым типам. Он не является также и просто массивом символов. Тип `String`, скорее, определяет объект, и для понимания его полного описания требуется понимание ряда характеристик объектов. Поэтому мы рассмотрим этот тип в последующих главах книги, после рассмотрения объектов. Однако чтобы читатели могли использовать простые строки в примерах программ, мы сейчас кратко опишем этот тип.

Тип `String` используют для объявления строковых переменных. Можно также объявлять массивы строк. Переменной типа `String` можно присваивать заключенную в кавычки строковую константу. Переменная типа `String` может быть присвоена другой переменной типа `String`. Объект типа `String` можно применять в качестве аргумента оператора `println()`. Например, рассмотрим следующий фрагмент кода:

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

В этом примере `str` — объект типа `String`. Ему присвоена строка `"тестовая строка"`, которая отображается оператором `println()`.

Как будет показано в дальнейшем, объекты тип `String` обладают многими характерными особенностями и атрибутами, которые делают их достаточно мощными и простыми в использовании. Однако в нескольких последующих главах мы будем применять их только в простейшей форме.

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

Опытные программисты на C/C++ знают, что эти языки поддерживают указатели. Однако в настоящей главе мы не разу их не упоминали. Причина этого проста: Java не поддерживает и не разрешает использование указателей. (Точнее говоря, Java не поддерживает указатели, которые доступны и/или могут быть изменены программистом.) Java не разрешает использование указателей, поскольку это позволило бы Java-программам преодолевать защитный барьер между средой выполнения Java и хост-компьютером. (Вспомните, что указателю может быть присвоен любой адрес в памяти — даже те адреса, которые могут находиться вне системы времени выполнения Java.) Поскольку в программах C/C++ указатели используются достаточно интенсивно, их утрата может казаться существенным недостатком Java. В действительности это не так. Среда Java спроектирована так, чтобы до тех пор, пока все действия выполняются в пределах среды выполнения, применение указателей не требовалось, и их использование не дает никаких преимуществ.

4

ГЛАВА

Операции

Язык Java предоставляет среду выполнения множества операций. Большинство их них может быть отнесено к одной из следующих четырех групп: арифметические операции, побитовые операции, операции сравнения и логические операции. В Java также определен ряд дополнительных операций, которые применяются в особых ситуациях. В этой главе описаны все операции Java, за исключением операции сравнения типов `instanceof`, которая рассматривается в главе 13.

Арифметические операции

Арифметические операции используются в математических выражениях так же, как они применяются в алгебре. Арифметические операции перечислены в табл. 4.1.

Таблица 4.1. Арифметические операции в Java

Операция	Описание
`+`	Сложение
`-`	Вычитание (также унарный минус)
`*`	Умножение
`/`	Деление
`%`	Деление по модулю
`++`	Инкремент
`+=`	Сложение с присваиванием
`-=`	Вычитание с присваиванием
`*=`	Умножение с присваиванием
`/=`	Деление с присваиванием
`%=`	Деление по модулю с присваиванием
`--`	Декремент

Операнды арифметических операций должны иметь числовой тип. Арифметические операции нельзя применять к типам `boolean`, но их можно применять к типам `char`, поскольку в Java этот тип, по сути, является поднабором типа `int`.

Основные арифметические операции

Все основные арифметические операции — сложение, вычитание, умножение и деление — действуют по отношению ко всем числовым типам так, как этого можно было бы ожидать. Операция вычитания имеет также унарную (с одним операндом) форму, которая изменяет знак ее единственного операнда. Следует помнить, что в случае применения операции деления к целочисленному типу результат не будет содержать дробного компонента.

Следующий пример простой программы демонстрирует применение арифметических операций. Он иллюстрирует также различие между делением с плавающей точкой и целочисленным делением.

```
// Демонстрация основных арифметических операций.
class BasicMath {
  public static void main(String args[]) {
    // арифметические операции с 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 со 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 отобразится следующий вывод:

```
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 операции деления. Эту операцию можно применять как к типам с плавающей точкой, так и к целочисленным типам. Следующий пример программы демонстрирует применение операции %:

```
// Демонстрация использования 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);
  }
}
```

При выполнении эта программа генерирует следующий вывод:

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

Составные арифметические операции с присваиванием

В Java имеются специальные операции, которые можно являются объединением арифметической операции с операцией присваивания. Как вы, вероятно, знаете, операторы вроде показанного ниже в программах встречаются достаточно часто:

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

В Java этот оператор можно записать следующим образом:

```
a += 4;
```

В этой версии оператора использована *составная операция присваивания* +=. Оба оператора выполняют одно и то же действие: они увеличивают значение переменной a на 4. А вот еще один пример:

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

который можно записать как:

```
a %= 2;
```

В этом случае операция %= вычисляет остаток от деления a/2 и помещает результат обратно в переменную a.

Составные операции с присваиванием существуют для всех арифметических операций с двумя операндами. Таким образом, любой оператор, имеющий форму

```
переменная=переменная операция выражение;
```

можно записать в виде

```
переменная операция = выражение;
```

Составные операции с присваиванием предоставляют два преимущества. Во-первых, они позволяют уменьшить объем вводимого кода, поскольку являются "сокращенным" вариантом соответствующих длинных форм. Во-вторых, их реализация в системе времени выполнения Java эффективнее реализации эквивалентных длинных форм. Поэтому в профессионально написанных Java-программах составные операции с присваиванием будут встречаться очень часто.

Ниже приведен пример программы, который демонстрирует практическое применение нескольких составных операций с присваиванием.

```
// Демонстрация применения 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 создает следующий вывод:

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

Инкремент и декремент

Операции ++ и -- — это операции инкремента и декремента. Они были представлены в главе 2. А в этой главе мы рассмотрим их подробно. Как вы вскоре убедитесь, эти операции обладают рядом особых свойств, которые делают их достаточно интересными. Рассмотрим, что именно делают операции инкремента и декремента.

Операция инкремента увеличивает значение операнда на единицу. Операция декремента уменьшает значение операнда на единицу. Например, следующий оператор:

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

с применением операции инкремента можно записать в таком виде:

```
x++;
```

Аналогично, оператор

```
x = x – 1;
```

эквивалентен оператору

```
x--;
```

Эти операции отличаются тем, что они могут быть записаны как в *постфиксной* форме, когда символ операции следует за операндом, как в приведенных примерах, так и в *префиксной* форме, когда он предшествует операнду. В приведенных примерах применение любой из этих форм не имеет никакого значения. Однако, когда операции инкремента/декремента являются частью более сложного выражения, проявляется внешне незначительное, но важное различие между этими двумя формами. В префиксной форме значение операнда увеличивается или уменьшается до извлечения значения для использования в выражении. В постфиксной форме предыдущее значение извлекается для использования в выражении, и лишь после этого значение операнда изменяется. Например:

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

В этом случае значение y устанавливается равным 43, как и можно было ожидать, поскольку увеличение значения выполняется перед присваиванием значения x переменной y. Таким образом, строка y=++x эквивалентна следующим двум операторам:

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

Однако если операторы записать как

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

значение переменной x извлекается до выполнения операции инкремента, и поэтому значение переменной y равно 42. Конечно, в обоих случаях значение переменной x установлено равным 43. Следовательно, строка y=x++; эквивалентна следующим двум операторам:

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

Следующая программа демонстрирует применение операции инкремента.

```
// Демонстрация применения операции ++.
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);
  }
}
```

Вывод этой программы выглядит следующим образом:

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

Побитовые операции

Java определяет несколько *побитовых операций*, которые могут применяться к целочисленным типам: `long`, `int`, `short`, `char` и `byte`. Эти операции воздействуют на отдельные биты операндов. Они перечислены в табл. 4.2.

Таблица 4.2. Побитовые операции в Java

Операция	Описание
`~`	Побитовая унарная операция NOT (НЕ)
`&`	Побитовое AND (И)
`|`	Побитовое OR (ИЛИ)
`^`	Побитовое исключающее OR
`>>`	Сдвиг вправо
`>>>`	Сдвиг вправо с заполнением нулями
`<<`	Сдвиг влево
`&=`	Побитовое AND с присваиванием
`|=`	Побитовое OR с присваиванием
`^=`	Побитовое исключающее OR с присваиванием
`>>=`	Сдвиг вправо с присваиванием
`>>>=`	Сдвиг вправо с заполнением нулями с присваиванием
`<<=`	Сдвиг влево с присваиванием

Поскольку побитовые операции манипулируют битами в целочисленном значении, важно понимать, какое влияние подобные манипуляции могут оказывать на значение. В частности, важно знать, как среда Java хранит целочисленные значения, и как она представляет отрицательные числа. Поэтому, прежде чем продолжить рассмотрение операций, кратко рассмотрим эти два вопроса.

Все целочисленные типы представляются двоичными числами различной длины. Например, значение типа `byte`, равное 42, в двоичном представлении имеет вид 00101010, в котором каждая позиция представляет степень двух, начиная с 2^0 в крайнем справа бите. Бит в следующей позиции будет представлять 2^1, или 2, следующий — 2^2, или 4, затем 8, 16, 32 и т.д. Таким образом, двоичное представление числа 42 содержит единичные биты в позициях 1, 3 и 5 (начиная с 0, крайней справа позиции). Следовательно, $42=2^1+2^3+2^5=2+8+32$.

Все целочисленные типы (за исключением `char`) — целочисленные типы со знаком. Это означает, что они могут представлять как положительные, так и отрицательные значения. В Java применяется кодирование, называемое двоичным дополнением, при котором отрицательные числа представляются посредством инвертирования всех битов значения (изменения 1 на 0 и наоборот) и последующего добавления 1 к результату. Например, –42 представляется путем инвертирования все битов в двоичном представлении числа 42, что дает значение 11010101, и добавления 1, что приводит к значению 11010110, или –42. Чтобы декодировать отрицательное число, необходимо вначале инвертировать все биты, а затем добавить 1 к результату. Например, инвертирование значения –42, или 11010110, приводит к значению 00101001, или 41, после добавления 1 к которому мы получаем 42.

Причина, по которой в Java (и большинстве других компьютерных языков) применяют двоичное дополнение, становится понятной при рассмотрении перехода через нуль. Если речь идет о значении типа `byte`, ноль представляется значением 00000000. В случае применения единичного дополнения простое инвертирование всех битов создает значение, равное 11111111, которое представляет отрицательный ноль. Проблема в том, что отрицательный ноль — недопустимое значение в целочисленной математике. Применение двоичного дополнения для представления отрицательных значений позволяет решить эту проблему. При этом к дополнению добавляется 1, что приводит к числу 100000000. Единичный бит оказывается сдвинутым влево слишком далеко, чтобы умещаться в значении типа `byte`. Тем самым достигается требуемое поведение, когда –0 эквивалентен 0, а 11111111 — код значения, равного –1. Хотя в приведенном примере мы использовали значение типа `byte`, тот же базовый принцип применяется и ко всем целочисленным типам Java.

Поскольку в Java для хранения отрицательных значений используется двоичное дополнение — и поскольку в Java все целочисленные значения являются значениями со знаком — применение побитовых операций может легко приводить к неожиданным результатам. Например, установка самого старшего бита равным 1 может привести к тому, что результирующее значение будет интерпретироваться как отрицательное число, независимо от того, к этому результату вы стремились или нет. Во избежание неприятных сюрпризов следует помнить, что независимо от того, как он был установлен, старший бит определяет знак целого числа.

Побитовые логические операции

Побитовые логические операции — это `&`, `|`, `^` и `~`. Результаты выполнения каждой из этих операций приведены в табл. 4.3. В ходе ознакомления с последующим материалом помните, что побитовые операции применяются к каждому отдельному биту каждого операнда.

Таблица 4.3. Результаты выполнения побитовых 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

Побитовое NOT

Называемая также операцией *побитового дополнения*, унарная операция NOT (НЕ), `~`, инвертирует все биты операнда. Например, число 42, которое имеет следующую последовательность битов:

```
00101010
```

в результате применения операции NOT преобразуется в:

```
11010101
```

Побитовое AND

Значение бита, полученное в результате побитовой операции AND, &, равно 1, если соответствующие биты в операндах также равны 1. Во всех остальных случаях значение результирующего бита равно 0. Например:

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

Побитовое OR

Результирующий бит, полученный в результате выполнения операции OR, |, равен 1, если соответствующий бит в любом из операторов равен 1, как показано в следующем примере:

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

Побитовое XOR

Результирующий бит, полученный в результате выполнения операции XOR, ^, равен 1, если соответствующий бит только в одном из операндов равен 1. Во всех других случаях результирующий бит равен 0. В следующем примере показано применение операции ^. Он демонстрирует также полезную особенность операции XOR. Обратите внимание на инвертирование последовательности битов числа 42 во всех случаях, когда второй операнд содержит бит, равный 1. Во всех случаях, когда второй операнд содержит бит, равный 0, значение первого операнда остается неизменным. Это свойство пригодится при выполнении некоторых операций с битами.

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

Использование побитовых логических операций

В следующей программе демонстрируется применение побитовых логических операций.

```
// Демонстрация побитовых 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 этом примере последовательности битов переменных `a` и `b` представляют все четыре возможных комбинации двух двоичных цифр: 0-0, 0-1, 1-0 и 1-1. О действии операций `|` и `&` на каждый бит можно судить по результирующим значениям переменных `c` и `d`. Значений, присвоенные переменных `e` и `f`, иллюстрируют действие операции `^`. Массив строк `binary` содержит читабельные двоичные представления чисел от 0 до 15. В этом примере массив индексирован, что позволяет увидеть двоичное представление каждого результирующего значения. Массив построен так, чтобы соответствующее строковое представление двоичного значения `n` хранилось в элементе массива `binary[n]`. Чтобы его можно было выводить посредством массива `binary`, значение `~a` уменьшается до значения, меньшего 16, путем его объединения со значением 0x0f (0000 1111 в двоичном представлении) операцией AND. Вывод этой программы имеет следующий вид:

```
  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, <<, сдвигает все биты значения влево на указанное число позиций. Она имеет следующую общую форму:

значение << *число*

Здесь *число* — количество позиций, на которое нужно сдвинуть влево биты в значении *значение*. То есть операция << смещает влево биты указанного значения на количество позиций, указанных в *число*. При каждом сдвиге влево самый старший бит сдвигается за пределы допустимого диапазона (и утрачивается), а ноль дописывается справа. Это означает, что при применении операции сдвига влево к операнду типа `int` биты утрачиваются, как только они сдвигаются за пределы 31 позиции. Если операнд имеет тип `long`, биты теряются после сдвига за пределы 63 позиции.

Автоматическое повышение типа, выполняемое в среде Java, приводит к непредвиденным результатам при выполнении сдвига в значениях типа `byte` и `short`. Как вы уже знаете, тип значений `byte` и `short` повышается до `int` при вычислении выражений. Более того, результат вычисления такого выражения также имеет тип `int`. Это означает, что результатом выполнения сдвига влево значения типа `byte` или `short` будет значение типа `int`, и сдвинутые влево биты не будут отброшены до тех пор, пока они не будут сдвинуты за пределы 31 позиции. Более того, при повышении до типа `int` отрицательное значение типа `byte` или `short` получит дополнительный знаковый разряд. Следовательно, старшие биты будут заполнены единицами. Поэтому выполнение операции сдвига влево при-

менительно к значению типа byte или short предполагает необходимость отбрасывания старших байтов результата типа int. Например, при выполнении сдвига влево в значении типа byte вначале будет осуществляться повышение типа значения до int, и лишь затем сдвиг. Это означает, что для получения требуемого сдвинутого значения типа byte необходимо отбросить три старших байта результата. Простейший способ достижения этого — обратное приведение результата к типу byte. Следующая программа демонстрирует эту концепцию.

```
// 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 генерирует следующий вывод:

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

Поскольку для выполнения вычислений тип переменной a повышается до int, сдвиг влево на две позиции значения 64 (0100 0000) приводит к значению i, равному 256 (1 0000 0000). Однако переменная b содержит значение, равное 0, поскольку после сдвига младший байт равен 0. Единственный единичный бит оказывается сдвинутым за пределы допустимого диапазона.

Поскольку каждый сдвиг влево на одну позицию, по сути, удваивает исходное значение, программисты часто используют это в качестве эффективной замены умножения на 2. Однако при этом следует соблюдать осторожность. При сдвиге единичного бита в старшую позицию (бит 31 или 63) значение становится отрицательным. Следующая программа демонстрирует это применение операции сдвига влево.

```
// Применение сдвига влево в качестве быстрого метода умножения на 2.
class MultByTwo {
  public static void main(String args[]) {
    int i;
    int num = 0xFFFFFFE;
    for(i=0; i<4; i++) {
    num = num << 1;
    System.out.println(num);
    }
  }
}
```

Программа генерирует следующий вывод:

```
536870908
1073741816
2147483632
-32
```

Начальное значение было специально выбрано таким, чтобы после сдвига влево на 4 позиции оно стало равным -32. Как видите, после сдвига единичного бита в позицию 31 число интерпретируется как отрицательное.

Сдвиг вправо

Операция сдвига вправо, >>, сдвигает все биты значения вправо на указанное количество позиций. В общем виде ее можно записать следующим образом:

```
значение >> число
```

Здесь *число* указывает количество позиций, на которое нужно сдвинуть вправо биты в значении *значение*. То есть операция >> перемещает все биты в указанном значении вправо на количество позиций, указанное в *число*.

Следующий фрагмент кода выполняет сдвиг вправо на две позиции в значении 32, в результате чего значение переменной a становится равным 8:

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

Когда какие-либо биты в значении "сдвигаются прочь", они теряются. Например, следующий фрагмент кода выполняет сдвиг вправо на две позиции в значении 35, что приводит к утрате двух младших битов и повторной установке значения переменной a равным 8:

```
int a = 35;
a = a >> 2; // a по-прежнему содержит 8
```

Чтобы лучше понять, как выполняется эта операция, рассмотрим ее применение к двоичным представлениям:

```
00100011      35
>> 2
00001000       8
```

При каждом сдвиге вправо выполняется деление значения на два с отбрасыванием любого остатка. Это свойство можно использовать для выполнения высокопроизводительного целочисленного деления на 2. Конечно, при этом нужно быть уверенным, что никакие биты не будут сдвинуты за пределы правой границы.

При выполнении сдвига вправо старшие (расположенные в крайних левых позициях) биты, освобожденные в результате сдвига, заполняются предыдущим содержимым старшего бита. Этот эффект называется *дополнительным знаковым разрядом* и служит для сохранения знака отрицательных чисел при их сдвиге вправо. Например, результат выполнения операции -8>>1 равен –4, что в двоичном представлении выглядит так:

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

Интересно отметить, что результат сдвига вправо значения –1 всегда равен –1, поскольку дополнительные знаковые разряды добавляют новые единицы к старшим битам.

Иногда при выполнении сдвига вправо появление дополнительных знаковых разрядов нежелательно. Например, следующая программа преобразует значение byte в соответствующее шестнадцатеричное строковое представление. Обратите внимание, что для

обеспечения возможности использования значения в качестве индекса массива шестнадцатеричных символов сдвинутое значение маскируется посредством его объединения со значением 0x0f операцией AND, что приводит к отбрасыванию любых битов дополнительных знаковых разрядов.

```
// Маскирование дополнительных знаковых разрядов.
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]);
   }
}
```

Вывод этой программы выглядит следующим образом:

```
b = 0xf1
```

Сдвиг вправо без учета знака

Как было показано, при каждом выполнении операция >> автоматически заполняет старший бит его предыдущим содержимым. В результате знак значения сохраняется. Однако иногда это нежелательно. Например, при выполнении сдвига вправо в какомлибо значении, которое не является числовым, использование дополнительных знаковых разрядов может быть нежелательным. Эта ситуация часто встречается при работе со значениями пикселей и графическими изображениями. Как правило, в этих случаях требуется сдвиг нуля в позицию старшего бита независимо от его первоначального значения. Такое действие называют *сдвигом вправо без учета знака*. Для его выполнения используют операцию сдвига вправо без учета знака Java, >>>, которая всегда вставляет ноль в позицию старшего бита.

Следующий фрагмент кода демонстрирует применение операции >>>. В этом примере значение переменной a установлено равным –1, все 32 бита двоичного представления которого равны 1. Затем в этом значении выполняется сдвиг вправо на 24 бита с заполнением старших 24 битов нулями и игнорированием обычно используемых дополнительных знаковых разрядов. В результате значение a становится равным 255.

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

Чтобы происходящее было понятнее, запишем эту же операцию в двоичной форме:

```
11111111 11111111 11111111 11111111 –1 в двоичном представлении типа int
>>>24
00000000 00000000 00000000 11111111 255 в двоичном представлении типа int
```

Часто операция >>> не столь полезна, как хотелось бы, поскольку она имеет смысл только для 32- и 64-разрядных значений. Помните, что в выражениях тип меньших значений автоматически повышается до int. Это означает применение дополнительных знаковых разрядов и выполнение сдвига по отношению к 32-разрядным, а не 8- или 16-разрядным значениям. То есть программист может подразумевать выполнение сдвига вправо без учета знака применительно к значению типа byte и заполнение нулями, начиная с бита 7.

Однако в действительности это не так, поскольку фактически сдвиг будет выполняться в 32-разрядном значении. Этот эффект демонстрирует следующая программа.

```
// Сдвиг без учета знака 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]);
   }
}
```

Из следующего вывода этой программы видно, что операция >>> не выполняет никаких действий по отношению к значениям типа `byte`. Для этого примера в качестве значения переменной `b` было выбрано произвольное отрицательное значение типа `byte`. Затем переменной `c` присваивается значение переменной `b` типа `byte`, сдвинутое вправо на четыре позиции, которое в связи с применением дополнительных знаковых разрядов равно `0xff`. Затем переменной `d` присваивается значение переменной `b` типа `byte`, сдвинутое вправо на четыре позиции без учета знака, которым должно было бы быть значение `0x0f`, но в действительности, из-за применения дополнительных знаковых разрядов во время повышения типа `b` до `int` перед выполнением сдвига, равное `0xff`. Последнее выражение устанавливает значение переменной `e` равным значению типа `byte` переменной `b`, замаскированному до 8 бит с помощью операции AND и затем сдвинутому вправо на четыре позиции, что дает ожидаемое значение, равное `0x0f`. Обратите внимание, что операция сдвига вправо без учета знака не применялась к переменной `d`, поскольку состояние знакового бита после выполнения операции AND было известно.

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

Побитовые составные операции с присваиванием

Подобно алгебраическим операциям, все двоичные побитовые операции имеют составную форму, которая объединяет побитовую операцию с операцией присваивания. Например, следующие два оператора, выполняющие сдвиг вправо на четыре позиции в значении переменной `a`, эквивалентны:

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

Аналогично, эквивалентны и следующие два оператора, которые присваивают переменной a результат выполнения побитовой операции a OR b:

```
a = a | b;
a |= b;
```

Следующая программа создает несколько целочисленных переменных, а затем использует составные побитовые операции 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 создает следующий вывод:

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

Операции сравнения

Операции сравнения определяют отношение одного операнда с другим. В частности, они определяют равенство и порядок следования. Операции сравнения перечислены в табл. 4.4.

Таблица 4.4. Операции сравнения в Java

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

Результат выполнения этих операций — значение типа boolean. Наиболее часто операции сравнения используют в выражениях, которые управляют оператором if и различными операторами цикла.

В Java можно сравнивать значения любых типов, в том числе целые значения, значения с плавающей точкой, символы и булевские значения, используя проверку равенства == и проверку неравенства !=. Обратите внимание, что Java равенство обозначают двумя знаками "равно", а не одним. (Одиночный знак "равно" — символ операции присваивания.) Сравнение с помощью операций упорядочения применимо только к числовым типам. То есть сравнение для определения того, какой из операндов больше или меньше другого, можно выполнять только для целочисленных операндов, операндов с плавающей точкой или символьных операндов.

Как уже было отмечено, результат выполнения операции сравнения представляет собой значение типа `boolean`. Например, следующий фрагмент кода вполне допустим:

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

В данном случае результат выполнения операции `a < b` (который равен `false`) сохраняется в переменной `c`.

Те читатели, которые знакомы с языками C/C++, должны обратить внимание на следующее. В программах на C/C++ следующие типы операторов встречаются очень часто:

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

В Java-программе эти операторы должны быть записаны следующим образом:

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

Это обусловлено тем, что в Java определение значений "истинно" и "ложно" отличается от их определений в языках C/C++. В C/C++ истинным считается любое ненулевое значение, а ложным — ноль. В Java значения `true` (истинно) и `false` (ложно) — нечисловые значения, которые никак не сопоставимы с нулевым или ненулевым значением. Поэтому, чтобы сравнить значение с нулевым или ненулевым значением, необходимо явно использовать одну или несколько операций сравнения.

Булевские логические операции

Описанные в этом разделе логические операции работают только с операндами типа `boolean`. Все логические операции с двумя операндами объединяют два значения типа `boolean`, образуя результирующее значение типа `boolean`. Булевские логические операции перечислены в табл. 4.5.

Таблица 4.5. Булевские логические операции в Java

Операция	Описание
`&`	Логическое AND (И)
`|`	Логическое OR (ИЛИ)
`^`	Логическое XOR (исключающее OR (ИЛИ))
`||`	Замыкающее OR

Окончание табл. 4.5

Операция	Описание
`&&`	Замыкающее AND
`!`	Логическое унарное NOT (НЕ)
`&=`	AND с присваиванием
`|=`	OR с присваиванием
`^=`	XOR с присваиванием
`==`	Равно
`!=`	Не равно
`?:`	Тернарная операция `if-then-else`

Логические булевские операции `&`, `|` и `^` действуют применительно к значениям типа `boolean` точно так же, как они действуют по отношению к битам целочисленных значений. Логическая операция `!` инвертирует булевское состояние: `!true == false` и `!false == true`. Результат выполнения каждой из логических операций приведен в табл. 4.6.

Таблица 4.6. Результаты выполнения булевских 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 приведена программа, которая выполняет практически те же действия, что и пример программы `BitLogic`, представленный ранее, но она работает с логическими значениями типа `boolean`, а не с двоичными разрядами.

```
// Демонстрация применения булевских 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 логические правила, что и к битам. Как видно из следующего вывода, в Java строковое представление значения типа `boolean` — значение одной из символьных констант `true` или `false`:

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

Замыкающие логические операции

Java предоставляет две интересные булевских операции, не встречающиеся во многих других языках программирования. Это вторые версии булевских операций AND и OR, получившие название *замыкающих* логических операций. Как видно из ранее приведенной таблицы, результат выполнения операции OR равен `true`, когда значение операнда `A` равно `true`, независимо от значения операнда `B`. Аналогично, результат выполнения операции AND равен `false`, когда значение операнда `A` равно `false`, независимо от значения операнда `B`. При использовании форм `||` и `&&` этих операций вместо `|` и `&` программа Java не будет вычислять значение правого операнда, если результат выражения можно определить по значению одного левого операнда. Это свойство очень удобно в тех случаях, когда значение правого операнда зависит от значения левого. Например, следующий фрагмент кода демонстрирует преимущество применения замыкающей логической операций для выяснения допустимости операции деления перед вычислением ее результата:

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

Благодаря применению замыкающей формы операции AND (`&&`) исключается риск возникновения исключения времени выполнения в случае равенства знаменателя (`denom`) нулю. Если бы эта строка кода была записана с применением одинарного символа `&` операции AND, программа вычисляла бы обе части выражения, что приводило бы к исключению времени выполнения при равенстве значения `denom` нулю.

Замыкающие формы операций AND и OR принято применять в тех случаях, когда требуется использование операций булевской логики, а односимвольные версии — исключительно для побитовых операций. Однако существуют исключения из этого правила. Например, рассмотрим следующий оператор:

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

В данном случае одиночный символ `&` гарантирует применение операции инкремента к значению `e` независимо от равенства 1 значения `c`.

Операция присваивания

Мы использовали операцию присваивания, начиная с главы 2. Теперь пора рассмотреть эту операцию формально. Символом *операции присваивания* служит одиночный знак равенства, =. В Java операция присваивания работает аналогично тому, как она работает во многих компьютерных языках. Она имеет следующую общую форму:

```
переменная = выражение;
```

В этом операторе тип *переменной* должен соответствовать типу *выражения*.

Операция присваивания обладает одной интересной особенностью, с которой вы, возможно, еще не знакомы: она позволяет создавать цепочки присваиваний. Например, рассмотрим следующий фрагмент кода:

```
int x, y, z;
x = y = z = 100; // устанавливает значения переменных x, y и z равными 100
```

В этом фрагменте кода единственный оператор устанавливает значения трех переменных x, y и z равными 100. Это обусловлено тем, что = — операция, которая использует значение правого выражения. Таким образом, значение выражения z=100 равно 100, которое затем присваивается переменной y, а затем — переменной x. Использование "цепочки присваивания" — удобный способ установки общего значения группы переменных.

Операция ?

Синтаксис Java содержит специальную *тернарную операцию*, которой можно заменять определенные типы операторов if-then-else. Это операция ?. Вначале она может казаться несколько непонятной, но со временем вы убедитесь в ее исключительной эффективности. Эта операция имеет следующую общую форму:

выражение1 **?** *выражение2* **:** *выражение3*

Здесь *выражение1* — любое выражение, приводящее к значению типа boolean. Если значение *выражения1* — true, программа вычисляет значение *выражения2*. В противном случае программа вычисляет значение *выражения3*. Результат выполнения операции **?** равен значению вычисленного выражения. И *выражение2*, и *выражение3* должны возвращать значение одного и того же типа, которым не может быть тип void.

Ниже приведен пример применения операции ?:

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

Когда Java-программа вычисляет это выражение присваивания, вначале она проверяет выражение слева от знака вопроса. Если значение denom равно 0, программа вычисляет выражение, указанное между знаками вопроса и двоеточия, и использует вычисленное значение в качестве значения всего выражения ?. Если значение denom не равно 0, программа вычисляет выражение, указанное после двоеточия, и использует его в качестве значения всего выражения ?. Затем значение, полученное в результате выполнения операции ?, присваивается переменной ratio.

Следующий пример программы демонстрирует применение операции ?. Эта программа служит для получения абсолютного значения переменной.

```
// Демонстрация использования операции ?.
class Ternary {
  public static void main(String args[]) {
    int i, k;
    i = 10;
    k = i < 0 ? -i : i; // получение абсолютного значения переменной i
    System.out.print("Абсолютное значение ");
    System.out.println(i + " равно " + k);
    i = -10;
    k = i < 0 ? -i : i; // получение абсолютного значения переменной i
    System.out.print("Абсолютное значение ");
    System.out.println(i + " равно " + k);
  }
}
```

Эта программа генерирует следующий вывод:

```
Абсолютное значение 10 равно 10
Абсолютное значение -10 равно 10
```

Приоритеты операций

Приоритеты операций Java, от высшего к низшему, описаны в табл. 4.7. Обратите внимание, что в первой строке таблицы указаны элементы, которые, как правило, не считают символами операций: круглые и квадратные скобки и символ точки. С технической точки зрения они являются *разделителями*, но в выражениях они действуют подобно операциям. Круглые скобки используют для изменения порядка выполнения операций. Как говорилось в предыдущей главе, квадратные скобки служат для индексации массивов. А символ точки используется для разыменования объектов, и эта операция будет рассмотрена в последующих главах книги.

Таблица 4.7. Приоритеты операций Java

Высший приоритет			
( )	[ ]	.	
++	– –	~	!
*	/	%	
+	–		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		
Низший приоритет			

Использование круглых скобок

Круглые скобки повышают приоритет заключенных в них операций. Часто это необходимо для получения требуемого результата. Например, рассмотрим следующее выражение:

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

Вначале это выражение добавляет 3 к значению b, а затем сдвигает значение a вправо на полученное количество позиций. То есть используя избыточные круглые скобки, это выражение можно было бы записать следующим образом:

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

Однако если вначале нужно выполнить сдвиг значения a вправо на b позиций, а затем добавить 3 к полученному результату, необходимо использовать круглые скобки следующим образом:

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

Кроме изменения обычного приоритета операций, иногда круглые скобки можно использовать для облегчения понимания смысла выражения. Сложные выражения могут оказаться трудными для понимания. Добавление избыточных, но облегчающих понимание смысла выражения, круглых скобок может способствовать исключению недоразумений в будущем. Например, какое из следующих выражений легче прочесть?

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

И еще один немаловажный момент: использование круглых скобок (избыточных или не избыточных) не ведет к снижению производительности программы. Поэтому добавление круглых скобок для повышения читабельности программы не оказывает на нее отрицательного влияния.

5
ГЛАВА

Управляющие операторы

В языках программирования управляющие операторы используются для реализации переходов и ветвлений в потоке выполнения команд программы в соответствии с ее состоянием. Управляющие операторы Java-программы можно разделить на следующие категории: операторы выбора, операторы цикла и операторы перехода. Операторы *выбора* позволяют программе выбирать различные ветви выполнения команд в соответствии с результатом выражения или состоянием переменной. Операторы *цикла* позволяют программе повторять выполнение одного или нескольких операторов (т.е. они образуют циклы). Операторы *перехода* обеспечивают возможность нелинейного выполнения программы. В этой главе мы рассмотрим все управляющие операторы Java.

Операторы выбора

В 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.

Оператор `if`

Оператор `if` был представлен в главе 2. А в этой главе мы рассмотрим его подробно. Оператор `if` — оператор условного выбора ветви Java-программы. Его можно использовать для направления выполнения программы по двум различным ветвям. Общая форма этого оператора выглядит следующим образом:

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

Здесь каждый *оператор* — это одиночный оператор или составной оператор, заключенный в фигурные скобки (т.е. *блок*). *Условие* — это любое выражение, возвращающее значение типа `boolean`. Выражение `else` не обязательно.

Оператор `if` работает следующим образом: если *условие* истинно, программа выполняет *оператор1*. В противном случае она выполняет *оператор2* (если он существует). Ни в одной из ситуаций программа не будет выполнять оба оператора.

Например, рассмотрим следующий фрагмент кода:

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

В данном случае, если значение a меньше значения b, значение переменной a устанавливается равным нулю. В противном случае значение переменной b устанавливается равным нулю. Ни в одной из ситуаций значения обеих переменных a и b не могут получить нулевые значения.

Чаще всего в управляющих выражениях оператора if будут использоваться операции сравнения. Однако это не обязательно. Для управления оператором if можно применять и одиночную переменную типа boolean, как показано в следующем фрагменте кода:

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

Помните, что только один оператор может следовать непосредственно за ключевым словом if или else. Если нужно включить больше операторов, придется создать код, подобный приведенному в следующем фрагменте:

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

В этом примере оба оператора, помещенные внутрь блока, будут выполняться, если значение переменной bytesAvailable больше нуля.

Некоторые программисты предпочитают использовать в операторе if фигурные скобки даже при наличии только одного оператора в каждом выражении. Это упрощает добавление операторов в дальнейшем и избавляет от беспокойства по поводу наличия фигурных скобок. Фактически пропуск определения блока в тех случаях, когда он необходим — достаточно распространенная ошибка. Например, рассмотрим следующий фрагмент кода:

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

Если судить по уровню отступа в коде, предполагалось, что оператор bytesAvailable=n; должен выполняться внутри выражения else. Однако, как вы помните, в Java-программе пробелы значения не имеют, и компилятор никак не может "узнать" намерения программиста. Компиляция этого кода будет выполняться без вывода каких-либо предупреждаю-

щих сообщений, но при выполнении программа будет вести себя не так, как было задумано. В следующем фрагменте ошибка предыдущего примера исправлена:

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

Вложенные операторы `if`

Вложенный оператор `if` — это оператор `if`, являющийся целью другого оператора `if` или `else`. В программах вложенные операторы `if` встречаются очень часто. При использовании вложенных операторов `if` важно помнить, что оператор `else` всегда связан с ближайшим оператором `if`, расположенным в одном с ним блоке и еще не связанным с другим оператором `else`. Например:

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

Как видно из комментариев, последний оператор `else` связан не с оператором `if(j<20)`, поскольку тот не находится в одном с ним блоке (несмотря на то, что он является ближайшим оператором `if`, который еще не связан с оператором `else`). Последний оператор `else` связан с оператором `if(i==10)`. Внутренний оператор `else` связан с оператором `if(k>100)`, поскольку тот является ближайшим внутри этого же блока.

Многозвенная структура `if-else-if`

Распространенной конструкцией программирования, построенной на основе последовательности вложенных операторов `if`, является *многозвенная структура* `if-else-if`. Она выглядит следующим образом:

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

Операторы `if` выполняются последовательно, сверху вниз. Как только одно из условий, управляющих оператором `if`, становится равным `true`, программа выполняет оператор, связанный с данным оператором `if`, и пропускает остальную часть многозвенной структуры. Если ни одно из условий не выполняется (не равно `true`), программа выполнит заключительный оператор `else`. Этот последний оператор служит условием,

используемым по умолчанию. То есть если проверка всех остальных условий дает отрицательный результат, программа выполняет последний оператор `else`. Если заключительный оператор `else` не указан, а результат проверки всех остальных условий равен `false`, программа не будет выполнять никаких действий.

Ниже приведен пример программы, в которой многозвенная структура `if-else-if` служит для определения времени года, к которому относится конкретный месяц.

```
// Демонстрация применения операторов 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 + ".");
  }
}
```

Программа создает следующий вывод:

```
Апрель относится к весне
```

Прежде чем продолжить прочтение главы, можете поэкспериментировать с этой программой. Вы убедитесь, что независимо от значения, присвоенного переменной `month`, внутри многозвенной структуры будет выполняться только один оператор присваивания.

Оператор `switch`

Оператор `switch` — оператор ветвления в Java. Он предлагает простой способ направления потока выполнения команд по различным ветвям кода в зависимости от значения управляющего выражения. Часто он оказывается эффективнее применения длинных последовательностей операторов `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, выполняемая по умолчанию
  }
```

Выражение должно иметь тип type, short, int или char. Тип каждого *значения*, указанного в операторах case должен быть совместим с типом *выражения*. (Для управления оператором switch можно использовать также перечислимое значение. Перечисления описаны в главе 12.) Каждое значение case должно быть уникальной символьной константой (т.е. постоянным значением, а не переменной). Дублирование значений case не допускается.

Оператор switch работает следующим образом. Значение выражения сравнивается с каждым из значений констант в операторах case. При обнаружении совпадения программа выполняет последовательность кода, следующую за данным оператором case. Если значения ни одной из констант не совпадает со значением выражения, программа выполняет оператор default. Однако этот оператор не обязателен. При отсутствии совпадений со значениями case и отсутствии оператора default программа не выполняет никаких дальнейших действий.

Оператор break внутри последовательности switch служит для прерывания последовательности операторов. Как только программа доходит до оператора break, она продолжает выполнение с первой строки кода, следующей за всем оператором switch. Этот оператор оказывает действие "выхода" из оператора switch.

Ниже представлен простой пример использования 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 3.");
      }
  }
}
```

Эта программа генерирует следующий вывод:

```
i равно нулю.
i равно единице.
i равно двум.
i равно трем.
i больше 3.
i больше 3.
```

Как видите, на каждой итерации цикла программа выполняет операторы, связанные с константой case, которая соответствует значению переменной i. Все остальные операто-

ры пропускаются. После того, как значение i становится больше 3, оно перестает соответствовать какой-либо константе case, поэтому программа выполняет оператор default.

Оператор break не обязателен. Если его опустить, программа продолжит выполнение со следующего оператора case. В некоторых случаях желательно использовать несколько операторов case без разделяющих их операторов break. Например, рассмотрим следующую программу:

```
// В последовательности switch операторы break необязательны.
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 генерирует следующий вывод:

```
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 равно или больше 10
i равно или больше 10
```

Как видите, программа выполняет каждый из операторов case, пока не достигнет оператора break (или конца оператора switch).

Хотя приведенный пример был создан специально в качестве иллюстрации, пропуск операторов break находит множество применений в реальных программах. В качестве более реалистичного примера мы рассмотрим измененную версию приведенной ранее программы со временами года. В этой версии мы использовали оператор switch, что позволило создать более эффективную реализацию.

```
// Усовершенствованная версия программы с временами 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 + ".");
  }
}
```

Вложенные операторы `switch`

Оператор `switch` можно использовать в последовательности операторов внешнего оператора `switch`. Такой оператор называют *вложенным* оператором `switch`. Поскольку оператор `switch` определяет собственный блок, каких-либо конфликтов между константами `case` внутреннего и внешнего операторов `switch` не происходит. Например, следующий фрагмент полностью допустим:

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

В данном случае оператор `case1:` внутреннего оператора `switch` не конфликтует с оператором `case1:` внешнего оператора `switch`. Программа сравнивает значение пере-

менной count только со списком ветвей case внешнего уровня. Если значение count равно 1, программа сравнивает значение переменной target с внутренним списком ветвей case.

В качестве итога можно отметить следующие три важных свойства оператора switch.

- Оператор switch отличается от оператора if тем, что он может выполнять проверку только равенства, в то время как оператор if может вычислять булевское выражение любого типа. То есть оператор switch ищет только соответствие между значением выражения и одной из констант case.
- Никакие две константы case в одном и том же операторе switch не могут иметь одинаковые значения. Конечно, внутренний оператор switch и содержащий его внешний оператор switch могут иметь одинаковые константы case.
- Как правило, оператор switch эффективнее набора вложенных операторов if.

Последнее свойство представляет особый интерес, поскольку позволяет понять работу компилятора Java. Компилируя оператор switch, компилятор Java будет проверять каждую из констант case и создавать "таблицу переходов", которую будет использовать для выбора ветви программы в зависимости от значения выражения. Поэтому в тех случаях, когда требуется осуществлять выбор в большой группе значений, оператор switch будет выполняться значительно быстрее последовательности операторов if-else. Это обусловлено тем, что компилятору известно, что все константы case имеют один и тот же тип, и их нужно просто проверять на предмет равенства значению выражения switch. Компилятор не располагает подобными сведениями о длинном списке выражений оператора if.

Операторы цикла

Операторами цикла Java являются for, while и do-while. Эти операторы образуют конструкции, обычно называемые *циклами*. Как наверняка известно читателям, циклы многократно выполняют один и тот же набор инструкций до тех пор, пока не будет удовлетворено условие завершения цикла. Как вы вскоре убедитесь, Java предлагает средства создания циклов, способные удовлетворить любые потребности программирования.

Цикл while

Оператор while — наиболее часто используемый оператор цикла Java. Он повторяет оператор или блок операторов до тех пор, пока значение его управляющего выражения истинно. Он имеет следующую общую форму:

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

Условием может быть любое булевское 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 если в цикле повторяется только один оператор.

В качестве примера рассмотрим цикл while, который выполняет обратный отсчет, начиная с 10, вывод ровно 10 строк "тактов":

```
// Демонстрация использования 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”:

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

Поскольку цикл `while` вычисляет свое условное выражение в начале цикла, тело цикла не будет выполнено ни разу, если в самом начале условие оказывается ложным. Например, в следующем фрагменте кода метод `println()` никогда не будет вызван:

```
int a = 10, b = 20;
while(a > b)
  System.out.println("Эта строка отображаться не будет");
```

Тело цикла `while` (или любого другого цикла Java) может быть пустым. Это обусловлено тем, что синтаксис Java допускает применение *нулевого оператора* (содержащего только символ точки с запятой). Например, рассмотрим следующую программу:

```
// Целевая часть цикла может быть пустой.
class NoBody {
  public static void main(String args[]) {
    int i, j;
    i = 100;
    j = 200;
    // вычисление среднего значения i и j
    while(++i < --j) ; // в этом цикле тело цикла отсутствует
    System.out.println("Среднее значение равно " + i);
  }
}
```

Эта программа вычисляет среднее значение `i` и `j`. Она генерирует следующий вывод:

```
Среднее значение равно 150
```

Этот цикл `while` работает следующим образом. Значение `i` увеличивается, а значение `j` уменьшается на единицу. Затем программа сравнивает эти два значения. Если новое значение `i` по-прежнему меньше нового значения `j`, цикл повторяется. Если значение `i` равно или больше значения `j`, выполнение цикла прекращается. По выходу из цикла переменная `i` будет содержать среднее значение исходных значений `i` и `j`. (Конечно, эта проце-

дура работает только в том случае, если в самом начале значение i меньше значения j.) Как видите, никакой потребности в наличии тела цикла не существует. Все действия выполняются внутри самого условного выражения. В профессионально написанном Java-коде короткие циклы часто не содержат тела, если само по себе управляющее выражение может выполнять все необходимые действия.

Цикл do-while

Как вы видели, если в начальный момент условное выражение, управляющее циклом while, ложно, тело цикла вообще не будет выполняться. Однако иногда желательно выполнить тело цикла хотя бы один раз, даже если в начальный момент условное выражение ложно. Иначе говоря, существуют ситуации, когда проверку условия прерывания цикла желательно выполнять в конце цикла, а не в его начале. К счастью, Java поддерживает именно такой цикл: do-while. Этот цикл всегда выполняет тело цикла 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 программа вначале выполняет тело цикла, а затем вычисляет условное выражение. Если это выражение истинно, цикл повторяется. В противном случае выполнение цикла прерывается. Как и во всех циклах Java, *выражение* должно быть булевским.

Ниже приведена измененная программа вывода тактов, которая демонстрирует использование цикла do-while. Она генерирует такой же вывод, что и предыдущая версия.

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

Хотя с технической точки зрения в приведенной программе цикл записан правильно, его можно переписать в более эффективном виде:

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

В этом примере операции декремента переменной n и сравнения результирующего значения с нулем объединены в одном выражении (--n>0). Программа работает следующим образом. Вначале она выполняет оператор --n, уменьшая значение n на единицу и возвращая новое значение переменной n. Затем программа сравнивает это значение с нулем. Если оно больше нуля, выполнение цикла продолжается. В противном случае цикл прерывается.

Цикл do-while особенно удобен при выборе пункта меню, поскольку обычно в этом случае требуется, чтобы тело цикла меню выполнялось, по меньшей мере, один раз. Рассмотрим следующую программу, которая реализует очень простую систему справки по операторам выбора и цикла Java:

```
// Использование 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:");
      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 интересующий 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; повторение)");
        System.out.println("  оператор;");
      break;
    }
  }
}
```

Пример вывода выполнения этой программы выглядит следующим образом:

```
Справка по:
   1. if
   2. switch
   3. while
   4. do-while
   5. for
Выберите интересующий пункт:
4
do-while:
do {
  оператор;
} while (условие);
```

В этой программе в цикле `do-while` осуществляется проверка допустимости введенного пользователем значения. Если это значение недопустимо, программа предлагает пользователю повторить ввод. Поскольку меню должно отобразиться, по меньшей мере, один раз, цикл `do-while` является прекрасным средством решения этой задачи.

Отметим еще несколько особенностей приведенного примера. Обратите внимание, что считывание символов с клавиатуры выполняется посредством вызова метода `System.in.read()`. Это — одна из функций консольного ввода Java. Хотя мы и отложим подробное рассмотрение методов консольного ввода-вывода до главы 13, покамест отметим, что в данном случае метод `System.in.read()` используется для выяснения выбора, осуществленного пользователем. Этот метод считывает символы из стандартного ввода (возвращаемые в виде целочисленных значений — именно потому тип возвращаемого значения был приведен к `char`). По умолчанию данные из стандартного ввода помещаются в буфер построчно, поэтому, чтобы любые введенные символы были пересланы программе, необходимо нажать клавишу <ENTER>.

Консольный ввод Java может вызывать некоторые затруднения при работе. Более того, большинство реальных Java-программ будут графическими и ориентированным на работу в оконном режиме. Поэтому в данной книге консольному вводу уделяется не очень много внимания. Однако в данном случае он удобен. Еще один важный момент. Поскольку мы используем метод `System.in.read()`, программа должна содержать выражение `throws java.io.IOException`. Эта строка необходима для обработки ошибок ввода. Она является составной частью системы обработки исключений Java, которая будет рассмотрена в главе 10.

Цикл `for`

Простая форма цикла `for` была представлена в главе 2. Вскоре читатели убедятся в больших возможностях и гибкости этой конструкции.

Начиная с версии JDK 5, в Java существуют две формы цикла `for`. Первая — традиционная форма, используемая начиная с исходной версии Java. Вторая — новая форма "for-each". Мы рассмотрим оба типа цикла `for`, начиная с традиционной формы.

Общая форма традиционного оператора `for` выглядит следующим образом:

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

Если в цикле будет повторяться только один оператор, фигурные скобки можно опустить.

Цикл for действует следующим образом. При первом запуске цикла программа выполняет *инициализационную* часть цикла. В общем случае это выражение, устанавливающее значение *управляющей переменной цикла*, которая действует в качестве счетчика, управляющего циклом. Важно понимать, что выражение инициализации выполняется только один раз. Затем программа вычисляет *условие*, которое должно быть булевским выражением. Как правило, выражение сравнивает значение управляющей переменной с целевым значением. Если это значение истинно, программа выполняет тело цикла. Если оно ложно, выполнение цикла прерывается. Затем программа выполняет часть *повторение* цикла. Обычно это выражение, которое увеличивает или уменьшает значение управляющей переменной. Затем программа повторяет цикл, при каждом прохождении вначале вычисляя условное выражение, затем выполняя тело цикла и выполняя выражение повторения. Процесс повторяется до тех пор, пока значение выражения повторения не станет ложным.

Ниже приведена версия программы подсчета "тактов", в которой использован цикл for.

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

Объявление управляющих переменных цикла внутри цикла for

Часто переменная, которая управляет циклом for, требуется только для него и не используется нигде больше. В этом случае переменную можно объявить внутри инициализационной части оператора for. Например, предыдущую программу можно переписать, объявляя управляющую переменную *n* типа int внутри цикла for:

```
// Объявление управляющей переменной цикла внутри цикла for.
class ForTick {
  public static void main(String args[]) {
    // в данном случае переменная n объявляется внутри цикла for
    for(int n=10; n>0; n--)
      System.out.println("такт " + n);
  }
}
```

При объявлении переменной внутри цикла for необходимо помнить о следующем важном обстоятельстве: область и время существования этой переменной полностью совпадают с областью и временем существования оператора for. (То есть область существования переменной ограничена циклом for.) Вне цикла for переменная прекратит свое существование. Если управляющую переменную цикла нужно использовать в других частях программы, ее нельзя объявлять внутри цикла for.

В тех случаях, когда управляющая переменная цикла не требуется нигде больше, большинство программистов Java предпочитают объявлять ее внутри оператора for. В качестве примера приведем простую программу, которая проверяет, является ли введенное число простым. Обратите внимание, что управляющая переменная цикла i объявлена внутри цикла for, поскольку она нигде больше не требуется.

```
// Проверка принадлежности к простым числам.
class FindPrime {
   public static void main(String args[]) {
      int num;
      boolean isPrime = true;
       num = 14;
      for(int i=2; i <= num/i; i++) {
         if((num % i) == 0) {
            isPrime = false;
            break;
         }
      }
      if(isPrime) System.out.println("Простое");
      else System.out.println("Не простое");
   }
}
```

Использование запятой

В ряде случаев требуется указание нескольких операторов в инициализационной и итерационной частях цикла `for`. Например, рассмотрим 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. Поскольку цикл управляется двумя переменными, желательно, чтобы их обе можно было бы включить в сам оператор `for`, а не выполнять обработку переменной `b` вручную. К счастью, Java предоставляет средства для выполнения этой задачи. Чтобы две или более переменных могли управлять циклом `for`, Java допускает указывать по несколько операторов как в инициализационной, так и итерационной частях оператора `for`. Один от другого операторы отделяются запятыми.

Используя запятые, предыдущий цикл `for` можно записать в более эффективном виде:

```
// Использование запятой.
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 этом примере в инициализационной части цикла мы устанавливаем начальные значения обеих управляющих переменных a и b. Оба разделенных запятой оператора в итерационной части выполняются при каждом повторении цикла. Программа генерирует следующий вывод:

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

На заметку! *Читатели, знакомые с языками C/C++, знают, что в этих языках запятая — символ операции, который можно использовать в любом допустимом выражении. Однако в Java это не так. В нем запятая служит разделителем.*

Разновидности цикла `for`

Цикл `for` поддерживает несколько разновидностей, которые увеличивают его возможности и повышают применимость. Гибкость этого цикла обусловлена тем, что его три части: инициализацию, проверку условий и итерационную не обязательно использовать только по прямому назначению. Фактически каждый из разделов оператора `for` можно применять в любых целях. Рассмотрим несколько примеров.

Одна из наиболее часто встречающихся вариаций предполагает использование условного выражения. В частности, это выражение не обязательно должно выполнять сравнение управляющей переменной цикла с каким-либо целевым значением. Фактически условием, управляющим циклом `for`, может быть любое булевское выражение. Например, рассмотрим следующий фрагмент:

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

В этом примере выполнение цикла `for` продолжается до тех пор, пока значение переменной `done` не будет установлено равным `true`. В этом цикле проверка значения управляющей переменной цикла `i` не выполняется.

Ниже приведена еще одна интересная разновидность цикла `for`. Инициализационное или итерационное выражения либо они оба могут отсутствовать, как показано в следующей программе:

```
// Части цикла for могут быть пустыми.
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 этом примере инициализационное и итерационное выражения вынесены за пределы цикла `for`. В результате соответствующие части оператора `for` пусты. Хотя в данном простом примере — действительно, его можно считать достаточно примитивным — это и не имеет никакого значения, в отдельных случаях такой подход имеет смысл. Например, если начальное условие определяется посредством сложного выражения где-то в другом месте программы, или если значение управляющей переменной цикла изменяется случайным образом в зависимости от действий, выполняемых внутри тела цикла, оставление этих частей цикла `for` пустыми может оказаться целесообразным.

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

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

Этот цикл может выполняться бесконечно, поскольку условие, по которому он был бы прерван, отсутствует. Хотя некоторые программы, такие как командные процессоры операционной системы, требуют наличия бесконечного цикла, большинство "бесконечных циклов" в действительности представляют собой всего лишь циклы с особыми условиями прерывания. Как вы вскоре убедитесь, существует способ прерывания цикла — даже бесконечного, подобного приведенному примеру — который не требует использования обычного условного выражения цикла.

Версия "for-each" цикла `for`

Начиная с версии JDK 5 в Java можно использовать вторую форму цикла `for`, реализующую цикл в стиле "for-each" ("для каждого"). Как вам, возможно, известно, в современной теории языков программирования все большее применение находит концепция циклов "for-each", которые быстро становится стандартными функциональными возможностями во многих языках. Цикл в стиле "for-each" предназначен для строго последовательного выполнения повторяющихся действий по отношению к коллекции объектов, такой как массив. В отличие некоторых языков, подобных C#, в котором для реализации циклов "for-each" используют ключевое слово `foreach`, в Java возможность применения цикла "for-each" реализована за счет усовершенствования цикла `for`. Преимущество этого подхода состоит в том, что для его реализации не требуется дополнительное ключевое слово, и никакой ранее существовавший код не разрушается. Цикл `for` в стиле "for-each" называют также *усовершенствованным* циклом `for`. Общая форма версии "for-each" цикла `for` имеет следующий вид:

```
for(тип итер-пер : коллекция) блок-операторов
```

Здесь *тип* указывает тип, а *итер-пер* — имя *итерационной переменной*, которая последовательно будет принимать значения из коллекции, от первого до последнего. Элемент *коллекция* указывает коллекцию, по которой должен выполняться цикл. С циклом `for` можно применять различные типы коллекций, но в этой главе мы будем использовать только массивы. (Другие типы коллекций, которые можно применять с циклом `for`, вроде определенных в каркасе коллекций Collection Framework, рассматриваются в последующих главах книги.) На каждой итерации цикла программа извлекает следующий элемент коллекции и сохраняет его в переменной *итер-пер*. Цикл выполняется до тех пор, пока не будут получены все элементы коллекции.

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

Чтобы понять побудительные причины применения циклов в стиле "for-each", рассмотрим тип цикла `for`, для замены которого предназначен этот стиль. В следующем фрагменте 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, мы последовательно считываем значения каждого из элементов массива `nums`. Таким образом, чтение всего массива выполняется строго последовательно. Это выполняется путем индексации массива `nums` вручную по управляющей переменной цикла `i`.

Цикл `for` в стиле "for-each" позволяет автоматизировать этот процесс. В частности, применение такого цикла позволяет не устанавливать значение счетчика цикла за счет указания его начального и конечного значений, и исключает необходимость индексации массива вручную. Вместо этого программа автоматически выполняет цикл по всему массиву, последовательно получая значения каждого из его элементов, от первого до последнего. Например, с учетом версии "for-each" цикла `for` предыдущий фрагмент можно переписать следующим образом:

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;
for(int x: nums) sum += x;
```

При каждом прохождении цикла переменной `x` автоматически присваивается значение, равное значению следующего элемента массива `nums`. Таким образом, на первой итерации `x` содержит 1, на второй — 2 и т.д. При этом не только упрощается синтаксис программы, но и исключается возможность ошибок выхода за пределы массива.

Ниже показан пример полной программы, иллюстрирующей применение описанной версии "for-each" цикла `for`.

```
// Использование 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;
    // использование стиля for-each для отображения и суммирования значений
    for(int x : nums) {
      System.out.println("Значение равно: " + x);
      sum += x;
    }
    System.out.println("Сумма равна: " + sum);
  }
}
```

Эта программа генерирует следующий вывод:

```
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 видно из этого вывода, оператор `for` в стиле "for-each" автоматически выполняет цикл по элементам массива, от наименьшего индекса к наибольшему.

Хотя повторение цикла `for` в стиле "for-each" выполняется до тех пор, пока не будут обработаны все элементы массива, цикл можно прервать и раньше, используя оператор `break`. Например, следующая программа суммирует значения пяти первых элементов массива `nums`.

```
// Использование 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 };

    // использование цикла for для отображения и суммирования значений
    for(int x : nums) {
      System.out.println("Значение равно: " + x);
      sum += x;
      if(x == 5) break; // прекращение цикла после получения 5 значений
    }
    System.out.println("Сумма пяти первых элементов равна: " + sum);
  }
}
```

Программа генерирует следующий вывод:

```
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 прекращается после получения значения пятого элемента. Оператор `break` можно использовать также и с другими циклами Java. Подробнее этот оператор будет рассмотрен в последующих разделах настоящей главы.

При использовании цикла в стиле "for-each" необходимо помнить о следующем важном обстоятельстве. Его итерационная переменная является переменной "только для чтения", поскольку она связана только с исходным массивом. Операция присваивания значения итерационной переменной не оказывает никакого влияния на исходный массив. Иначе говоря, содержимое массива нельзя изменять, присваивая новое значение итерационной переменной. Например, рассмотрим следующую программу:

```
// Переменная цикла 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();
  }
}
```

Первый цикл `for` увеличивает значение итерационной переменной на 10. Однако эта операция присваивания не оказывает никакого влияния на исходный массив `nums`, как видно из результата выполнения второго оператора `for`. Генерируемый программой вывод 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 версия цикла `for` применима также и к многомерным массивам. Однако следует помнить, что в Java многомерные массивы состоят из *массивов массивов*. (Например, двумерный массив — это массив одномерных массивов.) Это важно при выполнении итерации в многомерном массиве, поскольку результат каждой итерации — *следующий массив*, а не отдельный элемент. Более того, тип итерационной переменной цикла `for` должен быть совместим с типом получаемого массива. Например, в случае двумерного массива итерационная переменная должна быть ссылкой на одномерный массив. В общем случае при использовании цикла "for-each" для выполнения итерации в массиве размерности *N* получаемые объекты будут массивами размерности *N*-1. Дабы понять, что из этого следует, рассмотрим следующую программу. В ней вложенные циклы `for` служат для получения упорядоченных по строкам элементов двумерного массива.

```
// Использование цикла for в стиле for-each применительно к двумерному массиву.
class ForEach3 {
  public static void main(String args[]) {
    int sum = 0;
    int nums[][] = new int[3][5];

    // присвоение значений элементам массива nums
    for(int i = 0; i < 3; i++)
      for(int j=0; j < 5; j++)
        nums[i][j] = (i+1)*(j+1);

    // использование цикла for в стиле for-each для отображения
    // 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 генерирует следующий вывод:

```
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 этой программы заслуживает особого внимания:

```
for(int x[] : nums) {
```

Обратите внимание на способ объявления переменной `x`. Эта переменная — ссылка на одномерный массив целочисленных значений. Это необходимо потому, что результат выполнения каждой итерации цикла `for` — следующий *массив* в массиве `nums`, начиная с массива, указанного элементом `nums[0]`. Затем внутренний цикл `for` выполняет итерацию по каждому из этих массивов, отображая значения каждого из элементов.

Использование усовершенствованного цикла `for`

Поскольку каждый оператор `for` в стиле "for-each" может выполнять цикл по элементам массива только последовательно, начиная с первого и заканчивая последним, может показаться, что его применение ограничено. Однако это не так. Множество алгоритмов требуют использования именно этого механизма. Одним из наиболее часто используемых алгоритмов является поиск. Например, следующая программа использует цикл `for` 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;
    // использование цикла for в стиле for-each для в nums значения val
    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 цикла `for` полностью оправдан, поскольку поиск в неупорядоченном массиве предполагает последовательный просмотр каждого из элементов. (Конечно, если бы массив был упорядоченным, можно было бы использовать бинарный поиск, реализация которого требовала бы применения другого стиля цикла.) К другим типам приложений, которым применение циклов в стиле “for-each” предоставляет преимущества, относятся вычисление среднего значения, отыскание минимального или максимального значения в наборе, поиск дубликатов и т.п.

Хотя в примерах этой главы мы использовали массивы, цикл `for` в стиле “for-each” особенно удобен при работе с коллекциями, определенными в каркасе Collections Framework (Каркас коллекций), который описан во второй части книги. В более общем случае оператор `for` может выполнять цикл по элементам любой коллекции объектов, если эта коллекция удовлетворяет определенному набору ограничений, который описан в главе 17.

Вложенные циклы

Подобно всем другим языкам программирования, Java допускает использование вложенных циклов. То есть один цикл может выполняться внутри другого. Например, в следующей программе использованы 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 генерирует следующий вывод:

```
..........
.........
........
.......
......
.....
....
...
..
.
```

Операторы перехода

В Java определены три оператора перехода: `break`, `continue` и `return`. Они передают управление другой части программы. Рассмотрим каждый из них.

На заметку! *Кроме операторов перехода, рассмотренных в этом разделе, Java поддерживает еще один способ изменения порядка выполнения инструкций программы: обработку исключений. Обработка исключений предоставляет структурированный метод, посредством которого программа может обнаруживать и обрабатывать ошибки времени выполнения. Для поддержки этого метода служат ключевые слова* `try`, `catch`, `throws` *и* `finally`. *По сути, механизм обработки ошибок позволяет программе выполнять нелокальные ветви. Поскольку тема обработки исключений очень обширна, она рассмотрена в посвященной ей главе 10.*

Использование оператора `break`

В Java оператор `break` находит три применения. Во-первых, как уже было показано, он завершает последовательность операторов в операторе `switch`. Во-вторых, его можно использовать для выхода из цикла. И, в-третьих, его можно использовать в качестве "цивилизованной" формы оператора безусловного перехода ("goto"). Рассмотрим последние два применения.

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 программа встречает оператор `break` внутри цикла, она прекращает выполнение цикла, и управление передается оператору, следующему за циклом. Ниже показан простой пример.

```
// Использование 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 i равно 10
      System.out.println("i: " + i);
    }
    System.out.println("Цикл завершен.");
  }
}
```

Эта программа генерирует следующий вывод:

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Цикл завершен.
```

Как видите, хотя цикл for должен был бы выполняться для значений управляющей переменной от 0 до 99, оператор break приводит к более раннему выходу из него, когда значение переменной i становится равным 10.

Оператор break можно использовать в любых циклах Java, в том числе в преднамеренно бесконечных циклах. Например, в предыдущей программе можно было применить цикл while. Эта программа генерирует вывод, совпадающий с предыдущим.

```
// Использование 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 i равно 10
      System.out.println("i: " + i);
      i++;
    }
    System.out.println("Цикл завершен.");
  }
}
```

В случае его использования внутри набора вложенных циклов оператор break осуществляет выход только из самого внутреннего цикла. Например:

```
// Использование 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, если j равно 10
        System.out.print(j + " ");
      }
      System.out.println();
    }
    System.out.println("Циклы завершены.");
  }
}
```

Эта программа генерирует следующий вывод:

```
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. На внешний цикл он не оказывает никакого влияния.

При использовании оператора break необходимо помнить следующее. Во-первых, в цикле можно использовать более одного оператора break. Однако при этом следует соблюдать осторожность. Как правило, применение слишком большого числа операторов break приводит к деструктуризации кода. Во-вторых, оператор break, который завершает последовательность операторов в операторе switch, оказывает влияние только на данный оператор switch, а не на какие-либо содержащие его циклы.

Помните! *Оператор* `break` *не был разработан в качестве обычного средства выхода из цикла. Для этого служит условное выражение цикла. Оператор* `break` *следует использовать для выхода из цикла только в определенных особых ситуациях.*

Использование оператора `break` в качестве формы оператора безусловного перехода

Кроме его применения с операторами `switch` и циклами, оператор `break` можно использовать и сам по себе в качестве "цивилизованной" формы оператора безусловного перехода ("goto"). Язык Java не содержит оператора "goto", поскольку он позволяет выполнять ветвление программ произвольным и неструктурированным образом. Как правило, код, который управляется оператором "goto", труден для понимания и поддержки. Кроме того, этот оператор исключает возможность оптимизации кода для определенного компилятора. Однако в некоторых случаях оператор "goto" — ценная и вполне допустимая конструкция управления потоком команд. Например, оператор "goto" может быть полезен при выходе из набора вложенных циклов с большим количеством уровней. Для таких ситуаций Java определяет расширенную форму оператора `break`. Используя эту форму, можно, например, осуществлять выход из одного или нескольких блоков кода. Эти блоки не обязательно должны быть частью цикла или оператора `switch`. Они могут быть любым блоком. Более того, можно точно указать оператор, с которого будет продолжено выполнение программы, поскольку эта форма оператора `break` работает с метками. Как будет показано, оператор `break` предоставляет все преимущества оператора "goto", не порождая его проблемы. Общая форма оператора `break` с меткой имеет следующий вид:

```
break метка;
```

Чаще всего, *`метка`* — это имя метки, идентифицирующей блок кода. Им может быть как самостоятельный блок кода, так и целевой блок другого оператора. При выполнении этой формы оператора `break` управление передается названному блоку кода. Помеченный блок кода должен содержать оператор `break`, но он не обязательно должен быть непосредственно содержащим его блоком. В частности это означает, что оператор `break` с меткой можно применять для выхода из набора вложенных блоков. Однако его нельзя использовать для передачи управления внешнему блоку кода, который не содержит данный оператор `break`.

Чтобы пометить блок, необходимо поместить метку в его начале. *Метка* — это любой допустимый идентификатор Java, за которым следует двоеточие. Как только блок помечен, его метку можно использовать в качестве цели оператора `break`. В результате выполнение программы будет продолжено с *конца* помеченного блока. Например, следующая программа содержит три вложенных блока, каждый из которых помечен своей меткой. Оператор `break` приводит к переходу к концу блока с меткой `second` с пропуском двух операторов `println()`.

```
// Использование 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 генерирует следующий вывод:

```
Предшествует оператору break.
Этот оператор следует за блоком second.
```

Одно из наиболее распространенных применений оператора `break` с меткой — его использование для выхода из вложенных циклов. Например, в следующей программе внешний цикл выполняется только один раз:

```
// Использование 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 генерирует следующий вывод:

```
Проход 0: 0 1 2 3 4 5 6 7 8 9 Циклы завершены.
```

Как видите, когда внутренний цикл выполняет выход во внешний цикл, это приводит к завершению обоих циклов.

Следует иметь в виду, что нельзя выполнить переход к какой-либо метке, которая определена не для содержащего данный оператор `break` блока. Например, следующая программа содержит ошибку и поэтому ее компиляция будет невозможна:

```
// Эта программа содержит ошибку.
class BreakErr {
  public static void main(String args[]) {
    one: for(int i=0; i<3; i++) {
      System.out.print("Pass " + i + ": ");
    }
    for(int j=0; j<100; j++) {
      if(j == 10) break one; // ОШИБКА!
      System.out.print(j + " ");
    }
  }
}
```

Поскольку блок, помеченный меткой `one`, не содержит оператор `break`, передача управления этому внешнему блоку невозможна.

Использование оператора continue

Иногда требуется, чтобы повторение цикла осуществлялось с более раннего оператора его тела. То есть на данной конкретной итерации может требоваться продолжить выполнение цикла без обработки остального кода в его теле. По сути, 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 управляющему условному выражению цикла. В цикли `for` управление передается вначале итерационной части цикла `for`, а потом условному выражению. Во всех этих трех циклах любой промежуточный код пропускается.

Ниже приведен пример программы, в которой оператор `continue` используется для вывода двух чисел в каждой строке.

```
// Демонстрация применения 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 этом коде операция `%` служит для проверки четности значения переменной `i`. Если оно четное, выполнение цикл продолжается без перехода к новой строке. Программа генерирует следующий вывод:

```
0 1
2 3
4 5
6 7
8 9
```

Как и оператор `break`, оператор `continue` может содержать метку содержащего его цикла, который нужно продолжить. Ниже показан пример программы, в которой оператор `continue` применяется для вывода треугольной таблицы умножения чисел от 0 до 9.

```
// Использование 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 этом примере оператор `continue` прерывает цикл подсчета значений `j` и продолжает цикл со следующей итерации цикла подсчета `i`. Вывод этой программы имеет следующий вид:

```
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 применения оператора `continue` встречаются редко. Одна из причин состоит в том, что Java предлагает широкий выбор операторов цикла, удовлетворяющих требованиям большинства приложений. Однако в тех случаях, когда требуется более раннее начало новой итерации, оператор `continue` предоставляет структурированный метод выполнения этой задачи.

Оператор `return`

Последний из управляющих операторов — `return`. Его используют для выполнения явного возврата из метода. То есть он снова передает управление объекту, который вызвал данный метод. Как таковой этот оператор относится к операторам перехода. Хотя полное описание оператора `return` придется отложить до рассмотрения методов в главе 6, все же кратко ознакомимся с его особенностями.

Оператор `return` можно использовать в любом месте метода для возврата управления тому объекту, который вызвал данный метод. Таким образом, оператор `return` немедленно прекращает выполнение метода, в котором он находится. Следующий пример иллюстрирует это. В данном случае оператор `return` приводит к возврату управления системе времени выполнения Java, поскольку именно она вызывает метод `main()`.

```
// Демонстрация использования оператора return.
class Return {
  public static void main(String args[]) {
    boolean t = true;
    System.out.println("До выполнения возврата.");
    if(t) return; // возврат к вызывающему объекту
    System.out.println("Этот оператор выполняться не будет.");
  }
}
```

Вывод этой программы имеет вид:

```
До выполнения возврата.
```

Как видите, заключительный оператор `println()` не выполняется. Сразу после выполнения оператора `return` программа возвращает управление вызывающему объекту.

И последний нюанс: в приведенной программе использование оператора `if(t)` обязательно. Без него компилятор Java сигнализировал бы об ошибке “unreachable code” (“недостижимый код”), поскольку выяснил бы, что последний оператор `println()` никогда не будет выполняться. Во избежание этой ошибки в демонстрационном примере пришлось ввести компилятор в заблуждение с помощью оператора `if`.

6 ГЛАВА

Знакомство с классами

Класс — центральный компонент Java. Поскольку класс определяет форму и сущность объекта, он является той логической конструкцией, на основе которой построен весь язык. Как таковой, класс образует основу объектно-ориентированного программирования в среде Java. Любая концепция, которую нужно реализовать в Java-программе, должна быть помещена внутрь класса.

В связи с тем, что класс имеет такое большое значение для Java, эта и несколько следующих глав посвящены классам. В этой главе читатели ознакомятся с основными элементами класса и узнают, как можно использовать класс для создания объектов. Читатели ознакомятся также с методами, конструкторами и ключевым словом `this`.

Основы классов

Мы пользовались классами с самого начала этой книги. Однако до сих пор применялась только наиболее примитивная форма класса. Классы, созданные в предшествующих главах, служили только в качестве контейнеров метода `main()`, который мы использовали для ознакомления с основами синтаксиса Java. Как вы вскоре убедитесь, классы предоставляют значительно больше возможностей, чем те, которые были представлены до сих пор.

Вероятно, наиболее важное свойство класса то, что он определяет новый тип данных. После того как он определен, этот новый тип можно применять для создания объектов данного типа. Таким образом, класс — это *шаблон* объекта, а объект — это *экземпляр* класса. Поскольку объект является экземпляром класса, термины *объект* и *экземпляр* часто используются попеременно.

Общая форма класса

При определении класса объявляют его конкретную форму и сущность. Это выполняется путем указания данных, которые он содержит, и кода, воздействующего на эти данные. Хотя очень простые классы могут содержать только код или только данные, большинство классов, применяемых в реальных программах, содержит оба эти компонента. Как будет показано в дальнейшем, код класса определяет интерфейс к его данным.

Для объявления класса служит ключевое слово `class`. Использованные до сих пор классы в действительности представляли собой очень ограниченные примеры полной формы. 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;
     типимя_метода1(список_параметров) {
        // тело метода
     }
     типимя_метода2(список_параметров) {
        // тело метода
   }
   // ...
   типимя_методаN(список_параметров) {
     // тело метода
   }
}
```

Данные, или переменные, определенные внутри класса, называются *переменными экземпляра*. Код содержится внутри *методов*. Определенные внутри класса методы и переменные вместе называют *членами* класса. В большинстве классов действия с переменными экземпляров и доступ к ним выполняются через методы, определенные в этом классе. Таким образом, в общем случае именно методы определяют способ использования данных класса.

Определенные внутри класса переменные называют 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. Вскоре мы вернемся к рассмотрению этой концепции, но она слишком важна, чтобы можно было обойтись без хотя бы предварительного ознакомления с нею.

Все методы имеют ту же общую форму, что и метод `main()`, который мы использовали до сих пор. Однако большинство методов не будут указаны как `static` или `public`. Обратите внимание, что общая форма класса не содержит определения метода `main()`. Классы Java могут и не содержать этот метод. Его обязательно указывать только в тех случаях, когда данный класс служит начальной точкой программы. Более того, аплеты вообще не требуют использования метода `main()`.

На заметку! *Программисты на C++ обратят внимание, что объявление класса и реализация методов хранятся в одном месте, а не определены отдельно. Иногда эта особенность приводит к созданию очень больших файлов* `.java`*, поскольку любой класс должен быть полностью определен в одном файле исходного кода. Такая архитектура была принята для Java умышленно, поскольку разработчики посчитали, что хранение определения, объявления и реализации в одном файле упрощает сопровождение кода в течение длительного периода его эксплуатации.*

Простой класс

Изучение классов начнем с простого примера. Ниже приведен код класса `Box` (Параллелепипед), который определяет три переменных экземпляра: `width` (ширина), `height` (высота) и `depth` (глубина). В настоящий момент `Box` не содержит никаких методов (но вскоре мы добавим в него метод).

```
class Box {
   double width;
   double height;
   double depth;
}
```

Как уже было сказано, класс определяет новый тип данных. В данном случае новый тип данных назван `Box`. Это имя будет использоваться для объявления объектов типа `Box`. Важно помнить, что объявление `class` создает только шаблон, но не действительный объект. Таким образом, приведенный код не приводит к появлению никаких объектов типа `Box`.

Чтобы действительно создать объект `Box`, нужно использовать оператор, подобный следующему:

```
Box mybox = new Box(); // создание объекта mybox типа Box
```

После выполнения этого оператора `mybox` станет экземпляром класса `Box`. То есть он обретет "физическое" существование. Но пока можете не задумываться о нюансах этого оператора.

Повторим еще раз: при каждом создании экземпляра класса мы создаем объект, который содержит собственную копию каждой переменной экземпляра, определенной классом. Таким образом, каждый объект `Box` будет содержать собственные копии переменных экземпляра `width`, `height` и `depth`. Для получения доступа к этим переменным применяется операция *точки* (`.`). Эта операция связывает имя объекта с именем переменной экземпляра. Например, чтобы присвоить переменной `width` объекта `mybox` значение 100, нужно было бы использовать следующий оператор:

```
mybox.width = 100;
```

Этот оператор указывает компилятору, что копии переменной `width`, хранящейся внутри объекта `mybox`, нужно присвоить значение, равное 100. В общем случае операцию точки используют для доступа как к переменным экземпляра, так и к методам внутри объекта.

Ниже приведена полная программа, в которой используется класс `Box`.

```
/* Программа, использующая класс Box.
   Назовите этот файл BoxDemo.java
*/
class Box {
   double width;
   double height;
   double depth;
}
// Этот класс объявляет объект типа Box.
class BoxDemo {
   public static void main(String args[]) {
      Box mybox = new Box();
      double vol;
```

```
      // присваивание значений переменным экземпляра mybox
      mybox.width = 10;
      mybox.height = 20;
      mybox.depth = 15;
      // вычисление объема параллелепипеда
      vol = mybox.width * mybox.height * mybox.depth;
      System.out.println("Объем равен " + vol);
   }
}
```

Файлу этой программы нужно присвоить имя `BoxDemo.java`, поскольку метод `main()` определен в классе, названном `BoxDemo`, а не `Box`. Выполнив компиляцию этой программы, вы убедитесь в создании двух файлов `.class`: одного для `Box` и одного для `BoxDemo`. Компилятор Java автоматически помещает каждый класс в отдельный файл с расширением `.class`. В действительности классы `Box` и `BoxDemo` не обязательно должны быть объявлены в одном и том же исходном файле. Каждый класс можно было бы поместить в отдельный файл, названный соответственно `Box.java` и `BoxDemo.java`.

Чтобы запустить эту программу, нужно выполнить файл `BoxDemo.class`. В результате будет получен следующий вывод:

```
Объем равен 3000.0
```

Как было сказано ранее, каждый объект содержит собственные копии переменных экземпляра. Это означает, что при наличии двух объектов `Box` каждый из них будет содержать собственные копии переменных `depth`, `width` и `height`. Важно понимать, что изменения переменных экземпляра одного объекта не влияют на переменные экземпляра другого. Например, в следующем программе объявлены два объекта `Box`:

```
// Эта программа объявляет два объекта 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;
      // присваивание значений переменным экземпляра mybox1
      mybox1.width = 10;
      mybox1.height = 20;
      mybox1.depth = 15;
      //* присваивание других значений переменным
         экземпляра mybox2's */
      mybox2.width = 3;
      mybox2.height = 6;
      mybox2.depth = 9;
      // вычисление объема первого параллелепипеда
      vol = mybox1.width * mybox1.height * mybox1.depth;
      System.out.println("Объем равен " + vol);
      // вычисление объема второго параллелепипеда
      vol = mybox2.width * mybox2.height * mybox2.depth;
      System.out.println("Volume is " + vol);
   }
}
```

Эта программа генерирует следующий вывод:

```
Объем равен 3000.0
Объем равен 162.0
```

Как видите, данные объекта mybox1 полностью изолированы от данных, содержащихся в объекте mybox2.

Объявление объектов

Как мы уже отмечали, при создании класса вы создаете новый тип данных. Этот тип можно использовать для объявления объектов данного типа. Однако создание объектов класса — двухступенчатый процесс. Вначале необходимо объявить переменную типа класса. Эта переменная не определяет объект. Она представляет собой всего лишь переменную, которая может *ссылаться* на объект. Затем потребуется получить действительную, физическую копию объекта и присвоить ее этой переменной. Это можно выполнить с помощью операции new. Эта операция динамически (т.е. во время выполнения) распределяет память под объект и возвращает ссылку на него. В общих чертах эта ссылка представляет собой адрес объекта в памяти, распределенной операцией new. Затем эта ссылка сохраняется в переменной. Таким образом, в Java все объекты классов должны распределяться динамически. Рассмотрим эту процедуру более подробно.

В приведенном ранее примере программы строка, подобная следующей, используется для объявления объекта типа Box:

```
Box mybox = new Box();
```

Этот оператор объединяет только что описанные шаги. Чтобы каждый из шагов был более очевидным, его можно было переписать следующим образом:

```
Box mybox;           // объявление ссылки на объект
mybox = new Box();   // распределение памяти для объекта Box
```

Первая строка объявляет mybox в качестве ссылки на объект типа Box. После выполнения этой строки переменная mybox содержит значение null, свидетельствующее о том, что она еще не указывает на реальный объект. Любая попытка использования mybox на этом этапе приведет к возникновению ошибки времени компиляции. Следующая строка распределяет память под реальный объект и присваивает переменной mybox ссылку на этот объект. После выполнения второй строки переменную mybox можно использовать, как если бы она была объектом Box. Но в действительности переменная mybox просто содержит адрес памяти реального объекта Box. Эффект выполнения этих двух строк кода показан на рис. 6.1.

На заметку! *Читатели, которые знакомы с языками C/C++, вероятно заметили, что ссылки на объекты выглядят подобно указателям. В общих чертах это впечатление верно. Ссылка на объект похожа на указатель памяти. Основное различие между ними — и основное свойство, обеспечивающее безопасность программ Java — 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.*

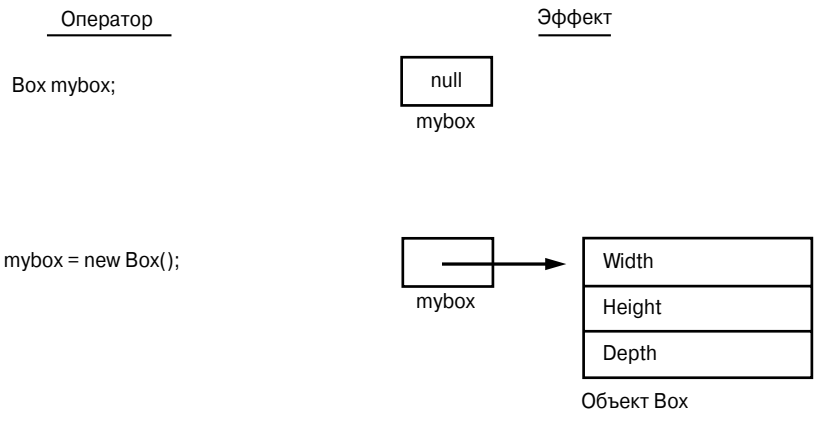

Рис. 6.1. Объявление объекта типа `Box`

Более подробное рассмотрение операции `new`

Как было сказано, операция `new` динамически распределяет память для объекта. Общая форма этой операции имеет следующий вид:

```
переменная_класса = new имя_класса();
```

Здесь *переменная_класса* — переменная создаваемого типа класса. *Имя_класса* — это имя класса, конкретизация которого выполняется. Имя класса, за которым следуют круглые скобки, указывает *конструктор* данного класса. Конструктор определяет действия, выполняемые при создании объекта класса. Конструкторы — важная часть всех классов, и они обладают множеством важных атрибутов. Большинство классов, используемых в реальных программах, явно определяют свои конструкторы внутри своего определения класса. Однако если никакой явный конструктор не указан, Java автоматически предоставит конструктор, используемый по умолчанию. Это же происходит в случае объекта `Box`. Пока мы будем пользоваться конструктором, заданным по умолчанию. Вскоре читатели научатся определять собственные конструкторы.

У читателей может возникнуть вопрос, почему не требуется использовать операцию `new` для таких элементов, как целые числа или символы. Это обусловлено тем, что элементарные типы Java реализованы в не виде объектов, а в виде "обычных" переменных. Это сделано для повышения эффективности. Как вы убедитесь, объекты обладают множеством свойств и атрибутов, которые требуют, чтобы Java-программа обрабатывала их иначе, чем элементарные типы. Отсутствие накладных расходов, связанных с обработкой объектов, при обработке элементарных типов позволяет эффективнее реализовать элементарные типы. Несколько позже мы приведем объектные версии элементарных типов, которые могут пригодиться в ситуациях, когда требуются полноценные объекты этих типов.

Важно понимать, что операция `new` распределяет память для объекта во время выполнения. Преимущество этого подхода состоит в том, что программа может создавать ровно столько объектов, сколько требуется во время ее выполнения. Однако поскольку объем памяти ограничен, возможна ситуация, когда операция `new` не сможет выделить память для объекта из-за ее нехватки. В этом случае возникнет исключение времени выполнения. (Обработка этого и других исключений описана в главе 10.) В примерах программ,

приведенных в этой книге, можно не беспокоиться по поводу недостатка объема памяти, но в реальных программах эту возможность придется учитывать.

Еще раз рассмотрим различие между классом и объектом. Класс создает новый тип данных, который можно использовать для создания объектов. То есть класс создает логический каркас, определяющий взаимосвязь между его членами. При объявлении объекта класса мы создаем экземпляр этого класса. Таким образом, класс — это логическая конструкция. А объект обладает физической сущностью. (То есть объект занимает область в памяти.) Важно помнить об этом различии.

Присваивание переменных объектных ссылок

При выполнении присваивания переменные объектных ссылок действуют иначе, чем можно было бы представить. Например, какие действия, по вашему мнению, выполняет следующий фрагмент?

```
Box b1 = new Box();
Box b2 = b1;
```

Можно подумать, что переменной `b2` присваивается ссылка на копию объекта, на которую ссылается переменная `b1`. То есть может показаться, что `b1` и `b2` ссылаются на отдельные и различные объекты. Однако это не так. После выполнения этого фрагмента кода обе переменные `b1` и `b2` будут ссылаться на *один и тот же* объект. Присваивание `b1` переменной `b2` не привело к распределению какой-то памяти или копированию какой-либо части исходного объекта. Эта операция присваивания приводит лишь к тому, что переменная `b2` ссылается на тот же объект, что и переменная `b1`. Таким образом, любые изменения, выполненные в объекте через переменную `b2`, окажут влияние на объект, на который ссылается переменная `b1`, поскольку это — один и тот же объект.

Эта ситуация отражена на рис. 6.2.

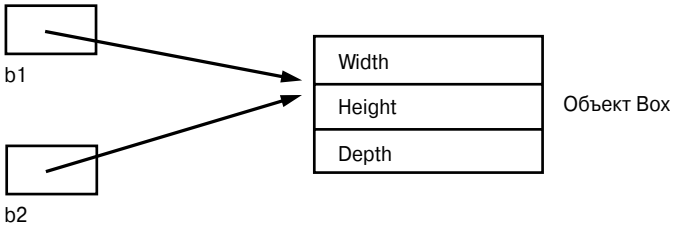

Рис. 6.2. Использование переменных объектных ссылок

Хотя и `b1` и `b2` ссылаются на один и тот же объект, эти переменные не связаны между собой никаким другим образом. Например, следующая операция присваивания значения переменной `b1` просто *разорвет связь* переменной `b1` с исходным объектом, не оказывая влияния на сам объект или на переменную `b2`:

```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

В этом примере значение b1 установлено равным null, но переменная b2 по-прежнему указывает на исходный объект.

Помните! *Присваивание ссылочной переменной одного объекта ссылочной переменной другого объекта не ведет к созданию копии объекта, а лишь создает копию ссылки.*

Знакомство с методами

Как было сказано в начале этой главы, обычно классы состоят из двух элементов: переменных экземпляра и методов. Поскольку язык Java предоставляет им столь большие возможности и гибкость, тема методов очень обширна. Фактически многие последующие главы посвящены методам. Однако чтобы можно было приступить к добавлению методов к своим классам, необходимо ознакомиться с рядом их основных характеристик.

Общая форма объявления метода выглядит следующим образом:

```
тип имя(список_параметров) {
   // тело метода
}
```

Здесь *тип* указывает тип данных, возвращаемых методом. Он может быть любым допустимым типом, в том числе типом класса, созданным программистом. Если метод не возвращает значение, типом его возвращаемого значения должен быть void. *Имя* служит для указания имени метода. Оно может быть любым допустимым идентификатором, кроме тех, которые уже используются другими элементами в текущей области определения. *Список_параметров* — последовательность пар "тип-идентификатор", разделенных запятыми. По сути, параметры — это переменные, которые принимают значения *аргументов*, переданных методу во время его вызова. Если метод не имеет параметров, список параметров будет пустым.

Методы, тип возвращаемого значения которых отличается от void, возвращают значение вызывающей процедуре с помощью следующей формы оператора return:

```
return значение;
```

Здесь *значение* — это возвращаемое значение.

В нескольких последующих разделах мы рассмотрим создание различных типов методов, в том числе как принимающие параметры, так и возвращающие значения.

Добавление метода к классу Box

Хотя было бы весьма удобно создать класс, который содержит только данные, в реальных программах подобное встречается редко. В большинстве случаев для осуществления доступа к переменным экземпляра, определенным классом, придется использовать методы. Фактически методы определяют интерфейсы большинства классов. Это позволяет программисту, который реализует класс, скрывать конкретную схему внутренних структур данных за более понятными абстракциями метода. Кроме определения методов, которые обеспечивают доступ к данным, можно определять также методы, используемые внутренне самим классом.

Теперь приступим к добавлению метода в класс Box. Просматривая предшествующие программы, легко прийти к выводу, что класс Box мог бы лучше справиться с вычислением объема параллелепипеда, чем класс BoxDemo. В конце концов, поскольку объем параллелепипеда зависит от его размеров, вполне логично, чтобы его вычисление выполнялось

в классе Box. Для этого в класс Box нужно добавить метод, как показано в следующем примере:

```
// Эта программа содержит метод внутри класса box.
class Box {
   double width;
   double height;
   double depth;
   // отображение объема 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();
      // присваивание значений переменным экземпляра mybox1
      mybox1.width = 10;
      mybox1.height = 20;
      mybox1.depth = 15;
      /* присваивание других значений переменным
         экземпляра mybox2 */
      mybox2.width = 3;
      mybox2.height = 6;
      mybox2.depth = 9;
      // отображение объема первого параллелепипеда
      mybox1.volume();
      // отображение объема второго параллелепипеда
      mybox2.volume();
   }
}
```

Эта программа генерирует следующий вывод, совпадающий с выводом предыдущей версии:

```
Объем равен 3000.0
Объем равен 162.0
```

Внимательно взгляните на следующие две строки кода:

```
mybox1.volume();
mybox2.volume();
```

В первой строке присутствует обращение к методу volume(), определенному в mybox1. То есть она вызывает метод volume() по отношению к объекту mybox1, для чего было использовано имя объекта, за которым следует символ операции точки. Таким образом, обращение к mybox1.volume() отображает объем параллелепипеда, определенного объектом mybox1, а обращение к mybox2.volume() — объем параллелепипеда, определенного объектом mybox2. При каждом вызове метода volume() он отображает объем указанного параллелепипеда.

Соображения, приведенные в следующих абзацах, облегчат понимание концепции вызова метода. При вызове метода mybox1.volume() система времени выполнения Java передает управление коду, определенному внутри метода volume(). По завершении выполнения всех операторов внутри метода управление возвращается вызывающей программе

и ее выполнение продолжается со строки, которая следует за вызовом метода. В самом общем смысле можно сказать, что метод — способ реализации подпрограмм в Java.

В методе `volume()` следует обратить внимание на один очень важный нюанс: ссылка на переменные экземпляра `width`, `height` и `depth` выполняется непосредственно, без указания перед ними имени объекта или операции точки. Когда метод использует переменную экземпляра, которая определена его классом, он выполняет это непосредственно, без указания явной ссылки на объект и без применения операции точки. Это становится понятным, если немного подумать. Метод всегда вызывается по отношению к какому-то объекту его класса. Как только этот вызов выполнен, объект известен. Таким образом, внутри метода вторичное указание объекта совершенно излишне. Это означает, что `width`, `height` и `depth` неявно ссылаются на копии этих переменных, хранящиеся в объекте, который вызывает метод `volume()`.

Подведем краткие итоги. Когда обращение к переменной экземпляра выполняется кодом, не являющимся частью класса, в котором определена переменная экземпляра, оно должно выполняться посредством объекта с применением операции точки. Однако когда это обращение осуществляется кодом, который является частью того же класса, где определена переменная экземпляра, ссылка на переменную может выполняться непосредственно. Эти же правила применимы и к методам.

Возвращение значения

Хотя реализация метода `volume()` переносит вычисление объема параллелепипеда внутрь класса `Box`, которому принадлежит этот метод, такой способ вычисления не является наилучшим. Например, что делать, если другой часть программы требуется знание объема параллелепипеда без его отображения? Более рациональный способ реализации метода `volume()` — вычисление объема параллелепипеда и возврат результата вызывающему объекту. Следующий пример — усовершенствованная версия предыдущей программы — выполняет именно эту задачу.

```
// Теперь метод volume() возвращает объем параллелепипеда.
class Box {
   double width;
   double height;
   double depth;
   // вычисление и возвращение объема
   double volume() {
      return width * height * depth;
   }
}
class BoxDemo4 {
   public static void main(String args[]) {
      Box mybox1 = new Box();
      Box mybox2 = new Box();
      double vol;
      // присваивание значений переменным экземпляра mybox1
      mybox1.width = 10;
      mybox1.height = 20;
      mybox1.depth = 15;
      /* присваивание других значений переменным экземпляра mybox2 */
      mybox2.width = 3;
      mybox2.height = 6;
      mybox2.depth = 9;
```

```
        // получение объема первого параллелепипеда
        vol = mybox1.volume();
        System.out.println("Объем равен " + vol);
        // получение объема второго параллелепипеда
        vol = mybox2.volume();
        System.out.println("Объем равен " + vol);
    }
}
```

Как видите, вызов метода `volume()` выполняется в правой части оператора присваивания. Правой частью этого оператора является переменная, в данном случае `vol`, которая будет принимать значение, возвращенное методом `volume()`. Таким образом, после выполнения такого оператора:

```
vol = mybox1.volume();
```

значение `mybox1.volume()` равно 3000, и этот объем сохраняется в переменной `vol`.

При работе с возвращаемыми значениями следует учитывать два важных обстоятельства.

- Тип данных, возвращаемых методом, должен быть совместим с возвращаемым типом, указанным методом. Например, если возвращаемым типом какого-либо метода является `boolean`, нельзя возвращать целочисленное значение.
- Переменная, принимающая возвращенное методом значение (такая как `vol` в данном случае), также должна быть совместима с возвращаемым типом, указанным для метода.

И еще один нюанс: предыдущую программу можно было бы записать в несколько более эффективной форме, поскольку в действительности переменная `vol` совершенно не нужна. Обращение к методу `volume()` можно было бы использовать в операторе `println()` непосредственно, как в следующей строке кода:

```
System.out.println("Объем равен " + mybox1.volume());
```

В этом случае при выполнении оператора `println()` метод `mybox1.volume()` будет вызываться автоматически, а возвращаемое им значение будет передаваться методу `println()`.

Добавление метода, принимающего параметры

Хотя некоторые методы не нуждаются в параметрах, большинство требует их передачи. Параметры позволяют обобщать метод. То есть метод с параметрами может работать с различными данными и/или применяться в ряде несколько различных ситуаций. В качестве иллюстрации рассмотрим очень простой пример. Ниже показан метод, который возвращает квадрат числа 10.

```
int square()
{
  return 10 * 10;
}
```

Хотя этот метод действительно возвращает 10^2, его применение очень ограничено. Однако если его изменить так, чтобы он принимал параметр, как показано в следующем примере, метод `square()` может стать значительно более полезным.

```
int square(int i)
{
  return i * i;
}
```

Теперь метод `square()` будет возвращать квадрат любого значения, с которым он вызван. То есть теперь метод `square()` является методом общего назначения, который может вычислять квадрат любого целочисленного значения, а не только числа 10.

Приведем примеры:

```
int x, y;
x = square(5); // x равно 25
x = square(9); // x равно 81
y = 2;
x = square(y); // x равно 4
```

В первом обращении к методу `square()` значение 5 будет передано параметру `i`. Во втором обращении параметр `i` примет значение, равное 9. Третий вызов метода передает значение переменной `y`, которое в этом примере составляет 2. Как видно из этих примеров, метод `square()` способен возвращать квадрат любых переданных ему данных.

Важно различать два термина: *параметр* и *аргумент*. *Параметр* — это переменная, определенная методом, которая принимает значение при вызове метода. Например, в методе `square()` параметром является `i`. *Аргумент* — это значение, передаваемое методу при его вызове. Например, `aquare(100)` передает 100 в качестве аргумента. Внутри метода `square()` параметр `i` получает это значение.

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;
```

Хотя этот код работает, он не очень удобен по двум причинам. Во-первых, он громоздок и чреват ошибками. Например, вполне можно забыть определить один из размеров. Во-вторых, в правильно спроектированных Java-программах доступ к переменным экземпляра должен осуществляться только через методы, определенные их классом. В будущем поведение метода можно изменить, но нельзя изменить поведение раскрытой переменной экземпляра.

Поэтому более рациональный подход установки размеров параллелепипеда — создание метода, который принимает размеры параллелепипеда в виде своих параметров и соответствующим образом устанавливает значение каждой переменной экземпляра. Эта концепция реализована в приведенной ниже программе.

```
// Эта программа использует метод с параметрами.
class Box {
   double width;
   double height;
   double depth;

   // вычисление и возвращение объема
   double volume() {
      return width * height * depth;
   }
```

```
    // установка размеров 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;
      // инициализация каждого экземпляра Box
      mybox1.setDim(10, 20, 15);
      mybox2.setDim(3, 6, 9);
      // получение объема первого параллелепипеда
      vol = mybox1.volume();
      System.out.println("Объем равен " + vol);
      // получение объема второго параллелепипеда
      vol = mybox2.volume();
      System.out.println("Объем равен " + vol);
    }
}
```

Как видите, метод `setDim()` использован для установки размеров каждого параллелепипеда. Например, при выполнении такого оператора:

```
mybox1.setDim(10, 20, 15);
```

значение 10 копируется в параметр `w`, 20 — в `h` и 15 — в `d`. Затем внутри метода `setDim()` значения `w`, `h` и `d` присваиваются соответственно переменным `width`, `height` и `depth`.

Многим читателям представленные в предшествующих разделах концепции будут знакомы. Однако если вы еще не знакомы с такими понятиями, как вызовы методов, аргументы и параметры, можете немного поэкспериментировать с ними, прежде чем продолжить изучение материала, изложенного в последующих разделах. Концепции 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 функций, предназначенных для увеличения удобства работы, таких как `setDim()`, было бы проще и удобнее, если бы все действия по установке переменных выполнялись при первом создании объекта. Поскольку необходимость инициализации возникает столь часто, Java позволяет объектам выполнять собственную инициализацию при их создании. Эта автоматическая инициализация осуществляется с помощью конструктора.

Конструктор инициализирует объект непосредственно во время создания. Его имя совпадает с именем класса, в котором он находится, а синтаксис аналогичен синтаксису метода. Как только он определен, конструктор автоматически вызывается непосредственно после создания объекта, перед завершением выполнения операции `new`. Конструкторы выглядят несколько непривычно, поскольку не имеют ни возвращаемого типа, ни даже

типа void. Это обусловлено тем, что неявно заданный возвращаемый тип конструктора класса — тип самого класса. Именно конструктор инициализирует внутреннее состояние объекта так, чтобы код, создающий экземпляр, с самого начала содержал полностью инициализированный, пригодный к использованию объект.

Пример класса Box можно изменить, чтобы значения размеров параллелепипеда присваивались при конструировании объекта. Для этого потребуется заменить метод setDim() конструктором. Вначале определим простой конструктор, который просто устанавливает одинаковые значения размеров для всех параллелепипедов. Эта версия программы имеет вид:

```
/* В этом примере класс Box использует конструктор
   для инициализации размеров параллелепипеда.
*/
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;
   }
   // вычисление и возвращение объема
   double volume() {
      return width * height * depth;
   }
}
class BoxDemo6 {
   public static void main(String args[]) {
      // объявление, распределение и инициализация объектов Box
      Box mybox1 = new Box();
      Box mybox2 = new Box();
      double vol;
      // получение объема первого параллелепипеда
      vol = mybox1.volume();
      System.out.println("Объем равен " + vol);
      // получение объема второго параллелепипеда
      vol = mybox2.volume();
      System.out.println("Объем равен " + vol);
   }
}
```

Эта программа генерирует следующий вывод:

```
Конструирование объекта Box
Конструирование объекта Box
Объем равен 1000.0
Объем равен 1000.0
```

Как видите, и mybox1, и mybox2 были инициализированы конструктором Box() при их создании. Поскольку конструктор присваивает всем параллелепипедам одинаковые размеры 10×10×10, и mybox1, и mybox2 будут иметь одинаковый объем. Оператор println() внутри конструктора Box() служит исключительно иллюстративным целям.

Большинство конструкторов не выводят никакой информации, а лишь выполняют инициализацию объекта.

Прежде чем продолжить, еще раз рассмотрим операцию `new`. Как вы уже знаете, при распределении памяти для объекта используют следующую общую форму:

```
переменная_класса = new имя_класса();
```

Теперь вам должно быть ясно, почему после имени класса требуются круглые скобки. В действительности этот оператор вызывает конструктор класса. Таким образом, в строке:

```
Box mybox1 = new Box();
```

операция `new Box()` вызывает конструктор `Box()`. Если конструктор класса не определен явно, Java создает для класса конструктор, который будет использоваться по умолчанию. Именно поэтому приведенная строка кода работала в предшествующих версиях класса `Box`, в которых конструктор не был определен. Конструктор, используемый по умолчанию, инициализирует все переменные экземпляра нулевыми значениями. Зачастую конструктора, используемого по умолчанию, вполне достаточно для простых классов, чего обычно нельзя сказать о более сложных. Как только конструктор определен, конструктор, заданный по умолчанию, больше не используется.

Конструкторы с параметрами

Хотя в предыдущем примере конструктор `Box()` инициализирует объект `Box`, он не особенно полезен — все параллелепипеды получают одинаковые размеры. Следовательно, необходим способ конструирования объектов `Box` с различными размерами. Простейшее решение этой задачи — добавление к конструктору параметров. Как легко догадаться, это делает конструктор значительно более полезным. Например, следующая версия класса `Box` определяет конструктор с параметрами, который устанавливает размеры параллелепипеда в соответствии со значениями этих параметров. Обратите особое внимание на способ создания объектов `Box`.

```
/* В этой программе класс Box использует конструктор с параметрами
   для инициализации размеров параллелепипеда.
*/
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;
   }
   // вычисление и возврат объема
   double volume() {
      return width * height * depth;
   }
}
class BoxDemo7 {
   public static void main(String args[]) {
      // объявление, распределение и инициализация объектов Box
      Box mybox1 = new Box(10, 20, 15);
```

```
    Box mybox2 = new Box(3, 6, 9);
    double vol;
    // получение объема первого параллелепипеда
    vol = mybox1.volume();
    System.out.println("Объем равен " + vol);
    // получение объема 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 со значениями, указанными в параметрах его конструктора. Например, в следующей строке:

```
Box mybox1 = new Box(10, 20, 15);
```

значения 10, 20 и 15 передаются конструктору `Box()` при создании объекта с помощью операции `new`. Таким образом, копии переменных `width`, `height` и `depth` будут содержать соответственно значения 10, 20 и 15.

Ключевое слово `this`

Иногда будет требоваться, чтобы метод ссылался на вызвавший его объект. Чтобы это было возможно, в Java определено ключевое слово `this`. Оно может использоваться внутри любого метода для ссылки на *текущий* объект. То есть `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 пояснения 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 предыдущая. Применение ключевого слова `this` избыточно, но совершенно правильно. Внутри метода `Box()` ключевое слово `this` всегда будет ссылаться на вызывающий объект. Хотя в данном случае это и излишне, в других случаях, один из которых рассмотрен в следующем разделе, ключевое слово `this` весьма полезно.

Сокрытие переменной экземпляра

Как вы знаете, в Java не допускается объявление двух локальных переменных с одним и тем же именем в одной и той же или во включающих одна другую областях определения. Интересно отметить, что могут существовать локальные переменные, в том числе формальные параметры методов, которые перекрываются с именами переменных экземпляра класса. Однако когда имя локальной переменной совпадает с именем переменной экземпляра, локальная переменная *скрывает* переменную экземпляра. Именно

поэтому внутри класса Box переменные width, height и depth не были использованы в качестве имен параметров конструктора Box(). В противном случае переменная width ссылалась бы на формальный параметр, скрывая переменную экземпляра width. Хотя обычно проще использовать различные имена, существует и другой способ выхода из подобной ситуации. Поскольку ключевое слово this позволяет ссылаться непосредственно на объект, его можно применять для разрешения любых конфликтов пространства имен, которые могут возникать между переменными экземпляра и локальными переменными. Например, ниже показана еще одна версия метода Box(), в которой имена width, height и depth использованы в качестве имен параметров, а ключевое слово this служит для обращения к переменным экземпляра по этим же именам.

```
// Этот код служит для разрешения конфликтов пространства имен.
Box(double width, double height, double depth) {
   this.width = width;
   this.height = height;
   this.depth = depth;
}
```

Небольшое предостережение: иногда подобное применение ключевого слова this может приводить к недоразумениям, и некоторые программисты стараются не применять имена локальных переменных и параметров, скрывающие переменные экземпляров. Конечно, множество программистов придерживаются противоположного мнения и считают целесообразным в целях облегчения понимания программ использовать одни и те же имена, а для предотвращения скрытия переменных экземпляров применяют ключевое слово this. Выбор того или иного подхода — дело личного вкуса.

Сборка мусора

Поскольку распределение памяти для объектов осуществляется динамически посредством операции new, у читателей может возникнуть вопрос, как уничтожаются такие объекты и как их память освобождается для последующего распределения. В некоторых языках, подобных C++, динамически распределенные объекты нужно освобождать вручную с помощью операции delete. В Java применяется другой подход. Освобождение памяти выполняется автоматически. Используемая для выполнения этой задачи технология называется *сборкой мусора*. Процесс проходит следующим образом: при отсутствии каких-либо ссылок на объект программа заключает, что этот объект больше не нужен, и занимаемую объектом память можно освободить. В Java не нужно явно уничтожать объекты, как это делается в C++. Во время выполнения программы сборка мусора выполняется только изредка (если вообще выполняется). Она не будет выполняться лишь потому, что один или несколько объектов существуют и больше не используются. Более того, в различных реализациях системы времени выполнения Java могут применяться различные подходы к сборке мусора, но в большинстве случаев при написании программ об этом можно не беспокоиться.

Метод `finalize()`

Иногда при уничтожении объект должен будет выполнять какое-либо действие. Например, если объект содержит какой-то ресурс, отличный от ресурса Java (вроде файлового дескриптора или шрифта), может требоваться гарантия освобождения этих ресурсов перед уничтожением объекта. Для подобных ситуаций Java предоставляет механизм,

называемый *финализацией*. Используя финализацию, можно определить конкретные действия, которые будут выполняться непосредственно перед удалением объекта сборщиком мусора.

Чтобы добавить в класс средство выполнения финализации, достаточно определить метод `finalize()`. Среда времени выполнения Java вызывает этот метод непосредственно перед удалением объекта данного класса. Внутри 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 со стороны какого-либо текущего состояния, так и косвенные ссылки через другие ссылочные объекты. Непосредственно перед освобождением ресурсов среда времени выполнения Java вызывает метод `finalize()` по отношению к объекту.

Общая форма метода `finalize()` имеет следующий вид:

```
protected void finalize( )
{
// здесь должен находиться код финализации
}
```

В этой синтаксической конструкции ключевое слово `protected` — спецификатор, который предотвращает доступ к методу `finalize()` со стороны кода, определенного вне его класса. Этот и другие спецификаторы доступа описаны в главе 7.

Важно понимать, что метод `finalize()` вызывается только непосредственно перед сборкой мусора. Например, он не вызывается при выходе объекта за рамки области определения. Это означает, что неизвестно, когда будет — и, даже будет ли вообще — выполняться метод `finalize()`. Поэтому программа должна предоставлять другие средства освобождения используемых объектом системных ресурсов и тому подобного. Нормальная работа программы не должна зависеть от метода `finalize()`.

На заметку! *Те читатели, которые знакомы с языком C++, знают, что он позволяет определять деструктор класса, который вызывается при выходе объекта за пределы области определения. Java не поддерживает эту концепцию и не допускает использование деструкторов. По своему функционированию метод `finalaize()` лишь отдаленно напоминает деструктор. По мере приобретения опыта программирования на Java вы убедитесь, что благодаря наличию подсистемы сборки мусора потребность в функциях деструктора очень незначительна.*

Класс `Stack`

Хотя класс `Box` удобен для иллюстрации основных элементов класса, его практическая ценность невелика. Чтобы читатели могли убедиться в реальных возможностях классов, изложение материала этой главы мы завершим более сложным примером. Как вы, возможно, помните из материала по основам объектно-ориентированного программирования (ООП), представленного в главе 2, одно из наибольших преимуществ ООП — это инкапсуляция данных и кода, который манипулирует этими данными. Как было показано, в языке Java класс — это механизм достижения инкапсуляции. Создавая класс, вы создаете новый тип данных, который определяет как сущность данных, которыми будет выполняться манипулирование, так и используемые для этого процедуры. Далее методы задают целостный и управляемый интерфейс к данным класса. Таким образом, класс можно использовать посредством его методов, не заботясь о нюансах его реализации или

о действительном способе управления данными внутри класса. В определенном смысле класс подобен "машине данных". Чтобы использовать машину посредством ее элементов управления, не требуются никакие знания о происходящем внутри нее. Фактически, поскольку подробности реализации сокрыты, внутренние детали можно изменять по мере необходимости. До тех пор, пока код использует класс через его методы, внутренние детали могут меняться, не вызывая побочных эффектов за пределами класса.

В качестве иллюстрации приведенных соображений рассмотрим один из типичных примеров инкапсуляции: стек. *Стек* хранит данные в порядке "первым вошел, последним вышел". То есть стек подобен стопке тарелок на столе — тарелка, которая была поставлена на стол первой, будет использована последней. Стеки управляются посредством двух операций, которые традиционно называются *заталкиванием* (в стек) и *выталкиванием* (из стека). Для помещения элемента на верхушку стека используется операция заталкивания. Для извлечения элемента из стека применяется операция выталкивания. Как вы убедитесь, инкапсуляции всего механизма стека не представляет сложности.

Ниже приведен код класса, названного `Stack`, который реализует стек целочисленных значений.

```
// Этот класс определяет целочисленный стек, который может хранить 10 значений.
class Stack {
   int stck[] = new int[10];
   int tos;
   // Инициализация верхушки стека
   Stack() {
      tos = -1;
   }
   // Заталкивание элемента в стек
   void push(int item) {
      if(tos==9)
         System.out.println("Стек полон.");
      else
         stck[++tos] = item;
   }
   // Выталкивание элемента из стека
   int pop() {
      if(tos < 0) {
         System.out.println("Стек не загружен.");
         return 0;
      }
      else
         return stck[tos--];
   }
}
```

Как видите, класс `Stack` определяет два элемента данных и три метода. Стек целочисленных значений хранится в массиве `stck`. Этот массив индексируется по переменной `tos`, которая всегда содержит индекс верхушки стека. Конструктор `Stack()` инициализирует `tos` значением -1, которое указывает на пустой стек. Метод `push()` помещает элемент в стек. Чтобы извлечь элемент, нужно вызвать метод `pop()`. Поскольку доступ к стеку осуществляется посредством методов `push()` и `pop()`, то, что стек хранится в массиве, в действительности не имеет значения при работе со стеком. Например, стек мог бы храниться в более сложной структуре данных вроде связного списка, но интерфейс, определенный методами `push()` и `pop()`, оставался бы неизменным.

Приведенный в следующем примере класс `TestStack` демонстрирует применение класса `Stack`. Он создает два целочисленных стека, заталкивает в каждый из них определенные значения, а затем выталкивает их из стека.

```
class TestStack {
  public static void main(String args[]) {
    Stack mystack1 = new Stack();
    Stack mystack2 = new Stack();

    // заталкивает числа в стек
    for(int i=0; i<10; i++) mystack1.push(i);
    for(int i=10; i<20; i++) mystack2.push(i);

    // выталкивает 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());
  }
}
```

Эта программа генерирует следующий вывод:

```
Стек в mystack1:
9
8
7
6
5
4
3
2
1
0
Стек в mystack2:
19
18
17
16
15
14
13
12
11
10
```

Как видите, содержимое обоих стеков различается.

И последнее замечание по поводу класса `Stack`. В том виде, каком он реализован, массив `stck`, который содержит стек, может быть изменен кодом, определенным вне класса `Stack`. Это делает класс `Stack` уязвимым для злоупотреблений и повреждений. В следующей главе будет показано, как можно исправить эту ситуацию.

7 ГЛАВА

Более пристальный взгляд на методы и классы

В этой главе мы продолжим рассмотрение методов и классов, начатое в предыдущей главе. В начале мы рассмотрим несколько вопросов, связанных с методами, в том числе перегрузку, передачу параметров и рекурсию. Затем мы снова обратимся к классам и рассмотрим управление доступом, использование ключевого слова `static` и один из наиболее важных встроенных классов Java — `String`.

Перегрузка методов

Java разрешает определение внутри одного класса двух или более методов с одним именем, если только объявления их параметров различны. В этом случае методы называют *перегруженными*, а процесс — *перегрузкой методов*. Перегрузка методов — один из способов поддержки полиморфизма в Java. Тем читателям, которые никогда не использовали язык, допускающий перегрузку методов, эта концепция вначале может показаться странной. Но, как вы вскоре убедитесь, перегрузка методов — одна из наиболее впечатляющих и полезных функциональных возможностей Java.

При вызове перегруженного метода для определения нужной версии Java использует тип и/или количество аргументов метода. Следовательно, перегруженные методы должны различаться по типу и/или количеству их параметров. Хотя возвращаемые типы перегруженных методов могут быть различны, самого возвращаемого типа не достаточно для различения двух версий метода. Когда Java встречает вызов перегруженного метода, она просто выполняет ту его версию, параметры которой соответствуют аргументам, использованным в вызове.

Следующий простой пример иллюстрирует перегрузку метода.

```
// Демонстрация перегрузки методов.
class OverloadDemo {
  void test() {
    System.out.println("Параметры отсутствуют");
  }
```

```
  // Проверка перегрузки на наличие одного целочисленного параметра.
  void test(int a) {
    System.out.println("a: " + a);
  }
  // Проверка перегрузки на наличие двух целочисленных параметров.
  void test(int a, int b) {
    System.out.println("a и b: " + a + " " + b);
  }
  // Проверка перегрузки на 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;
    // вызов всех версий метода test()
    ob.test();
    ob.test(10);
    ob.test(10, 20);
    result = ob.test(123.25);
    System.out.println("Результат ob.test(123.25): " + result);
  }
}
```

Эта программа генерирует следующий вывод:

```
Параметры отсутствуют
a: 10
a и b: 10 20
double a: 123.25
Результат ob.test(123.25): 15190.5625
```

Как видите, метод `test()` перегружается четыре раза. Первая версия не принимает никаких параметров, вторая принимает один целочисленный параметр, третья — два целочисленных параметра, а четвертая — один параметр типа `double`. То, что четвертая версия метода `test()` возвращает также значение, не имеет никакого значения для перегрузки, поскольку возвращаемый тип никак не влияет на разрешение перегрузки.

При вызове перегруженного метода Java ищет соответствие между аргументами, которые были использованы для вызова метода, и параметрами метода. Однако это соответствие не обязательно должно быть полным. В некоторых случаях к разрешению перегрузки может применяться автоматическое преобразование типов Java. Например, рассмотрим следующую программу:

```
// Применение автоматического преобразования типов к перегрузке.
class OverloadDemo {
  void test() {
    System.out.println("Параметры отсутствуют");
  }
  // Проверка перегрузки на наличие двух целочисленных параметров.
  void test(int a, int b) {
    System.out.println("a и b: " + a + " " + b);
  }
```

```
  // Проверка перегрузки на наличие параметра типа double
  void test(double a) {
    System.out.println("Внутреннее преобразование 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); // этот оператор вызовет test(double)
    ob.test(123.2); // этот оператор вызовет test(double)
  }
}
```

Программа генерирует следующий вывод:

```
Параметры отсутствуют
a и b: 10 20
Внутреннее преобразование test(double) a: 88
Внутреннее преобразование test(double) a: 123.2
```

Как видите, эта версия класса `OverloadDemo` не определяет перегрузку `test(int)`. Поэтому при вызове метода `test()` с целочисленным аргументом внутри класса `Overload` какой-то соответствующий метод отсутствует. Однако Java может автоматически преобразовывать тип `integer` в тип `double`, и это преобразование может использоваться для разрешения вызова. Поэтому после того, как версия `test(int)` не обнаружена, Java повышает тип `i` до `double`, а затем вызывает метод `test(double)`. Конечно, если бы метод `test(int)` был определен, вызвался бы он. Java будет использовать автоматическое преобразование типов только при отсутствии полного соответствия.

Перегрузка методов поддерживает полиморфизм, поскольку это один из способов реализации в Java концепции “один интерфейс, несколько методов”. Для пояснения приведем следующие рассуждения. В тех языках, которые не поддерживают перегрузку методов, каждому методу должно быть присвоено уникальное имя. Однако часто желательно реализовать, по сути, один и тот же метод для различных типов данных. Например, рассмотрим функцию вычисления абсолютного значения. Обычно в языках, которые не поддерживают перегрузку, существует три или более версии этой функции со слегка различающимися именами. Например, в C функция `abs()` возвращает абсолютное значение значения типа `integer`, `labs()` — значения типа `long integer`, а `fabs()` — значения с плавающей точкой. Поскольку язык C не поддерживает перегрузку, каждая функция должна обладать собственным именем, несмотря на то, что все три функции выполняют по существу одно и то же действие. В результате в концептуальном смысле ситуация становится более сложной, чем она есть на самом деле. Хотя каждая из функций построена на основе одной и той же концепции, программист вынужден помнить три имени. В Java подобная ситуация не возникает, поскольку все методы вычисления абсолютного значения могут использовать одно и то же имя. И действительно, стандартная библиотека классов Java содержит метод вычисления абсолютного значения, названный `abs()`. Перегрузки этого метода для обработки всех численных типов определены в Java-классе `Math`. Java выбирает для вызова нужную версию метода `abs()` в зависимости от типа аргумента.

Перегрузка ценна тем, что она позволяет обращаться к схожим методам по общему имени. Таким образом, имя `abs` представляет *общее действие*, которое должно выполняться. Выбор нужной *конкретной* версии для данной ситуации — задача компилятора. Программисту нужно помнить только об общем выполняемом действии. Полиморфизм позволяет свести несколько имен к одному. Хотя приведенный пример весьма прост, если эту концепцию расширить, легко убедиться в том, что перегрузка может облегчить выполнение более сложных задач.

При перегрузке метода каждая версия этого метода может выполнять любые необходимые действия. Не существует никакого правила, в соответствии с которым перегруженные методы должны быть связаны между собой. Однако со стилистической точки зрения перегрузка методов предполагает определенную связь. Таким образом, хотя одно и то же имя можно использовать для перегрузки несвязанных методов, поступать так не следует. Например, имя `sqr` можно было бы использовать для создания методов, которые возвращают *квадрат* целочисленного значения и *квадратный корень* значения с плавающей точкой. Но эти две операции принципиально различны. Такое применение перегрузки методов противоречит ее исходному назначению. В частности, следует перегружать только тесно связанные операции.

Перегрузка конструкторов

Наряду с перегрузкой обычных методов можно также выполнять перегрузку методов конструкторов. Фактически перегруженные конструкторы станут нормой, а не исключением, для большинства классов, которые вам придется создавать для реальных программ. Чтобы это утверждение было понятным, вернемся к классу `Box`, разработанному в предыдущей главе. Его последняя версия имеет следующий вид:

```
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;
   }
   // вычисление и возврат значения
   double volume() {
      return width * height * depth;
   }
}
```

Как видите, конструктор `Box()` требует передачи трех параметров. Это означает, что все объявления объектов `Box` должны передавать конструктору `Box()` три аргумента. Например, следующий оператор недопустим:

```
Box ob = new Box();
```

Поскольку конструктор `Box()` требует передачи трех аргументов, его вызов без аргументов — ошибка. Эта ситуация порождает три важных вопроса. Что если нужно было просто определить параллелепипед и его начальные размеры не имели значения (или не были известны)? Или, нужно иметь возможность инициализировать куб, указывая толь-

ко один размер, который должен использоваться для всех трех измерений? При текущем определении класса Box все эти дополнительные возможности недоступны.

К счастью, решение подобных проблем достаточно просто: достаточно перегрузить конструктор Box, чтобы он учитывал только что описанные ситуации. Ниже приведена программа, которая содержит усовершенствованную версию класса Box, выполняющую эту задачу.

```
/* В этом примере класс Box определяет три конструктора для
   инициализации размеров параллелепипеда различными способами.
*/
class Box {
   double width;
   double height;
   double depth;
   // конструктор, используемый при указании всех измерений
   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;     // значение -1 используется для указания
      height = -1;    // неинициализированного
      depth = -1;     // параллелепипеда
   }
   // конструктор, используемый при создании куба
   Box(double len) {
      width = height = depth = len;
   }
   // вычисление и возврат объема
   double volume() {
      return width * height * depth;
   }
}
class OverloadCons {
   public static void main(String args[]) {
      // создание параллелепипедов с применением различных конструкторов
      Box mybox1 = new Box(10, 20, 15);
      Box mybox2 = new Box();
      Box mycube = new Box(7);
      double vol;
      // получение объема первого параллелепипеда
      vol = mybox1.volume();
      System.out.println("Объем mybox1 равен " + vol);
      // получение объема второго параллелепипеда
      vol = mybox2.volume();
      System.out.println("Объем mybox2 равен " + vol);
      // получение объема куба
      vol = mycube.volume();
      System.out.println("Объем mycube равен " + vol);
   }
}
```

Эта программа создает следующий вывод:

```
Объем mybox1 равен 3000.0
Объем mybox2 равен -1.0
Объем mycube равен 343.0
```

Как видите, соответствующий перегруженный конструктор вызывается в зависимости от параметров, указанных при выполнении операции `new`.

Использование объектов в качестве параметров

До сих пор в качестве параметров методов мы использовали только простые типы. Однако передача методам объектов — и вполне допустима, и достаточно распространена. Например, рассмотрим следующую короткую программу:

```
// Методам можно передавать объекты.
class Test {
  int a, b;
  Test(int i, int j) {
    a = i;
    b = j;
  }
  // возврат значения true, если параметр o равен вызывающему объекту
  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 создает следующий вывод:

```
ob1 == ob2: true
ob1 == ob3: false
```

Как видите, метод `equals()` внутри метода `Test` проверяет равенство двух объектов и возвращает результат этой проверки. То есть он сравнивает вызывающий объект с тем, который был ему передан. Если они содержат одинаковые значения, метод возвращает значение `true`. В противном случае он возвращает значение `false`. Обратите внимание, что параметр `o` в методе `equals()` указывает `Test` в качестве типа. Хотя `Test` — тип класса, созданный программой, он используется совершенно так же, как встроенные типы Java.

Одно из наиболее часто встречающихся применений объектов-параметров — в конструкторах. Часто приходится создавать новый объект так, чтобы вначале он не отличался от какого-то существующего объекта. Для этого потребуется определить конструктор, который в качестве параметра принимает объект его класса. Например, следующая версия класса `Box` позволяет выполнять инициализацию одного объекта другим:

```
// В этой версии Box допускает инициализацию одного объекта другим.
class Box {
  double width;
  double height;
  double depth;
  // Обратите внимание на этот конструктор. Он использует объект типа Box.
  Box(Box ob) { // передача объекта конструктору
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }
  // конструктор, используемый при указании всех измерений
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }
  // конструктор, используемый, если ни одно из изменений не указано
  Box() {
    width = -1;     // значение -1 используется для указания
    height = -1;    // не инициализированного
    depth = -1;     // параллелепипеда
  }
  // конструктор, используемый при создании куба
  Box(double len) {
    width = height = depth = len;
  }
  // вычисление и возврат объема
  double volume() {
    return width * height * depth;
  }
}
class OverloadCons2 {
  public static void main(String args[]) {
    // создание параллелепипедов с применением различных конструкторов
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box();
    Box mycube = new Box(7);
    Box myclone = new Box(mybox1); // создание копии объекта mybox1
    double vol;
    // получение объема первого параллелепипеда
    vol = mybox1.volume();
    System.out.println("Объем mybox1 равен " + vol);
    // получение объема второго параллелепипеда
    vol = mybox2.volume();
    System.out.println("Объем mybox2 равен " + vol);
    // получение объема куба
    vol = mycube.volume();
    System.out.println("Объем куба равен " + vol);
    // получение объема клона
    vol = myclone.volume();
    System.out.println("Объем клона равен " + vol);
  }
}
```

Как вы убедитесь, приступив к созданию собственных классов, чтобы объекты можно было конструировать удобным и эффективным образом, нужно располагать множеством форм конструкторов.

Более пристальный взгляд на передачу аргументов

В общем случае существует два способа, которыми компьютерный язык может передавать аргументы подпрограмме. Первый способ — *вызов по значению*. При использовании этого подхода *значение* аргумента копируется в формальный параметр подпрограммы. Следовательно, изменения, выполненные в параметре подпрограммы, не влияют на аргумент. Второй способ передачи аргумента — *вызов по ссылке*. При использовании этого подхода параметру передается ссылка на аргумент (а не его значение). Внутри подпрограммы эта ссылка используется для обращения к реальному аргументу, указанному в вызове. Это означает, что изменения, выполненные в параметре, будут влиять на аргумент, использованный в вызове подпрограммы. Как вы убедитесь, в Java применяются оба подхода, в зависимости от передаваемых данных.

В Java элементарный тип передается методу по значению. Таким образом, все происходящее с параметром, который принимает аргумент, не оказывает влияния вне метода. Например, рассмотрим следующую программу:

```
// Элементарные типы 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 перед вызовом: " +
                       a + " " + b);
    ob.meth(a, b);
    System.out.println("a и b после вызова: " +
                       a + " " + b);
  }
}
```

Вывод этой программы имеет следующий вид:

```
a и b перед вызовом: 15 20
a и b после вызова: 15 20
```

Как видите, выполняемые внутри метода `meth()` операции не влияют на значения `a` и `b`, использованные в вызове. Их значения не изменились на 30 и 10.

При передаче объекта методу ситуация изменяется коренным образом, поскольку по существу объекты передаются посредством вызова по ссылке. Следует помнить, что при создании переменной типа класса создается лишь ссылка на объект. Таким образом, при передаче этой ссылки методу, принимающий ее параметр будет ссылаться на тот же объект, на который ссылается аргумент. По сути это означает, что объекты передаются мето-

дам посредством вызова по ссылке. Изменения объекта внутри метода *влияют* на объект, использованный в качестве аргумента. Например, рассмотрим такую программу:

```
// Объекты передаются по ссылке.
class Test {
  int a, b;
  Test(int i, int j) {
    a = i;
    b = j;
  }
  // передача объекта
  void meth(Test o) {
    o.a *= 2;
    o.b /= 2;
  }
}
class CallByRef {
  public static void main(String args[]) {
    Test ob = new Test(15, 20);
    System.out.println("ob.a и ob.b перед вызовом: " +
                       ob.a + " " + ob.b);
    ob.meth(ob);
    System.out.println("ob.a и ob.b после вызова: " +
                       ob.a + " " + ob.b);
  }
}
```

Эта программа генерирует следующий вывод:

```
ob.a и ob.b перед вызовом: 15 20
ob.a и ob.b после вызова: 30 10
```

Как видите, в данном случае действия внутри метода `meth()` влияют на объект, использованный в качестве аргумента.

Интересно отметить, что когда ссылка на объект передается методу, сама ссылка передается посредством вызова по значению. Однако поскольку передаваемое значение ссылается на объект, копия этого значения все равно будет ссылаться на тот же объект, что и соответствующий аргумент.

Помните! *Когда элементарный тип передается методу, это выполняется посредством вызова по значению. Объекты передаются неявно с помощью вызова по ссылке.*

Возврат объектов

Метод может возвращать любой тип данных, в том числе созданные типы классов. Например, в следующей программе метод `icrByTen()` возвращает объект, в котором значение переменной `a` на 10 больше значения этой переменной в вызывающем объекте.

```
// Возвращение 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 генерирует следующий вывод:

```
ob1.a: 2
ob2.a: 12
ob2.a после второго увеличения значения: 22
```

Как видите, при каждом вызове метода `incrByTen()` программа создает новый объект и возвращает ссылку на него вызывающей процедуре.

Приведенная программа иллюстрирует еще один важный момент: поскольку все объекты распределяются динамически с помощью операции `new`, программисту не нужно беспокоиться о том, чтобы объект не вышел за пределы области определения, т.к. выполнение метода, в котором он был создан, прекращается. Объект будет существовать до тех пор, пока где-либо в программе будет существовать ссылка на него. При отсутствии какой-либо ссылки на него объект будет уничтожен во время следующей уборки мусора.

Рекурсия

В Java поддерживается *рекурсия*. Рекурсия — это процесс определения чего-либо в терминах самого себя. Применительно к программированию на Java рекурсия — это атрибут, который позволяет методу вызывать самого себя. Такой метод называют *рекурсивным*.

Классический пример рекурсии — вычисление факториала числа. Факториал числа *N* — это произведение всех целых чисел от 1 до *N*. Например, факториал 3 равен 1×2×3, или 6. Вот как можно 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));
   }
}
```

Вывод этой программы имеет вид:

```
Факториал 3 равен 6
Факториал 4 равен 24
Факториал 5 равен 120
```

Для тех, кто не знаком с рекурсивными методами, работа метода `fact()` может быть не совсем понятна. Вот как работает этот метод. При вызове метода `fact()` с аргументом, равным 1, функция возвращает 1. В противном случае она возвращает произведение `fact(n-1)*n`. Для вычисления этого выражения программа вызывает метод `fact()` с аргументом 2. Это приведет к третьему вызову метода с аргументом, равным 1. Затем этот вызов возвратит значение 1, которое будет умножено на 2 (значение `n` во втором вызове метода). Этот результат (равный 2) возвращается исходному вызову метода `fact()` и умножается на 3 (исходное значение `n`). В результате мы получаем ответ, равный 6. В метод `fact()` можно было бы вставить операторы `println()`, которые будут отображать уровень каждого вызова и промежуточные результаты.

Когда метод вызывает самого себя, новым локальным переменным и параметрам выделяется место в стеке и код метода выполняется с этими новыми начальными значениями. При каждом возврате из рекурсивного вызова старые локальные переменные и параметры удаляются из стека, и выполнение продолжается с момента вызова внутри метода. Рекурсивные методы выполняют действия, подобные выдвиганию и складыванию телескопа.

Из-за дополнительной перегрузки ресурсов, связанной с дополнительными вызовами функций, рекурсивные версии многих подпрограмм могут выполняться несколько медленнее их итерационных аналогов. Большое количество обращений к методу могут вызвать переполнение стека. Поскольку параметры и локальные переменные сохраняются в стеке, а каждый новый вызов создает новые копии этих значений, это может привести к переполнению стека. В этом случае система времени выполнения Java будет генерировать исключение. Однако, вероятно, об этом можно не беспокоиться, если только рекурсивная подпрограмма не начинает себя вести странным образом.

Основное преимущество применения рекурсивных методов состоит в том, что их можно использовать для создания более понятных и простых версий некоторых алгоритмов, чем при использовании итерационных аналогов. Например, алгоритм быстрой сортировки достаточно трудно реализовать итерационным методом. А некоторые типы алгоритмов, связанных с искусственным интеллектом, легче всего реализовать именно с помощью рекурсивных решений.

При использовании рекурсивных методов нужно позаботиться о том, чтобы где-либо в программе присутствовал оператор `if`, осуществляющий возврат из рекурсивного метода без выполнения рекурсивного вызова. В противном случае, будучи вызванным, метод никогда не выполнит возврат. Эта ошибка очень часто встречается при работе с рекурсией. Поэтому во время разработки советуем как можно чаще использовать операторы `println()`, чтобы можно было следить за происходящим и прервать выполнение в случае ошибки.

Рассмотрим еще один пример рекурсии. Рекурсивный метод `printArray()` выводит первые `i` элементов массива `values`.

```
// Еще один пример рекурсии.
class RecTest {
  int values[];
  RecTest(int i) {
    values = new int[i];
  }
  // рекурсивное отображение элементов 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 генерирует следующий вывод:

```
[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 вы уже знаете, инкапсуляция связывает данные с манипулирующим ими кодом. Однако инкапсуляция предоставляет еще один важный атрибут: *управление доступом*. Посредством инкапсуляции можно управлять тем, какие части программы могут получать доступ к членам класса. Управление доступом позволяет предотвращать злоупотребления. Например, предоставляя доступ к данным только посредством четко определенного набора методов, можно предотвратить злоупотребление этими данными. Таким образом, если класс реализован правильно, он создает “черный ящик”, который можно использовать, но внутренний механизм которого защищен от повреждения. Однако представленные ранее классы не полностью соответствуют этой цели. Например, рассмотрим класс `Stack`, представленный в конце главы 6. Хотя методы `push()` и `pop()` действительно предоставляют управляемый интерфейс стека, этот интерфейс не обязателен для использования. То есть другая часть программы может обойти эти методы и обратиться к стеку непосредственно. Понятно, что в “плохих руках” эта возможность может приво-

дить к проблемам. В этом разделе мы представим механизм, с помощью которого можно строго управлять доступом к различным членам класса.

Способ доступа к члену класса определяется *спецификатором доступа*, который изменяет его объявление. В Java определен обширный набор спецификаторов доступа. Некоторые аспекты управления доступом связаны главным образом с наследованием и пакетами. (*Пакет* — это, по сути, группирование классов.) Эти составляющие механизма управления доступом Java будут рассмотрены в последующих разделах. А пока начнем с рассмотрения применения управления доступа к отдельному классу. Когда основы управления доступом станут понятными, освоение других аспектов не представит особой сложности.

Спецификаторами доступа Java являются `public` (общедоступный), `private` (приватный) и `protected` (защищенный). Java определяет также уровень доступа, предоставляемый по умолчанию. Спецификатор `protected` применяется только при использовании наследования. Остальные спецификаторы доступа описаны далее в этой главе.

Начнем с определения спецификаторов `public` и `private`. Когда член класса изменяется спецификатором доступа `public`, он становится доступным для любого другого кода. Когда член класса указан как `private`, он доступен только другим членам этого же класса. Теперь вам должно быть понятно, почему методу `main()` всегда предшествует спецификатор `public`. Этот метод вызывается кодом, расположенным вне данной программы — т.е. системой времени выполнения Java. При отсутствии спецификатора доступа по умолчанию член класса считается общедоступным внутри своего собственного пакета, но недоступным для кода, расположенного вне этого пакета. (Пакеты рассматриваются в следующей главе.)

В уже разработанных нами классах все члены класса использовали режим доступа, определенный по умолчанию, который, по сути, является общедоступным. Однако, как правило, это не будет соответствовать реальным требованиям. Обычно будет требоваться ограничить доступ к членам данных класса — предлагая доступ только через методы. Кроме того, в ряде случаев придется определять приватные методы класса.

Спецификатор доступа предшествует остальной спецификации типа члена. То есть оператор объявления члена должен начинаться со спецификатора доступа. Например:

```
public int i;
private double j;
private int myMethod(int a, char b) { // ...
```

Чтобы влияние использования общедоступного и приватного доступа было понятно, рассмотрим следующую программу:

```
/* Эта программа демонстрирует различие между спецификаторами
   public и private.
*/
class Test {
  int a;            // доступ, определенный по умолчанию
  public int b;     // общедоступный доступ
  private int c;    // приватный доступ
  // методы доступа к c
  void setc(int i) {// установка значения переменной c
    c = i;
  }
  int getc() {      // получение значения переменной c
    return c;
  }
}
```

```
class AccessTest {
  public static void main(String args[]) {
    Test ob = new Test();
    // Эти операторы правильны, a и b доступны непосредственно
    ob.a = 10;
    ob.b = 20;
    // Этот оператор неверен и может вызвать ошибку
// ob.c = 100; // Ошибка!
    // Доступ к объекту c должен осуществляться посредством методов его класса
    ob.setc(100); // OK
    System.out.println("a, b, и c: " + ob.a + " " + ob.b + " " + ob.getc());
  }
}
```

Как видите, внутри класса `Test` использован метод доступа, заданный по умолчанию, что в данном примере равносильно указанию доступа `public`. Объект `b` явно указан как `public`. Объект `c` указан как приватный. Это означает, что он недоступен для кода, переделенного вне его класса. Поэтому внутри класса `AccessTest` объект `c` не может применяться непосредственно. Доступ к нему должен осуществляться посредством его общедоступных методов `setc()` и `getc()`. Удаление символа комментария из начала строки:

```
// ob.c = 100; // Ошибка!
```

сделало бы компиляцию этой программы невозможной из-за нарушений правил доступа.

В качестве более реального примера применения управления доступа рассмотрим следующую усовершенствованную версию класса `Stack`, код которого был приведен в конце 6 главы:

```
// Этот класс определяет целочисленный стек, который может содержать 10 значений.
class Stack {
  /* Теперь stck и tos являются приватными. Это означает,
     что они не могут быть случайно или намеренно
     изменены так, чтобы повредить стек.
  */
  private int stck[] = new int[10];
  private int tos;
  // Инициализация верхушки стека
  Stack() {
    tos = -1;
  }
  // Проталкивание элемента в стек
  void push(int item) {
    if(tos==9)
      System.out.println("Стек полон.");
    else
      stck[++tos] = item;
  }
  // Выталкивание элемента из стека
  int pop() {
    if(tos < 0) {
      System.out.println("Стек не загружен.");
      return 0;
    }
    else
      return stck[tos--];
  }
}
```

Как видите, теперь обе переменные: и stck, содержащая стек, и tos, содержащая индекс верхушки стека, указаны как private. Это означает, что обращение к ним или их изменение могут осуществляться только через методы push() и pop(). Например, указание переменной tos как приватной препятствует случайной установке другими частями программы ее значения выходящим за пределы конца массива stck.

Следующая программа — усовершенствованная версия класса Stack. Чтобы убедиться в том, что члены класса stck и tos действительно недоступны, попытайтесь удалить символы комментария из строк операторов.

```
class TestStack {
  public static void main(String args[]) {
    Stack mystack1 = new Stack();
    Stack mystack2 = new Stack();

    // проталкивание чисел в стек
    for(int i=0; i<10; i++) mystack1.push(i);
    for(int i=10; i<20; i++) mystack2.push(i);

    // выталкивание этих чисел 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;
  }
}
```

Хотя обычно методы будут обеспечивать доступ к данным, которые определены классом, это не обязательно. Переменная экземпляра вполне может быть общедоступной, если на то имеются веские причины. Например, для простоты переменные экземпляров в большинстве простых классов, созданных в этой книге, определены как общедоступные. Однако в большинстве классов, применяемых в реальных программах, манипулирование данными должно будет выполняться только посредством методов. В следующей главе мы вернемся к теме управления доступом. Вы убедитесь, что управление доступом особенно важно при использовании наследования.

Что такое static

В некоторых случаях желательно определить член класса, который будет использоваться независимо от любого объекта этого класса. Обычно обращение к члену класса должно выполняться только в сочетании с объектом его класса. Однако можно создать член класса, который может использоваться самостоятельно, без ссылки на конкретный экземпляр. Чтобы создать такой член, в начало его объявления нужно поместить ключевое слово static. Когда член класса объявлен как static (статический), он доступен до создания каких-либо объектов его класса и без ссылки на какой-либо объект. Статическими могут быть объявлены как методы, так и переменные. Наиболее распространенный пример статического члена — метод main(). Этот метод объявляют как static, поскольку он должен быть объявлен до создания любых объектов.

Переменные экземпляров, объявленные как `static`, по существу являются глобальными переменными. При объявлении объектов их класса программа не создает никаких копий переменной `static`. Вместо этого все экземпляры класса совместно используют одну и ту же статическую переменную.

На методы, объявленные как `static`, накладывается ряд ограничений.

- Они могут вызывать только другие статические методы.
- Они должны осуществлять доступ только к статическим переменным.
- Они ни коим образом не могут ссылаться на члены типа `this` или `super`. (Ключевое слово `super` связано с наследованием и описывается в следующей главе.)

Если для инициализации переменных типа `static` нужно выполнить вычисления, можно объявить статический блок, который будет выполняться только один раз при первой загрузке класса. В следующем примере показан класс, который содержит статический метод, несколько статических переменных и статический блок инициализации:

```
// Демонстрация статических переменных, методов и блоков.
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);
   }
}
```

Сразу после загрузки класса `UseStatic` программа выполняет все операторы `static`. Вначале значение `a` устанавливается равным `3`, затем программа выполняет блок `static`, который выводит сообщение, а затем инициализирует переменную `b` значением `a*4`, или `12`. Затем программа вызывает метод `main()`, который обращается к методу `meth()`, передавая параметру `x` значение `42`. Три оператора `println()` ссылаются на две статических переменные `a` и `b` на локальную переменную `x`.

Вывод этой программы имеет такой вид:

```
Статический блок инициализирован.
x = 42
a = 3
b = 12
```

За пределами класса, в котором они определены, статические методы и переменные могут использоваться независимо от какого-либо объекта. Для этого достаточно указать имя их класса, за которым должна следовать операция точки. Например, если метод типа `static` нужно вызвать извне его класса, это можно выполнить, используя следующую общую форму:

```
имя_класса.метод( )
```

Здесь *имя_класса* — имя класса, в котором объявлен метод тип `static`. Как видите, этот формат аналогичен применяемому для вызова нестатических методов через переменные объектных ссылок. Статическая переменная доступна аналогичным образом — посредством операции точки, следующей за именем класса. Так в Java реализованы управляемые версии глобальных методов и переменных.

Приведем пример. Внутри метода `main()` обращение к статическому методу `callme()` и статической переменной `b` осуществляется посредством имени их класса `StaticDemo`.

```
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);
   }
}
```

Вывод этой программы выглядит следующим образом:

```
a = 42
b = 99
```

Знакомство с ключевым словом `final`

Переменная может быть объявлена как `final` (окончательная). Это позволяет предотвратить изменение содержимого переменной. Это означает, что переменная типа `final` должна быть инициализирована во время ее объявления. Например:

```
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 все последующие части программы могут пользоваться переменной `FILE_OPEN` и прочими так, как если бы они были константами, без риска изменения их значений.

В практике программирования на Java принято идентификаторы всех переменных типа `final` записывать прописными буквами. Переменные, объявленные как `final`, не занимают отдельную область памяти для каждого экземпляра — т.е., по сути, они являются константами.

Ключевое слово `final` можно применять также к методам, но в этом случае его значение существенно отличается от применяемого к переменным. Это второе применение ключевого слова `final` описано в следующей главе, посвященной наследованию.

Повторное рассмотрение массивов

Массивы были представлены ранее в этой книге до того, как мы рассмотрели классы. Теперь, имея представление о классах, можно сделать важный вывод относительно мас-

сивов: все они реализованы как объекты. В связи с этим существует специальный атрибут массива, который наверняка пригодится. 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 свойство.

```
// Эта программа демонстрирует член 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 генерирует следующий вывод:

```
длина a1 равна 10
длина a2 равна 8
длина a3 равна 4
```

Как видите, программа отображает размер каждого массива. Имейте в виду, что значение переменной `length` никак не связано с количеством действительно используемых элементов. Оно отражает лишь то количество элементов, которое может содержать массив.

Член `length` может находить применение во множестве ситуаций. Например, ниже показана усовершенствованная версия класса `Stack`. Как вы, возможно, помните, предшествующие версии этого класса всегда создавали десятиэлементный стек. Следующая версия позволяет создавать стеки любого размера. Значение `stck.length` служит для предотвращения переполнения стека.

```
// Усовершенствованный класс Stack, в котором использован член длины массива.
class Stack {
  private int stck[];
  private int tos;
  // распределение и инициализация стека
  Stack(int size) {
    stck = new int[size];
    tos = -1;
  }
  // Проталкивание элемента в стек
  void push(int item) {
    if(tos==stck.length-1) // использование члена длины массива
      System.out.println("Стек полон.");
    else
      stck[++tos] = item;
  }
  // Выталкивание элемента 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);
      // проталкивание чисел в стек
      for(int i=0; i<5; i++) mystack1.push(i);
      for(int i=0; i<8; i++) mystack2.push(i);
      // выталкивание этих чисел 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, что программа создает два стека: один глубиной в пять элементов, а второй — в шесть. Как видите, то, что массивы поддерживают информацию о своей длине, упрощает создание стеков любого размера.

Представление вложенных и внутренних классов

Java позволяет определять класс внутри другого класса. Такие классы называют *вложенными классами*. Область определения вложенного класса ограничена областью определения внешнего класса. Таким образом, если класс B определен внутри класса A, класс B не может существовать независимо от класса A. Вложенный класс имеет доступ к членам, в том числе приватным, класса, в который он вложен. Однако внешний класс не имеет доступ к членам вложенного класса. Вложенный класс, который объявлен непосредственно внутри области определения своего внешнего класса, является его членом. Можно также объявлять вложенные классы, являющиеся локальными для блока.

Существует два типа вложенных классов: *статические* и *нестатические*. Статический вложенный класс — класс, к которому применен модификатор `static`. Поскольку он является статическим, он должен обращаться к своему внешнему классу посредством объекта. То есть он не может непосредственно ссылаться на члены своего внешнего класса. Из-за этого ограничения статические вложенные классы используются редко.

Наиболее важный тип вложенного класса — *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 так же, как это делают остальные нестатические члены внешнего класса.

Следующая программа иллюстрирует определение и использование внутреннего класса. Класс `Outer` содержит одну переменную экземпляра `outer_x`, один метод экземпляра `test()` и определяет один внутренний класс `Inner`.

```
// Демонстрация использования 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();
  }
}
```

Это приложение генерирует следующий вывод:

```
вывод: outer_x = 100
```

В этой программе внутренний класс `Inner` определен в области определения класса `Outer`. Поэтому любой кода в классе `Inner` может непосредственно обращаться к переменной `outer_x`. Метод экземпляра `display()` определен внутри класса `Inner`. Этот метод отображает значение переменной `outer_x` в стандартном выходном потоке. Метод `main()` экземпляра `InnerClassDemo` создает экземпляр класса `Outer` и вызывает его метод `test()`. Этот метод создает экземпляр класса `Inner` и вызывает метод `display()`.

Важно понимать, что экземпляр класса `Inner` может быть создан только внутри области определения класса `Outer`. Компилятор Java генерирует сообщение об ошибке, если любой код вне класса `Outer` пытается инициализировать класс `Inner`. (В общем случае экземпляр внутреннего класса должен создаваться содержащим его диапазоном.) Однако экземпляр класса `Inner` можно создать снаружи класса `Outer`, уточняя имя внутреннего класса именем внешнего класса — например `Outer.Inner`.

Как уже было сказано, внутренний класс имеет доступ ко всем элементам своего внешнего класса, но не наоборот. Члены внутреннего класса известны только внутри области определения внутреннего класса и не могут быть использованы внешним классом. Например:

```
// Компиляция этой программы будет не возможна.
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 не известна!
  }
}
class InnerClassDemo {
  public static void main(String args[]) {
    Outer outer = new Outer();
    outer.test();
  }
}
```

В этом примере переменная `y` объявлена как переменная экземпляра класса `Inner`. Поэтому она не известна за пределами класса и не может использоваться методом `showy()`.

Хотя мы уделили основное внимание внутренним классам, определенным в качестве членов внутри области определения внешнего класса, внутренние классы можно определять внутри области определения любого блока. Например, вложенный класс можно определить внутри блока, определенного методом, или даже внутри тела цикла `for`, как показано в следующем примере:

```
// Определение внутреннего класса внутри цикла 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();
  }
}
```

Вывод, генерируемый этой версией программы, показан ниже.

```
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
```

Хотя вложенные классы применимы не во всех ситуациях, они особенно удобны при обработке событий. Мы вернемся к теме вложенных классов в главе 22. В ней представлены внутренние классы, которые можно использовать для упрощения кода, предназначенного для обработки определенных типов событий. Читатели ознакомятся также с *анонимными внутренними классами*, являющимися внутренними классами без имен.

И последнее: первоначальная спецификация Java версии 1.0 не допускала использования вложенных классов. Они появились в версии Java 1.1.

Описание класса `String`

Хотя класс `String` подробно будет рассмотрен во второй части этой книги, здесь уместно кратко ознакомить с ним читателей, поскольку мы будем использовать строки в некоторых последующих примерах первой части. Вероятно, `String` — наиболее часто используемый класс из библиотеки классов Java. Очевидная причина этого в том, что строки — исключительно важный элемент программирования.

Прежде всего, следует уяснить, что любая создаваемая строка в действительности представляет собой объект типа `String`. Даже строковые константы в действительности являются объектами `String`. Например, в операторе:

```
System.out.println("Это — также объект String");
```

строка `"Это — также объект String"` — константа типа `String`.

Во-вторых, объекты типа `String` являются неизменными. После того как объект типа `String` создан, его содержимое не может изменяться. Хотя это может казаться серьезным ограничением, на самом деле это не так по двум причинам.

- Если нужно изменить строку, всегда можно создать новую строку, содержащую все изменения.
- В Java определен равноправный класс `String`, `StringBuffer`, допускающий изменение строк, что позволяет выполнять в Java все обычные манипуляции со строками. (Класс `StringBuffer` описан во второй части этой книги.)

Существует множество способов создания строк. Простейший из них — воспользоваться оператором вроде следующего:

```
String myString = "тестовая строка";
```

Как только объект `String` создан, его можно использовать во всех ситуациях, в которых допустимо использование строк. Например, следующий оператор отображает содержимое `myString`:

```
System.out.println(myString);
```

Для объектов типа `String` в Java определена одна операция: +. Она служит для объединения двух строк. Например, оператор:

```
String myString = "Мне" + " нравится " + "Java.";
```

Приводит к тому, что содержимым переменной `myString` становится строка `"Мне нравится Java"`.

Следующая программа иллюстрирует описанные концепции:

```
// Демонстрация применения 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 создает следующий вывод:

```
Первая строка
Вторая строка
Первая строка и вторая строка
```

Класс `String` содержит несколько методов, которые можно использовать. Опишем некоторые из них. С помощью метода `equals()` можно проверять равенство двух строк. Метод `length()` позволяет выяснить длину строки. Вызывая метод `charAt()`, можно получить символ с указанным индексом. Ниже приведены общие формы этих трех методов:

```
boolean equals(String объект)
int length( )
char charAt(int индекс)
```

Следующая программа демонстрирует применение этих методов:

```
// Демонстрация некоторых методов 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 strOb1: " +
                       strOb1.length());
    System.out.println("Символ с индексом 3 в 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 генерирует следующий вывод:

```
Длина strOb1: 12
Символ с индексом 3 в strOb1: s
strOb1 != strOb2
strOb1 == strOb3
```

Конечно, подобно тому, как могут существовать массивы любого другого типа объектов, могут существовать и массивы строк. Например:

```
// Демонстрация использования 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]);
  }
}
```

Вывод этой программы имеет вид:

```
str[0]: один
str[1]: два
str[2]: три
```

Как вы убедитесь из следующего раздела, строковые массивы играют важную роль во многих Java-программах.

Использование аргументов командной строки

Иногда будет требоваться передать определенную информацию программе во время ее запуска. Для этого используют *аргументы командной строки* метода `main()`. Аргумент командной строки — это информация, которую во время запуска программы задают в командной строке непосредственно после ее имени. Доступ к аргументам командной строки внутри Java-программы не представляет сложности — они хранятся в виде строк в массиве `String`, переданного методу `main()`. Первый аргумент командной строки хранится в элементе массива `args[0]`, второй — в элементе `args[0]` и т.д. Например, следующая программа отображает все аргументы командной строки, с которыми она вызывается.

```
// Отображение всех аргументов 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]);
  }
}
```

Попытайтесь выполнить эту программу, введя следующую строку:

```
java CommandLine this is a test 100 -1
```

В результате отобразится следующий вывод:

```
args[0]: this
args[1]: is
args[2]: a
args[3]: test
args[4]: 100
args[5]: -1
```

Помните! *Все аргументы командной строки передаются как строки. Численные значения нужно вручную преобразовать в их внутренние представления, как поясняется в главе 16.*

Varargs: аргументы переменной длины

В JDK 5 была добавлена новая функциональная возможность, которая упрощает создание методов, принимающих переменное количество аргументов. Эта функциональная возможность получила название *varargs* (сокращение термина *variable-length arguments* — аргументы переменной длины). Метод, который принимает переменное число аргументов, называют *методом переменной арности*, или просто *методом varargs*.

Ситуации, в которых методу нужно передавать переменное количество аргументов, встречаются не так уж редко. Например, метод, который открывает подключение к Internet, может принимать имя пользователя, пароль, имя файла, протокол и тому подобное, но применять значения, заданные по умолчанию, если какие-либо из этих сведений опущены. В этой ситуации было бы удобно передавать только те аргументы, для которых заданные по умолчанию значения не применимы. Еще один пример — метод `printf()`, входящий в состав библиотеки ввода-вывода Java. Как будет показано в главе 19, он принимает переменное число аргументов, которые форматирует, а затем выводит.

До появления 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 перегруженные версии метода — по одной для каждого возможного способа вызова метода. Хотя этот способ подходит для ряда случаев, он применим только к узкому классу ситуаций.

В тех случаях, когда максимальное число возможных аргументов было большим или неизвестным, применялся второй подход, при котором аргументы помещались в массив, а затем массив передавался методу. Следующая программа иллюстрирует этот подход.

```
// Использование массива для передачи методу переменного
// количества аргументов. Это старый стиль подхода
// 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 способ 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 создает следующий вывод:

```
Количество аргументов: 1 Содержимое: 10
Количество аргументов: 3 Содержимое: 1 2 3
Количество аргументов: 0 Содержимое:
```

В программе методу `vaTest()` аргументы передаются через массив `v`. Этот старый подход к обработке аргументов переменной длины позволяет методу `vaTest()` принимать любое число аргументов. Однако он требует, чтобы эти аргументы были вручную помещены в массив до вызова метода `vaTest()`. Создание массива при каждом вызове метода `vaTest()` не только трудоемкая, но и чреватая ошибками задача. Функциональная возможность использования методов varargs обеспечивает более простой и эффективный подход.

Для указания аргумента переменной длины используют три точки (`...`). Например, вот как метод `vaTest()` можно записать с использованием аргумента переменной длины:

```
static void vaTest(int ... v) {
```

Эта синтаксическая конструкция указывает компилятору, что метод `vaTest()` может вызываться с нулем или более аргументов. В результате `v` неявно объявляется как массив типа `int[]`. Таким образом, внутри метода `vaTest()` доступ к `v` осуществляется с использованием синтаксиса обычного массива. Предыдущая программа с применением метода vararg приобретает следующий вид:

```
// Демонстрация использования аргументов переменной длины.
class VarArgs {
  // теперь vaTest() использует аргументы 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
    // vaTest() с переменным числом аргументов.
    vaTest(10);          // 1 аргумент
    vaTest(1, 2, 3);     // 3 аргумента
    vaTest();            // без аргументов
  }
}
```

Вывод этой программы совпадает с выводом исходной версии.

Отметим две важные особенности этой программы. Во-первых, как уже было сказано, внутри метода `vaTest()` переменная `v` действует как массив. Это обусловлено тем, что `v` *является* массивом. Синтаксическая конструкция `...` просто указывает компилятору, что метод будет использовать переменное количество аргументов, и что эти аргументы будут храниться в массиве, на который ссылается переменная `v`. Во-вторых, в методе `main()` метод `vaTest()` вызывается с различным количеством аргументов, в том числе, и вовсе без аргументов. Аргументы автоматически помещаются в массив и передаются переменной `v`. В случае отсутствия аргументов длина массива равна нулю.

Наряду с параметром переменной длины массив может содержать “нормальные” параметры. Однако параметр переменной длины должен быть последним параметром, объявленным методом. Например, следующее объявление метода вполне допустимо:

```
int doIt(int a, int b, double c, int ... vals) {
```

В данном случае первые три аргумента, указанные в обращении к методу `doIt()`, соответствуют первым трем параметрам. Все остальные аргументы считаются принадлежащими параметру `vals`.

Помните, что параметр vararg должен быть последним. Например, следующее объявление записано неправильно:

```
int doIt(int a, int b, double c, int ... vals, boolean stopFlag) { // Ошибка!
```

В этом примере предпринимается попытка объявления обычного параметра после параметра типа vararg, что недопустимо.

Существует еще одно ограничение, о котором следует знать: метод должен содержать только одни параметр типа varargs. Например, следующее объявление также неверно:

```
int doIt(int a, int b, double c, int ... vals, double ... morevals) { // Ошибка!
```

Попытка объявления второго параметра типа vararg недопустима.

Рассмотрим измененную версию метода `vaTest()`, которая принимает обычный аргумент и аргумент переменной длины:

```
// Использование аргумента переменой длины совместно со стандартными
аргументами.
class VarArgs2 {
  // В этом примере msg — обычный параметр,
  // а v — параметр vararg.
  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 vararg: ", 10);
    vaTest("Три параметра vararg: ", 1, 2, 3);
    vaTest("Без параметров vararg: ");
  }
}
```

Вывод это программы имеет вид:

```
Один параметр vararg: 1 Содержимое: 10
Три параметра vararg: 3 Содержимое: 1 2 3
Без параметров vararg: 0 Содержимое:
```

Перегрузка методов vararg

Метод, который принимает аргумент переменной длины, можно перегружать. Например:

```
// Параметры vararg 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 создает следующий вывод:

```
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
```

Приведенная программа иллюстрирует оба возможных способа перегрузки метода vararg. Во-первых, типы его параметра vararg могут быть различными. Именно это имеет место в вариантах `vaRest(int ...)` и `vaTest(boolean ...)`. Помните, что конструкция `...` вынуждает компилятор обрабатывать параметр как массив указанного типа. Поэтому, подобно тому, как можно выполнять перегрузку методов, используя различные типы параметров массива, можно выполнять перегрузку методов vararg, используя различные типы аргументов переменной длины. В этом случае система Java использует различие в типах для определения нужного варианта перегруженного метода.

Второй способ перегрузки метода vararg — добавление обычного параметра. Именно это было сделано для `vaTest(String, int ...)`. В данном случае для определения нужного метода система Java использует и количество аргументов, и их тип.

На заметку! *Метод, поддерживающий varargs, может быть перегружен также методом, который не поддерживает эту функциональную возможность. Например, в приведенной ранее программе метод `vaTest()` может быть перегружен методом `vaTest(int x)`. Эта специализированная версия вызывается только при наличии аргумента `int`. В случае передаче методу двух и более аргументов `int` программа будет использовать varargs-версию метода `vaTest(int...v)`.*

Параметры переменной длины и неопределенность

При перегрузке метода, принимающего аргумент переменной длины, могут случаться непредвиденные ошибки. Они связаны с неопределенностью, которая может возникать при вызове перегруженного метода с аргументом переменной длины. Например, рассмотрим следующую программу:

```
// Аргументы переменной длины, перегрузка и неопределенность.
//
// Эта программа содержит ошибку, и ее компиляция
// будет невозможна!
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);              // OK
    vaTest(true, false, false);   // OK
    vaTest();                     // Ошибка: неопределенность!
  }
}
```

В этой программе перегрузка метода `vaTest()` выполняется вполне корректно. Однако ее компиляция будет невозможна из-за следующего вызова:

```
vaTest(); // Ошибка: неопределенность!
```

Поскольку параметр типа vararg может быть пустым, этот вызов может быть преобразован в обращение к `vaTest(int ...)` или к `vaTest(boolean ...)`. Оба варианта допустимы. Поэтому вызов принципиально неоднозначен.

Рассмотрим еще один пример неопределенности. Следующие перегруженные версии метода `vaTest()` изначально неоднозначны, несмотря на то, что одна из них принимает обычный параметр:

```
static void vaTest(int ... v) { // ...
static void vaTest(int n, int ... v) { // ...
```

Хотя списки параметров метода `vaTest()` различны, компилятор не имеет возможности разрешения следующего вызова:

```
vaTest(1)
```

Должен ли он быть преобразован в обращение к `vaTest(int ...)` с одним аргументом переменной длины или в обращение к `vaTest(int, int ...)` без аргументов переменной длины? Компилятор не имеет возможности ответить на этот вопрос. Таким образом ситуация неоднозначна.

Из-за ошибок неопределенности, подобных описанным, в некоторых случаях придется пренебрегать перегрузкой и просто использовать два различных имени метода. Кроме того, в некоторых случаях ошибки неопределенности служат признаком концептуальных изъянов программы, которые можно устранить путем более тщательного построения решения задачи.

8 ГЛАВА

Наследование

Наследование — одно из фундаментальных понятий объектно-ориентированного программирования, поскольку оно позволяет создавать иерархические классификации. Используя наследование, можно создать общий класс, который определяет характеристики, общие для набора связанных элементов. Затем этот класс может наследоваться другими, более специализированными классами, каждый из которых будет добавлять свои уникальные характеристики. В терминологии Java наследуемый класс называют *суперклассом*. Наследующий класс носит название *подкласса*. Следовательно, подкласс — это специализированная версия суперкласса. Он наследует все переменные экземпляра и методы, определенные суперклассом, и добавляет собственные, уникальные элементы.

Основы наследования

Чтобы наследовать класс, достаточно просто вставить определение одного класса в другой с использованием ключевого слова `extends`. В качестве иллюстрации рассмотрим короткий пример. Следующая программа создает суперкласс `A` и подкласс `B`. Обратите внимание на использование ключевого слова `extends` для создания подкласса класса `A`.

```
// Простой пример наследования.
// Создание суперкласса.
class A {
   int i, j;
   void showij() {
      System.out.println("i и j: " + i + " " + j);
   }
}
// Создание подкласса путем расширения класса A.
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 superOb: ");
    superOb.showij();
    System.out.println();

    /* Подкласс имеет доступ ко всем общедоступным членам
       своего суперкласса. */
    subOb.i = 7;
    subOb.j = 8;
    subOb.k = 9;
    System.out.println("Содержимое subOb: ");
    subOb.showij();
    subOb.showk();
    System.out.println();
    System.out.println("Сумма i, j и k в subOb:");
    subOb.sum();
  }
}
```

Эта программа создает следующий вывод:

```
Содержимое superOb:
i и j: 10 20

Содержимое subOb:
i и j: 7 8
k: 9

Сумма i, j и k в subOb:
i+j+k: 24
```

Как видите, подкласс `B` включает в себя все члены своего суперкласса `A`. Именно поэтому `subOb` имеет доступ к переменным `i` и `j` и может вызывать метод `showij()`. Кроме того, внутри метода `sum()` возможна непосредственная ссылка на переменные `i` и `j`, как если бы они были частью класса `B`.

Несмотря на то что `A` — суперкласс класса `B`, он также является полностью независимым, самостоятельным классом. То, что класс является суперклассом подкласса, не означает невозможность его самостоятельного использования. Более того, подкласс может быть суперклассом другого подкласса.

Общая форма объявления класса, который наследует от суперкласса, следующая:

```
class имя_подкласса extends имя_суперкласса {
  // тело класса
}
```

Для каждого создаваемого подкласса можно указывать только один суперкласс. Java не поддерживает наследование нескольких суперклассов в одном подклассе. Как было сказано, можно создать иерархию наследования, в которой подкласс становится суперклассом другого подкласса. Однако никакой класс не может быть собственным суперклассом.

Доступ к членам и наследование

Хотя подкласс включает в себя все члены своего суперкласса, он не может получать доступ к тем членам суперкласса, которые объявлены как `private`. Например, рассмотрим следующую простую иерархию классов:

```
/* В иерархии классов приватные члены остаются приватными
   для своего класса.
   Эта программа содержит ошибку, и ее компиляция
   будет невозможна.
*/
// Создание суперкласса.
class A {
   int i;             // общедоступная по умолчанию
   private int j;     // приватная для A
   void setij(int x, int y) {
      i = x;
      j = y;
   }
}
// Переменная j класса A в этом классе недоступна.
class B extends A {
   int total;
   void sum() {
      total = i + j;  // ОШИБКА, j в этом классе недоступна
   }
}
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);
   }
}
```

Компиляция этой программы будет невозможна, поскольку ссылка на переменную `j` внутри метода `sum()` класса `B` приводит к нарушению правил доступа. Поскольку переменная `j` объявлена как `private`, она доступна только другим членам ее собственного класса. Подкласс не имеет к ней доступа.

Помните! *Член класса, который объявлен как приватный, останется приватным для своего класса. Он недоступен любому коду за пределами его класса, в том числе подклассам.*

Более реальный пример

Рассмотрим более реальный пример, который поможет проиллюстрировать возможности наследования. В нем мы расширим последнюю версию класса `Box`, разработанную в предыдущей главе, добавив в нее четвертый компонент, который назовем `weight` (вес). Таким образом, новый класс будет содержать ширину, высоту, глубину и вес параллелепипеда.

```
// В этой программе наследование используется для расширения класса Box.
class Box {
   double width;
   double height;
   double depth;
   // конструирование клона объекта
   Box(Box ob) { // передача объекта конструктору
      width = ob.width;
      height = ob.height;
      depth = ob.depth;
   }
   // конструктор, используемый при указании всех измерений
   Box(double w, double h, double d) {
      width = w;
      height = h;
      depth = d;
   }
   // конструктор, используемый, если ни одно из изменений не указано
   Box() {
      width = -1;     // значение -1 используется для указания
      height = -1;    // неинициализированного
      depth = -1;     // параллелепипеда
   }
   // конструктор, используемый при создании куба
   Box(double len) {
      width = height = depth = len;
   }
   // вычисление и возврат объема
   double volume() {
      return width * height * depth;
   }
}
   // Расширение класса Box включением в него 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 генерирует следующий вывод:

```
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`. Он может просто расширять `Box` в соответствии с конкретными целями.

Основное преимущество наследования состоит в том, что как только суперкласс, который определяет общие атрибуты набора объектов, создан, его можно использовать для создания любого числа более специализированных классов. Каждый подкласс может точно определять свою собственную классификацию. Например, следующий класс наследует характеристики класса `Box` и добавляет атрибут цвета.

```
// Этот код расширяет класс Box, включая в него атрибут 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;
   }
}
```

Помните, как только суперкласс, который определяет общие аспекты объекта, создан, он может наследоваться для создания специализированных классов. Каждый подкласс добавляет собственные уникальные атрибуты. В этом заключается сущность наследования.

Переменная суперкласса может ссылаться на объект подкласса

Ссылочной переменной суперкласса может быть присвоена ссылка на любой подкласс, производный от данного суперкласса. Этот аспект наследования будет весьма полезен во множестве ситуаций. 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();

      // присваивание объекту BoxWeight ссылки на ссылку объекта Box
      plainbox = weightbox;
      vol = plainbox.volume(); // OK, метод volume() определен в Box
      System.out.println("Объем plainbox равен " + vol);
```

```
      /* Следующий оператор ошибочен, поскольку plainbox
         не определяет член weight. */
// System.out.println("Вес plainbox равен " + plainbox.weight);
   }
}
```

В этом примере `weightbox` — ссылка на объекты `BoxWeight`, а `painbox` — ссылка на объекты `Box`. Поскольку `BoxWeight` — подкласс класса `Box`, ссылке `painbox` можно присваивать ссылку на объект `weightbox`.

Важно понимать, что доступные объекты определяются типом ссылочной переменной, а не типом объекта, на который она ссылается. То есть при присваивании ссылочной переменной суперкласса ссылки на объект подкласса доступ предоставляется только к указанным в ней частям объекта, определенного суперклассом. Именно поэтому объект `plainbox` не имеет доступа к переменной `weight` даже в том случае, когда он ссылается на объект `BoxWeight`. Если немного подумать, это становится понятным — суперклассу не известно, что именно подкласс добавляет в него. Поэтому последняя строка кода в предыдущем фрагменте оформлена в виде комментария. Ссылка объекта `Box` не имеет доступа к полю `weight`, поскольку оно не определено в классе `Box`.

Хотя все сказанное выше может казаться несколько запутанным, эти особенности находят ряд практических применений, два из которых рассматриваются в последующих разделах данной главы.

Использование ключевого слова `super`

В предшествующих примерах классы, производные от класса `Box`, были реализованы не столь эффективно и надежно, как могли бы. Например, конструктор `BoxWeight` явно инициализирует поля `width`, `height` и `depth` класса `Box`. Это не только ведет к дублированию кода суперкласса, что весьма неэффективно, но и предполагает наличие у подкласса доступа к этим членам. Однако в ряде случаев придется создавать суперкласс, подробности реализации которого доступны только для него самого (т.е. с приватными членами данных). В этом случае подкласс никак не сможет самостоятельно непосредственно обращаться или инициализировать эти переменные. Поскольку инкапсуляция — один из главных атрибутов ООП, не удивительно, что Java предлагает решение этой проблемы. Во всех случаях, когда подклассу нужно сослаться на его непосредственный суперкласс, это можно выполнить с помощью ключевого слова `super`.

Ключевое слово `super` имеет две общих формы. Первую используют для вызова конструктора суперкласса, а вторую — для обращения к члену суперкласса, скрытому членом подкласса. Рассмотрим обе формы.

Использование ключевого слова `super` для вызова конструкторов суперкласса

Подкласс может вызывать конструктор, определенный его суперклассом, с помощью следующей формы ключевого слова `super`:

```
super(список_аргументов);
```

`Список_аргументов` определяет любые аргументы, требуемые конструктору в суперклассе. Оператор `super()` всегда должен быть первым выполняемым внутри конструктора подкласса.

В качестве иллюстрации использования оператора `super()` рассмотрим следующую усовершенствованную версию класса `BoxWeight()`:

```
// Теперь класс BoxWeight использует ключевое слово super
// для инициализации своих атрибутов объекта Box.
class BoxWeight extends Box {
  double weight;    // вес параллелепипеда
  // инициализация переменных width, height и depth с помощью super()
  BoxWeight(double w, double h, double d, double m) {
    super(w, h, d); // вызов конструктора суперкласса
    weight = m;
  }
}
```

В этом примере метод `BoxWeight` вызывает `super()` с аргументами `w`, `h` и `d`. Это приводит к вызову конструктора `Box()`, который инициализирует `width`, `height` и `depth`, используя переданные ему значения этих параметров. Теперь класс `BoxWeight` не инициализирует эти значения самостоятельно. Ему нужно инициализировать только свое уникальное значение — `weight`. В результате при необходимости эти значения могут оставаться приватными значениями класса `Box`.

В приведенном примере метод `super()` был вызван с тремя аргументами. Поскольку конструкторы могут быть перегруженными, `super()` можно вызывать, используя любую форму, определенную суперклассом. Программа выполнит тот конструктор, который соответствует указанным аргументам. В качестве примера приведем полную реализацию класса `BoxWeight`, которая предоставляет конструкторы для различных способов конструирования параллелепипедов. В каждом случае конструктор `super()` вызывается с соответствующими аргументами. Обратите внимание, что внутри класса `Box` его члены `width`, `height` и `depth` объявлены как приватные.

```
// Полная реализация класса BoxWeight.
class Box {
  private double width;
  private double height;
  private double depth;

  // конструирование клона объекта
  Box(Box ob) { // передача объекта конструктору
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }

  // конструктор, используемый при указании всех измерений
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }

  // конструктор, используемый, если ни одно из измерений не указано
  Box() {
    width = -1;     // значение -1 используется для указания
    height = -1;    // неинициализированного
    depth = -1;     // параллелепипеда
  }
```

```
  // конструктор, используемый при создании куба
  Box(double len) {
    width = height = depth = len;
  }
  // вычисление и возврат объема
  double volume() {
    return width * height * depth;
  }
}
// Теперь BoxWeight полностью реализует все конструкторы.
class BoxWeight extends Box {
  double weight;              // вес параллелепипеда
  // конструирование клона объекта
  BoxWeight(BoxWeight ob) {   // передача объекта конструктору
    super(ob);
    weight = ob.weight;
  }
  // конструктор, используемый при указании всех параметров
  BoxWeight(double w, double h, double d, double m) {
    super(w, h, d);           // вызов конструктора суперкласса
    weight = m;
  }
  // конструктор, используемый по умолчанию
  BoxWeight() {
    super();
    weight = -1;
  }
  // конструктор, используемый 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 of 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 генерирует следующий вывод:

```
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 в классе `BoxWeight`:

```
// конструирование клона объекта
BoxWeight(BoxWeight ob) { // передача объекта конструктору
  super(ob);
  weight = ob.weight;
}
```

Обратите внимание, что конструктор `super()` выполняет передачу объекту типа `BoxWeight`, а не типа `Box`. Тем не менее, это все равно ведет к вызову конструктора `Box(Box.ob)`. Как уже было отмечено, переменную суперкласса можно использовать для ссылки на любой объект, унаследованный их этого класса. Таким образом, объект `BoxWeight` можно передать конструктору `Box`. Конечно, классу `Box` будут известны только его собственные члены.

Рассмотрим основные концепции применения конструктора `super()`. Когда подкласс вызывает конструктор `super()`, он вызывает конструктор своего непосредственного суперкласса. Таким образом, `super()` всегда ссылается на суперкласс, расположенный в иерархии непосредственно над вызывающим классом. Это положение справедливо даже в случае многоуровневой иерархии. Кроме того, оператор `super()` всегда должен быть первым оператором, выполняемым внутри конструктора подкласса.

Второе применение ключевого слова `super`

Вторая форма ключевого слова `super` действует подобно ключевому слову `this`, за исключением того, что она всегда ссылается на суперкласс подкласса, в котором она использована. Общая форма этого применения ключевого слова `super` имеет следующий вид:

```
super.член
```

Здесь *член* может быть методом либо переменной экземпляра.

Вторая форма применения ключевого слова super наиболее подходит в тех ситуациях, когда имена членов подкласса скрывают члены суперкласса с такими же именами. Рассмотрим следующую простую иерархию классов:

```
// Использование ключевого слова super для предотвращения скрытия имени.
class A {
  int i;
}
// Создание подкласса посредством расширения класса A.
class B extends A {
  int i;            // эта переменная i скрывает переменную i в классе A
  B(int a, int b) {
    super.i = a;    // i в классе A
    i = b;          // i в классе B
  }
  void show() {
    System.out.println("i в суперклассе: " + super.i);
    System.out.println("i в подклассе: " + i);
  }
}
class UseSuper {
  public static void main(String args[]) {
    B subOb = new B(1, 2);
    subOb.show();
  }
}
```

Эта программа отображает следующее:

```
i в суперклассе: 1
i в подклассе: 2
```

Хотя переменная экземпляра i в классе B скрывает переменную i в классе A, ключевое слово super позволяет получить доступ к переменной i, определенной в суперклассе. Как вы увидите, ключевое слово super можно использовать также для вызова методов, которые скрываются подклассом.

Создание многоуровневой иерархии

До сих пор мы использовали простые иерархии классов, которые состояли только из суперкласса и подкласса. Однако можно строить иерархии, которые содержат любое количество уровней наследования. Как уже отмечалось, вполне допустимо использовать подкласс в качестве суперкласса другого подкласса. Например, класс C может быть подклассом класса B, который, в свою очередь, является подклассом класса A. В подобных ситуациях каждый подкласс наследует все характеристики всех его суперклассов. В приведенном примере класс C наследует все характеристики классов B и A. В качестве примера многоуровневой иерархии рассмотрим следующую программу. В ней подкласс BoxWeight использован в качестве суперкласса для создания подкласса Shipment. Shipment наследует все характеристики классов BoxWeight и Box и добавляет поле cost, которое содержит стоимость поставки такого пакета.

```
// Расширение класса BoxWeight за счет включения в него стоимости доставки.
// Начнем с создания класса Box.
class Box {
  private double width;
  private double height;
  private double depth;
  // конструирование клона объекта
  Box(Box ob) { // передача объекта конструктору
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
  }
  // конструктор, используемый при указании всех измерений
  Box(double w, double h, double d) {
    width = w;
    height = h;
    depth = d;
  }
  // конструктор, используемый, когда ни одно из измерений не указано
  Box() {
    width = -1;     // значение -1 используется для указания
    height = -1;    // неинициализированного
    depth = -1;     // параллелепипеда
  }
  // конструктор, используемый при создании куба
  Box(double len) {
    width = height = depth = len;
  }
  // вычисление и возврат объема
  double volume() {
    return width * height * depth;
  }
}
// Добавление веса.
class BoxWeight extends Box {
  double weight;              // вес параллелепипеда
  // конструирование клона объекта
  BoxWeight(BoxWeight ob) {   // передача объекта конструктору
    super(ob);
    weight = ob.weight;
  }
  // конструктор, используемый при указании всех параметров
  BoxWeight(double w, double h, double d, double m) {
    super(w, h, d);           // вызов конструктора суперкласса
    weight = m;
  }
  // конструктор, используемый по умолчанию
  BoxWeight() {
    super();
    weight = -1;
  }
  // конструктор, используемый при создании куба
  BoxWeight(double len, double m) {
    super(len);
    weight = m;
  }
}
```

```
  // Добавление стоимости доставки.
  class Shipment extends BoxWeight {
    double cost;
  // конструирование клона объекта
  Shipment(Shipment ob) {     // передача объекта 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);        // вызов конструктора суперкласса
    cost = c;
  }
  // конструктор, используемый по умолчанию
  Shipment() {
    super();
    cost = -1;
  }
  // конструктор, используемый 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);
  }
}
```

Эта программа создает следующий вывод:

```
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, класс Shipment может использовать ранее определенные классы Box и BoxWeight, добавляя только ту дополнительную информацию, которая требуется для его собственного специализированного применения. В этом состоит одно из ценных свойств наследования. Оно позволяет повторно использовать код.

Приведенный пример иллюстрирует важный аспект: конструктор super() всегда ссылается на конструктор ближайшего в суперкласса в иерархии. Конструктор super() в классе Shipment вызывает конструктор класса BoxWeight. Конструктор super() в классе BoxWeight вызывает конструктор класса Box. Если в иерархии классов конструктор суперкласса требует передачи ему параметров, все подклассы должны передавать эти параметры "по эстафете". Данное утверждение справедливо независимо от того, нуждается ли подкласс в собственных параметрах.

На заметку! *В приведенном 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 помещены в отдельные файлы и компилироваться независимо друг от друга. Фактически, использование отдельных файлов — норма, а не исключение при создании иерархий классов.*

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 заканчивая подклассом. Более того, поскольку super() должен быть первым оператором, выполняемым в конструкторе подкласса, этот порядок остается неизменным, независимо от того, используется ли форма super(). Если конструктор super() не применяется, программа использует конструктор каждого суперкласса, заданный по умолчанию или не содержащий параметров. В следующей программе демонстрируется порядок выполнения конструкторов.

```
// Демонстрация порядка вызова конструкторов.
// Создание суперкласса.
class A {
  A() {
    System.out.println("Внутри конструктора A.");
  }
}

// Создание подкласса посредством расширения класса A.
class B extends A {
  B() {
    System.out.println("Внутри конструктора B.");
  }
}

// Создание еще одного подкласса посредством расширения класса B.
class C extends B {
  C() {
    System.out.println("Внутри конструктора C.");
  }
}
```

```
class CallingCons {
  public static void main(String args[]) {
    C c = new C();
  }
}
```

Эта программа генерирует следующий вывод:

```
Внутри конструктора A
Внутри конструктора B
Внутри конструктора C
```

Как видите, конструкторы вызываются в порядке наследования.

Если немного подумать, становится ясно, что выполнение конструкторов в порядке наследования имеет смысл. Поскольку суперкласс ничего не знает о своих подклассах, любая инициализация, которую он должен выполнить, полностью независима и, возможно, обязательна для выполнения любой инициализации, выполняемой подклассом. Поэтому она должна выполняться первой.

Переопределение методов

Если в иерархии классов имя и сигнатура типа метода подкласса совпадает с атрибутами метода суперкласса, говорят, что метод подкласса *переопределяет* метод суперкласса. Когда переопределенный метод вызывается из подкласса, он всегда будет ссылаться на версию этого метода, определенную подклассом. Версия метода, определенная суперклассом, будет сокрыта. Рассмотрим следующий пример:

```
// Переопределение метода.
class A {
  int i, j;
  A(int a, int b) {
    i = a;
    j = b;
  }
  // отображение 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;
  }
  // отображение k — этот метод переопределяет метод show() 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(); // этот оператор вызывает метод show() класса B
  }
}
```

Эта программа создает следующий вывод:

```
k: 3
```

Когда программа вызывает метод show() по отношению к объекту типа B, она использует версию этого метода, определенную внутри класса B. То есть версия метода show(), определенная внутри класса B, переопределяет версию, объявленную внутри класса A.

Если нужно получить доступ к версии переопределенного метода, определенного в суперклассе, это можно сделать с помощью ключевого слова super. Например, в следующей версии класса B версия метода show(), объявленная в суперклассе, вызывается внутри версии подкласса. Это позволяет отобразить все переменные экземпляров.

```
class B extends A {
   int k;
   B(int a, int b, int c) {
      super(a, b);
      k = c;
   }
   void show() {
      super.show(); // этот оператор вызывает метод show() класса A
      System.out.println("k: " + k);
   }
}
```

Подстановка этой версии класса A в предыдущую программу приведет к следующему выводу:

```
i и j: 1 2
k: 3
```

В этой версии super.show() вызывает версию метода show(), определенную в суперклассе.

Переопределение метода выполняется *только* в том случае, если имена и сигнатуры типов двух методов идентичны. В противном случае два метода являются просто перегруженными. Например, рассмотрим измененную версию предыдущего примера.

```
// Методы с различающимися сигнатурами являются
// перегруженными, а не переопределенными.
class A {
   int i, j;
   A(int a, int b) {
      i = a;
      j = b;
   }
   // отображение i и j
   void show() {
      System.out.println("i и j: " + i + " " + j);
   }
}

// Создание подкласса посредством расширения класса A.
class B extends A {
   int k;
   B(int a, int b, int c) {
      super(a, b);
      k = c;
   }
```

```
    // перегрузка метода 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: ");   // вызов метода show() класса B
      subOb.show();            // вызов метода show() класса A
    }
}
```

Эта программа создает следующий вывод:

```
Это k: 3
i и j: 1 2
```

Версия метода `show()`, определенная в классе `B`, принимает строковый параметр. В результате ее сигнатура типа отличается от сигнатуры метода в классе `B`, который не принимает никаких параметров. Поэтому никакое переопределение (или сокрытие имени) не происходит. Вместо этого просто выполняется перегрузка версии метода `show()`, определенной в классе `A`, версией, определенной в классе `B`.

Динамическая диспетчеризация методов

Хотя приведенные в предыдущем разделе примеры демонстрируют механизм переопределения методов, они не показывают всех возможностей. Действительно, если бы переопределение методов служило лишь для удобства работы с пространством имен, оно представляло бы только определенный теоретический интерес, но имело бы очень небольшое практическое значение. Однако это не так. Переопределение методов служит основой для одной из наиболее мощных концепций Java — *динамической диспетчеризации методов*. Динамическая диспетчеризация методов — механизм, посредством которого разрешение обращения к переопределенному методу осуществляется во время выполнения, а не во время компиляции.

Динамическая диспетчеризация методов важна потому, что именно с ее помощью Java реализует полиморфизм времени выполнения.

Рассмотрение этой концепции начнем с повторной формулировки одного важного принципа: ссылочная переменная суперкласса может ссылаться на объект подкласса. Система Java использует этот факт для разрешения обращений к переопределенным методам во время выполнения. Вот как это происходит. Когда вызов переопределенного метода реализуется посредством ссылки на суперкласс, Java выбирает нужную версию этого метода в зависимости от типа объекта ссылки в момент вызова. Таким образом, этот выбор осуществляется во время выполнения. При ссылке на различные типы объектов программа будет обращаться к различным версиям переопределенного метода. Иначе говоря, выбор для выполнения версии переопределенного метода осуществляется в зависимости от *типа объекта ссылки* (а не от типа ссылочной переменной). Следовательно, если суперкласс содержит метод, переопределяемый подклассом, то при наличии ссылки на различные типы объектов через ссылочную переменную суперкласса программа будет выполнять различные версии метода.

В следующем примере иллюстрируется динамическая диспетчеризация методов.

```
// Динамическая диспетчеризация методов
class A {
  void callme() {
    System.out.println("Внутри метода callme класса A");
  }
}
class B extends A {
  // переопределение метода callme()
  void callme() {
    System.out.println("Внутри метода callme класса B");
  }
}
class C extends A {
  // переопределение метода callme()
  void callme() {
    System.out.println("Внутри метода callme класса C");
  }
}
class Dispatch {
  public static void main(String args[]) {
    A a = new A();  // объект типа A
    B b = new B();  // объект типа B
    C c = new C();  // объект типа C
    A r;            // получение ссылки типа A
    r = a;          // r ссылается на объект A
    r.callme();     // вызов версии метода callme, определенной в A
    r = b;          // r ссылается на объект B
    r.callme();     // вызов версии метода callme, определенной в B
    r = c;          // r ссылается на объект C
    r.callme();     // вызов версии метода callme, определенной в C
  }
}
```

Эта программа генерирует следующий вывод:

```
Внутри метода callme класса A
Внутри метода callme класса B
Внутри метода callme класса C
```

Эта программа создает один суперкласс `A` и два его подкласса: `B` и `C`. Подклассы `B` и `C` переопределяют метод `callme()`, объявленные в классе `A`. Внутри метода `main()` программа объявляет объекты типов `A`, `B` и `C`. Программа объявляет также ссылку типа `A` по имени `r`. Затем программа по очереди присваивает переменной `r` ссылку на каждый тип объекта и использует эту ссылку для вызова метода `callme()`. Как видно из вывода, выполняемая версия метода `callme()` определяется по типу объекта ссылки во время выполнения. Если бы выбор осуществлялся по типу ссылочной переменной, `r`, вывод отражал бы три обращения к методу `callme()` класса `A`.

На заметку! *Те читатели, которые знакомы с C++ или C#, должны заметить, что переопределенные методы в Java подобны виртуальным функциям в этих языках.*

Для чего нужны переопределенные методы?

Как уже было сказано, переопределенные методы позволяют Java поддерживать полиморфизм времени выполнения. Большое значение полиморфизма для объектно-ориентированного программирования обусловлено следующей причиной: он позволяет общему классу указывать методы, которые станут общими для всех его производных классов, в то же время позволяя подклассам определять конкретные реализации некоторых или всех этих методов. Переопределенные методы — еще один используемый в Java способ реализации аспекта полиморфизма под названием "один интерфейс, множество методов".

Одно из основных условий успешного применения полиморфизма — понимание того, что суперклассы и подклассы образуют иерархию по степени увеличения специализации. В случае его правильного применения суперкласс предоставляет все элементы, которые подкласс может использовать непосредственно. Он определяет также те методы, которые производный класс должен реализовать самостоятельно. Это позволяет подклассу определять собственные методы при сохранении единообразия интерфейса. Таким образом, объединяя наследование и переопределенные методы, суперкласс может определять общую форму методов, которые будут использоваться всеми его подклассами.

Динамический, реализуемый во время выполнения полиморфизм — один из наиболее мощных механизмов объектно-ориентированной архитектуры, обеспечивающих повторное использование и надежность кода. Возможность существующих библиотек кода вызывать методы применительно к экземплярам новых классов без повторной компиляции при сохранении четкого абстрактного интерфейса — чрезвычайно мощное средство.

Использование переопределения методов

Рассмотрим более реальный пример использования переопределения методов. Следующая программа создает суперкласс `Figure`, который хранит размеры двумерного объекта. Она определяет также метод `area()`, который вычисляет площадь объекта. Программа создает два класса, производных от класса `Figure`: `Rectangle` и `Triangle`. Каждый из этих подклассов переопределяет метод `area()`, чтобы он возвращал соответственно площадь четырехугольника и треугольника.

```
// Применение полиморфизма времени 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);
   }
   // переопределение метода 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);
    }
    // переопределение метода 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 создает следующий вывод:

```
В области четырехугольника.
Площадь равна 45
В области треугольника.
Площадь равна 40
Область фигуры не определена.
Площадь равна 0
```

Двойственный механизм наследования и полиморфизма времени выполнения позволяет определить единый интерфейс, используемый несколькими различными, но сходными типами объектов. В данном случае, если объект является производным от `Figure`, его площадь можно вычислять, вызывая метод `area()`. Интерфейс выполнения этой операции остается неизменным, независимо от типа фигуры.

Использование абстрактных классов

В ряде ситуаций нужно будет определять суперкласс, который объявляет структуру определенной абстракции без предоставления полной реализации каждого метода. То есть иногда придется создавать суперкласс, определяющий только обобщенную форму, которую будут совместно использовать все его подклассы, добавляя необходимые детали. Такой класс определяет сущность методов, которые должны реализовать подклассы. Например, такая ситуация может возникать, когда суперкласс не в состоянии создать полноценную реализацию метода. Именно такая ситуация имела место в классе `Figure` в предыдущем примере. Определение метода `area()` — просто шаблон. Он не будет вычислять и отображать площадь объекта какого-либо типа.

Как вы убедитесь в процессе создания собственных библиотек классов, отсутствие полного определения метода в контексте суперкласса — не столь уж редкая ситуация. Эту проблему можно решать двумя способами. Один из них, как было показано в предыдущем примере — просто вывод предупреждающего сообщения. Хотя этот подход и полезен в определенных ситуациях — например, при отладке — обычно он не годится. Могут существовать методы, которые должны быть переопределены подклассом, чтобы подкласс имел какой-либо смысл. Рассмотрим класс `Triangle`. Он лишен всякого смысла, если метод `area()` не определен. В этом случае необходим способ убедиться в том, что подкласс действительно переопределяет все необходимые методы. В Java для этого служит *абстрактный метод*.

Потребовать, чтобы определенные методы переопределялись подклассом, можно посредством указания модификатора типа `abstract`. Иногда такие методы называют относящимися к *компетенции подкласса*, поскольку в суперклассе для них никакой реализации не предусмотрено. Таким образом, подкласс должен переопределять эти методы — он не может просто использовать версию, определенную в суперклассе. Для объявления абстрактного метода используют следующую общую форму:

```
abstract тип имя(список_параметров);
```

Как видите, в этой форме тело метода отсутствует.

Любой класс, который содержит один или более абстрактных методов, должен быть также объявлен как абстрактный. Для этого достаточно поместить ключевое слово `abstract` перед ключевым словом `class` в начале объявления класса. Абстрактный класс не может содержать какие-то объекты. То есть абстрактный класс не может быть непосредственно конкретизирован с помощью операции `new`. Такие объекты были бы бесполезны, поскольку абстрактный класс определен не полностью. Нельзя также объявлять абстрактные конструкторы или абстрактные статические методы. Любой подкласс абстрактного класса должен либо реализовать все абстрактные методы суперкласса, либо также быть объявлен абстрактным.

Ниже приведен простой пример класса, содержащего абстрактный метод, и класса, который реализует этот метод.

```
// Простой пример применения 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 класса B.");
  }
}

class AbstractDemo {
  public static void main(String args[]) {
    B b = new B();
    b.callme();
    b.callmetoo();
  }
}
```

Обратите внимание, что в этой программе класс A не содержит объявлений каких-либо объектов. Как уже было сказано, конкретизация абстрактного класса невозможна. И еще один нюанс: класс A реализует конкретный метод callmetoo(). Это вполне допустимо. Абстрактные классы могут содержать любое необходимое количество конкретных реализаций.

Хотя абстрактные классы не могут быть использованы для конкретизации объектов, их можно применять для создания ссылок на объекты, поскольку в Java полиморфизм времени выполнения реализован посредством ссылок на суперкласс. Поэтому должна существовать возможность создания ссылки на абстрактный класс, которая может использоваться для указания на объект подкласса. Применение этого свойства показано в следующем примере.

Используя абстрактный класс, можно усовершенствовать созданный ранее класс Figure. Поскольку понятие площади неприменимо к неопределенной двумерной фигуре, следующая версия программы объявляет метод area() внутри класса Figure как abstract. Конечно, это означает, что все классы, производные от Figure, должны переопределять метод area().

```
// Использование абстрактных методов и классов.
abstract class Figure {
   double dim1;
   double dim2;
   Figure(double a, double b) {
      dim1 = a;
      dim2 = b;
   }
   // теперь метод area является абстрактным
      abstract double area();
}
class Rectangle extends Figure {
   Rectangle(double a, double b) {
      super(a, b);
   }
   // переопределение метода 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);
   }
   // переопределение метода area для четырехугольника
   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; // этот оператор допустим, никакой объект не создается
```

```
      figref = r;
      System.out.println("Площадь равна " + figref.area());
      figref = t;
      System.out.println("Площадь равна " + figref.area());
    }
}
```

Как видно из комментария внутри метода `main()`, объявление объектов типа `Figure` более недопустимо, поскольку теперь этот класс является абстрактным. И все подклассы класса `Figure` должны переопределять метод `area()`. Чтобы убедиться в этом, попытайтесь создать подкласс, который не переопределяет метод `area()`. Это приведет к ошибке времени компиляции.

Хотя создание объекта типа `Figure` недопустимо, можно создать ссылочную переменную типа `Figure`. Переменная `figref` объявлена как ссылка на `Figure` — т.е. ее можно использовать для ссылки на объект любого класса, производного от `Figure`. Как мы уже поясняли, разрешение переопределенных методов во время выполнения осуществляется путем ссылки на суперкласс.

Использование ключевого слова `final` в сочетании с наследованием

Существуют три способа использования ключевого слова `final`. Во-первых, его можно применять для создания эквивалента именованной константы. Это применение было описано в предыдущей главе. Остальные два применения относятся к наследованию. Давайте рассмотрим их.

Использование ключевого слова `final` для предотвращения переопределения

Хотя переопределение методов — одно из наиболее мощных средств Java, в некоторых случаях его желательно избежать. Чтобы запретить переопределение метода, в начале его объявления необходимо указать ключевое слово `final`. Методы, объявленные как `final`, переопределяться не могут. Следующий фрагмент кода иллюстрирует это применение ключевого слова `final`.

```
class A {
  final void meth() {
    System.out.println("Это метод final.");
  }
}
class B extends A {
  void meth() { // ОШИБКА! Этот метод не может быть переопределен.
    System.out.println("Не допускается!");
  }
}
```

Поскольку метод `meth()` объявлен как `final`, он не может быть переопределен в классе `B`. Попытка выполнить это переопределение приведет к ошибке времени компиляции.

Иногда методы, объявленные как `final`, могут способствовать увеличению производительности программы. Компилятор вправе вставлять вызовы этих методов *непосредственно в строку*, поскольку он "знает", что они не будут переопределены подклассом. Часто

при вызове небольшого метода типа final компилятор Java может копировать байт-код подпрограммы непосредственно в строку скомпилированного кода вызывающего метода, тем самым снижая значительные накладные расходы системных ресурсов, связанные с вызовом метода. Помещение методов типа final в строку вызывающего кода — лишь потенциальная возможность. Обычно Java разрешает вызовы методов динамически, во время выполнения. Такой подход называют *поздним связыванием*. Однако поскольку методы типа final не могут переопределяться, обращение к такому методу может быть разрешено во время компиляции. Этот подход называют *ранним связыванием*.

Использование ключевого слова final для предотвращения наследования

Иногда будет требоваться предотвратить наследование класса. Для этого в начале объявления класса необходимо поместить ключевое слово final. Объявление класса как final неявным образом объявляет все его методы также как final. Как легко догадаться, одновременное объявление класса как abstract и как final недопустимо, поскольку абстрактный класс принципиально является незавершенным и только его подклассы предоставляют полную реализацию методов.

Ниже приведен пример класса типа final.

```
final class A {
  // ...
}
// Следующий класс недопустим.
class B extends A { // ОШИБКА! Класс A не может иметь подклассы.
  // ...
}
```

Как видно из комментария, класс B не может наследовать от класса A, поскольку A объявлен как final.

Класс Object

В Java определен один специальный класс — Object. Все остальные классы являются подклассами этого класса. То есть Object — суперкласс всех остальных классов. Это означает, что ссылочная переменная типа Object может ссылаться на объект любого другого класса. Кроме того, поскольку массивы реализованы в виде классов, переменная типа Object может ссылаться также на любой массив.

Класс Object определяет методы, описанные в табл. 8.1, которые доступны в любом объекте.

Методы getClass(), notify(), notifyAll() и wait() объявлены как final. Остальные методы можно переопределять. Эти методы описаны в других главах книги. Однако обратите внимание на два метода: equals() и toString(). Метод equals() сравнивает содержимое двух объектов. Если объекты эквивалентны, он возвращает значение true, если нет — false.

Точное определение равенства зависит от типа сравниваемых объектов. Метод toString() возвращает строку, которая содержит описание объекта, по отношению к которому он вызван. Кроме того, этот метод автоматически вызывается при выводе объекта с помощью метода println(). Многие классы переопределяют этот метод. Это позволяет им приспосабливать описание специально для создаваемых ими объектных типов.

Таблица 8.1. Методы класса `Object`

Метод	Назначение
`Object clone()`	Создает новый объект, не отличающийся от клонируемого объекта.
`boolean equals(Object object)`	Определяет, равен ли один объект другому.
`void finalize()`	Вызывается перед удалением неиспользуемого объекта.
`Class getClass()`	Получает класс объекта во время выполнения.
`int hashCode()`	Возвращает хеш-код, связанный с вызывающим объектом.
`void notify()`	Возобновляет выполнение потока, который ожидает вызывающего объекта.
`void notifyAll()`	Возобновляет выполнение всех потоков, которые ожидают вызывающего объекта.
`String toString()`	Возвращает строку, которая описывает объект.

`void wait()`
`void wait(long milliseconds)`
`void wait(long milliseconds,`
`int nanoseconds)`
	Ожидает другого потока выполнения.

9
ГЛАВА

Пакеты и интерфейсы

В этой главе рассматриваются две наиболее новаторских концепции Java: пакеты и интерфейсы. *Пакеты* — это контейнеры классов, которые используются для сохранения изолированности пространства имен класса. Например, пакет позволяет создать класс по имени `List`, который можно хранить в отдельном пакете, не беспокоясь о возможных конфликтах с другим классом `List`, хранящимся в каком-то другом месте. Пакеты хранятся в иерархической структуре и явно импортируются в определения новых классов.

В предшествующих главах было описано использование методов для определения интерфейса к данным класса. С помощью ключевого слова `interface` Java позволяет полностью абстрагировать интерфейс от его реализации. Используя это ключевое слово, можно указать набор методов, которые могут быть реализованы одним или более классов. В действительности сам по себе интерфейс не определяет никакой реализации. Хотя они подобны абстрактным классам, интерфейсы предоставляют дополнительную возможность: один класс может реализовать более одного интерфейса. И наоборот, класс может наследоваться только от одного суперкласса (абстрактного или не абстрактного).

Пакеты

В предшествующих главах для всех примеров классов мы использовали имена из одного пространства имен. Это означает, что во избежание конфликта имен для каждого класса нужно было указывать уникальное имя. По истечении некоторого времени при отсутствии какого-либо способа управления пространством имен может возникнуть ситуация, когда выбор удобных описательных имен отдельных классов станет затруднительным. Кроме того, требуется также какой-нибудь способ обеспечения того, чтобы выбранное имя класса было достаточно уникальным и не конфликтовало с именами классов, выбранными другими программистами. (Представьте себе небольшую группу программистов, спорящих о том, кто имеет право использовать имя “Foobar” в качестве имени класса. Или вообразите себе все сообщество Internet, спорящее о том, кто первым назвал класс “Espresso”.) К счастью, Java предоставляет механизм разбиения пространства имен на более удобные для управления фрагменты. Этим механизмом служит пакет. Пакет служит одновременно механизмом и присвоения имен, и управления видимостью.

Внутри пакета можно определить классы, не доступные коду вне этого пакета. Можно также определить члены класса, которые видны только другим членам этого же пакета. Такой механизм позволяет классам располагать полными сведениями друг о друге, но не предоставлять эти сведения остальному миру.

Определение пакета

Создание пакета является простой задачей: достаточно включить команду `package` в качестве первого оператора исходного файла Java. Любые классы, объявленные внутри этого файла, будут принадлежать указанному пакету. Оператор `package` определяет пространство имен, в котором хранятся классы. Если оператор `package` опущен, имена классов помещаются в используемый по умолчанию пакет без имени. (Именно поэтому до сих пор нам не нужно было беспокоиться об определении пакетов.) Хотя для коротких примеров программ пакет, используемый по умолчанию, вполне подходит, он не годится для реальных приложений. В большинстве случаев для кода придется определять пакет.

Оператор `package` имеет следующую общую форму:

```
package пакет;
```

Пакет задает имя пакета. Например, показанный ниже оператор создает пакет `MyPackage`:

```
package MyPackage;
```

Для хранения пакетов система Java использует каталоги файловой системы. Например, файлы `.class` любых классов, объявленных в качестве составной части пакета `MyPackage`, должны храниться в каталоге `MyPackage`. Помните, что регистр символов имеет значение, а имя каталога должно в точности совпадать имени пакета.

Один и тот же оператор `package` может присутствовать в более чем одном файле. Этот оператор просто указывает пакет, к которому принадлежат классы, определенные в данном файле. Он не препятствует тому, чтобы другие классы в других файлах были частью этого же пакета. Большинство пакетов, используемых в реальных программах, распределено по множеству файлов.

Java позволяет создавать иерархию пакетов. Для этого применяется символ точки. Оператор многоуровневого пакета имеет следующую общую форму:

```
package пакет1[.пакет2[.пакет3]];
```

Иерархия пакетов должна быть отражена в файловой системе среды разработки Java. Например, в среде Windows пакет, объявленный как:

```
package java.awt.image;
```

должен храниться в каталоге `java\awt\image`. Необходимо тщательно проверять правильность выбора имен пакетов. Имя пакета нельзя изменить, не изменяя имя каталога, в котором хранятся классы.

Поиск пакетов и переменная среды CLASSPATH

Как было сказано в предыдущем разделе, пакеты отображаются на каталоги. Это обстоятельство порождает важный вопрос: откуда системе времени выполнения Java известно, где следует искать создаваемые пакеты? Ответ на него состоит из следующих частей: во-первых, по умолчанию в качестве отправной точки система времени выполнения Java использует текущий рабочий каталог. Следовательно, если пакет находится в подкаталоге

текущего каталога, он будет найден. Во-вторых, путь или пути к каталогу можно указать, устанавливая значение переменной среды CLASSPATH. В-третьих, java и javac можно использовать с параметром -classpath, указывающим путь к классам.

Например, рассмотрим следующую спецификацию пакета:

```
package MyPack;
```

Чтобы программа могла найти пакет MyPack, должно выполняться одно из следующих двух условий. Либо программа должна выполняться из каталога, расположенного непосредственно над каталогом MyPack, либо переменная среды CLASSPATH должна содержать путь к каталогу MyPack, либо параметр -classpath должен указывать путь к каталогу MyPack во время выполнения программы с помощью java.

При использовании двух последних способов путь класса *не должен содержать* сам пакет MyPack. Он должен просто указывать *путь* к этому каталогу. Например, в среде Windows, если путь к каталогу MyPack имеет вид:

```
C:\MyPrograms\Java\MyPack
```

то путь класса к MyPack будет выглядеть так:

```
C:\MyPrograms\Java
```

Простейший способ проверки примеров, приведенных в этой книге — просто создание каталогов пакетов в текущем каталоге разработки, помещение файлов .class в соответствующие каталоги и последующий запуск программ из каталога разработки. В следующем примере использован именно этот подход.

Краткий пример пакета

Памятуя о приведенных в предыдущем разделе соображениях, можете попытаться использовать следующий простой пакет:

```
// 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 файл AccountBalance.java и поместите его в каталог MyPack.

Затем выполните компиляцию файла. Убедитесь, что результирующий файл .class также помещен в каталог MyPack. Затем попробуйте выполнить класс AccountBalance, вводя следующую командную строку:

```
java MyPack.AccountBalance
```

Помните, что при выполнении этой команды текущим должен быть каталог, расположенный над каталогом MyPack, либо переменная среды CLASSPATH должна содержать соответствующий путь.

Как мы уже поясняли, теперь класс AccountBalance — часть пакета MyPack. Это означает, что его нельзя выполнять самостоятельно. То есть нельзя использовать следующую командную строку:

```
java AccountBalance
```

Имя AccountBalance требует уточнения именем его пакета.

Защита доступа

В предшествующих главах вы узнали о различных аспектах механизма управления доступом Java и его спецификаторах. Например, вы уже знаете, что доступ к приватному члену класса предоставляется только другим членам этого класса. Пакеты добавляют к управлению доступом еще одно измерение. Как вы вскоре убедитесь, Java предоставляет множество уровней защиты, обеспечивая очень точное управление видимостью переменных и методов внутри классов, подклассов и пакетов.

Классы и пакеты одновременно служат средствами инкапсуляции и хранилищем пространства имен и области определения переменных и методов. Пакеты играют роль контейнеров классов и других подчиненных пакетов. Классы служат контейнерами данных и кода. Класс — наименьшая единица абстракции Java. Вследствие взаимодействия между классами и пакетами Java определяет четыре категории видимости 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 спецификатора доступа — private, public и protected — предоставляют разнообразные способы создания множество уровней доступа, необходимых для этих категорий. Взаимосвязь между ними описана в табл. 9.1.

Хотя на первый взгляд механизм управления доступом Java может показаться сложным, следующие соображения могут облегчить его понимание. Любой компонент, объявленный как public, доступен из любого кода. Любой компонент, объявленный как private, не виден для компонентов, расположенных вне его класса. Если член не содержит явного спецификатора доступа, он видим подклассам и другим классам в данном пакете. Этот уровень доступа используется по умолчанию. Если нужно, чтобы элемент был виден за пределами его текущего пакета, но только классам, которые являются непосредственными подклассами данного класса, элемент должен быть объявлен как protected.

Правила доступа, описанные в табл. 9.1, применимы только к членам класса. Для класса, не являющегося вложенным, может быть указан только один из двух возможных уровней доступа: заданный по умолчанию и public. Когда класс объявлен как public, он

доступен любому другому коду. Если для класса указан уровень доступа, определенный по умолчанию, он доступен только для кода внутри данного пакета. Когда класс является общедоступным, он должен быть единственным общедоступным классом, объявленным в файле, и имя файла должно совпадать с именем класса.

Таблица 9.1. Доступ к членам класса

	Private	Модификатор отсутствует	Protected	Public
Один и тот же класс	Да	Да	Да	Да
Подкласс класса этого же пакета	Нет	Да	Да	Да
Класс этого же пакета, не являющийся подклассом	Нет	Да	Да	Да
Подкласс класса другого пакета	Нет	Нет	Да	Да
Класс другого пакета, не являющийся подклассом класса данного пакета	Нет	Нет	Нет	Да

Пример защиты доступа

Следующий пример демонстрирует использование всех комбинаций модификаторов управления доступом. Он содержит два пакета и пять классов. Не забудьте, что классы двух различных пакетов должны храниться в каталогах, имена которых совпадают с именами соответствующих пакетов — в данном случае `p1` и `p2`.

Исходный файл первого пакета определяет три класса `Protection`, `Derived` и `SamePackage`. Первый класс определяет четыре переменных `int` — по одной в каждом из допустимых режимов защиты доступа. Переменная `n` объявлена с уровнем защиты, используемым по умолчанию, `n_pri` — как `private`, `n_pro` — `protected`, а `n_pub` — `public`.

В этом примере все другие классы будут предпринимать попытку обращения к переменным экземпляра этого класса. Строки, компиляция которых невозможна из-за нарушений правил доступа, оформлены в виде комментариев. Перед каждой из этих строк помещен комментарий с указанием точек программы, из которых был бы возможен доступ к этому уровню защиты.

Второй класс, `Derived` — подкласс класса `Protection` этого же пакета `p1`. Он предоставляет классу `Derived` доступ ко всем переменным класса `Protection`, кроме переменной `n_pri`, объявленной как `private`. Третий класс, `SamePackage`, не является подклассом класса `Protection`, но он 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 = "4 + 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);
//   class only
//   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 оставшиеся две ситуации управления доступом. Первый класс, `Protection2` — это подкласс класса `p1.Protection`. Он имеет доступ ко всем переменным класса `p1.Protection`, кроме `n_pri` (поскольку она объявлена как `private`) и `n`, которая объявлена с уровнем защиты, используемым по умолчанию. Вспомните, что заданный по умолчанию режим доступа 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`, которая была объявлена как `public`.

Файл `Protection2.java` содержит следующий код:

```
package p2;
class Protection2 extends p1.Protection {
  Protection2() {
    System.out.println("унаследованный конструктор 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 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 проверки работы этих двух пакетов можно использовать следующие два тестовых файла. Тестовый файл для пакета `p1` имеет вид:

```
// Демонстрационный пакет p1.
package p1;
// Конкретизация различных классов пакета p1.
public class Demo {
   public static void main(String args[]) {
      Protection ob1 = new Protection();
      Derived ob2 = new Derived();
      SamePackage ob3 = new SamePackage();
   }
}
```

Следующий файл — тестовый файл пакета `p2`:

```
// Демонстрационный пакет p2.
package p2;
// Конкретизация различных классов пакета p2.
public class Demo {
   public static void main(String args[]) {
      Protection2 ob1 = new Protection2();
      OtherPackage ob2 = new OtherPackage();
   }
}
```

Импорт пакетов

Если вспомнить, что пакеты предлагают эффективный механизм изоляции различных классов друг от друга, 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 хранятся в каком-либо именованном пакете. Поскольку внутри пакетов классы должны быть полностью определены именем

или именами их пакетов, длинное, разделенное точками имя пути пакета каждого используемого класса может оказаться слишком громоздким. Поэтому, чтобы определенные классы или весь пакет можно было сделать видимыми, в Java включен оператор `import`. После того как класс импортирован, на него можно ссылаться непосредственно, используя только его имя. Оператор `import` служит только для удобства программистов и не является обязательным с технической точки зрения для создания завершенной Java-программы. Однако если в приложении придется ссылаться на несколько десятков классов, оператор `import` значительно уменьшит объем вводимого кода.

В исходном файле Java-программы операторы `import` должны следовать непосредственно за оператором `package` (если таковой имеется) перед любыми определениями классов. Оператор `import` имеет следующую общую форму:

```
import пакет1[.пакет2].(имя_класса|*);
```

В этой форме *пакет1* — имя пакета верхнего уровня, *пакет2* — имя подчиненного пакета внутри внешнего пакета, отделенное символом точки (`.`). Глубина вложенности пакетов практически не ограничена ничем, кроме файловой системы. И, наконец, *имя_класса* может быть задано либо явно, либо с помощью символа звездочки (`*`), который указывает компилятору Java о необходимости импорта всего пакета. Следующий фрагмент демонстрирует применение обеих форм оператора:

```
import java.util.Date;
import java.io.*;
```

Внимание! *Использование формы с применением символа звездочки может привести к увеличению времени компиляции — особенно при импорте нескольких больших пакетов. Поэтому советуем явно указывать имена классов, которые нужно использовать, а не импортировать пакеты полностью. Однако использование формы с применением звездочки никак не влияет на производительность системы времени выполнения или на размеры классов.*

Все стандартные классы, поставляемые с системой Java, хранятся в пакете `java`. Основные функции языка хранятся в пакете `java.lang` внутри пакета `java`. Обычно каждый пакет или класс, который нужно использовать, приходится импортировать. Но поскольку система Java бесполезна без многих функций, определенных в пакете `java.lang`, компилятор неявно импортирует его для всех программ. Это эквивалентно присутствию следующей строки в каждой из программ:

```
import java.lang.*;
```

При наличии в двух различных пакетах, импортируемых с применением формы со звездочкой, классов с одинаковыми именами компилятор никак на это не отреагирует, если только не будет предпринята попытка использования одного из этих классов. В этом случае возникнет ошибка времени компиляции, и имя класса придется указать явно, задавая его пакет.

Полностью определенное имя класса с указанием полной иерархии пакетов можно использовать везде, где можно допускается имя класса. Например, в следующем фрагменте кода присутствует оператор импорта:

```
import java.util.*;
class MyDate extends Date {
}
```

Этот же пример без оператора `import` выглядит следующим образом:

```
class MyDate extends java.util.Date {
}
```

В этой версии объект `Date` полностью определен.

Как видно в табл. 9.1, при импорте пакета в импортирующем коде классам, не являющимся подклассами классов пакета, будут доступны только те элементы пакета, которые объявлены как `public`. Например, если нужно, чтобы приведенный ранее класс `Balance` пакета `MyPack` был доступен в качестве самостоятельного класса вне пакета `MyPack`, его необходимо объявить как `public` и поместить в отдельный файл, как показано в следующем примере:

```
/* Теперь класс Balance, его конструктор и его метод show()
   являются общедоступными. Это означает, что вне их пакета они
   могут использоваться кодом, не являющимся подклассом 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, теперь класс `Balance` объявлен как `public`. Его конструктор и метод `show()` также объявлены как `public`. Это означает, что они доступны любому коду вне пакета `MyPack`. Например, класс `TestBalance` импортирует пакет `MyPack` и поэтому может использовать класс `Balance`:

```
import MyPack.*;
class TestBalance {
   public static void main(String args[]) {
      /* Поскольку класс Balance объявлен как public, его можно
         использовать и вызывать его конструктор. */
      Balance test = new Balance("J. J. Jaspers", 99.88);
      test.show(); // можно также вызывать метод show()
   }
}
```

В качестве эксперимента удалите спецификатор `public` из класса `Balance`, а затем попытайтесь выполнить компиляцию класса `TestBalance`. Как уже было сказано, это приведет к возникновению ошибок.

Интерфейсы

Применение ключевого слова `interface` позволяет полностью абстрагировать интерфейс класса от его реализации. То есть с использованием ключевого слова `interface` можно задать действия, которые должен выполнять класс, но не то, как именно он должен это делать. Синтаксически интерфейсы аналогичны классам, но не содержат переменных

экземпляров, а объявления их методов не содержат тела 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 число интерфейсов.

Чтобы реализовать интерфейс, класс должен создать полный набор методов, определенных интерфейсом. Однако каждый класс может определять нюансы своей реализации данного интерфейса. Ключевое слово `interface` позволяет в полной мере использовать концепцию полиморфизма под названием "один интерфейс, несколько методов".

Интерфейсы предназначены для поддержки динамического разрешения методов во время выполнения. Обычно чтобы вызов метода мог выполняться из одного класса в другом, оба класса должны присутствовать во время компиляции, дабы компилятор Java мог проверить совместимость сигнатур методов. Само по себе это требование создает статическую и нерасширяемую среду обработки классов. В такой системе функциональные возможности неизбежно передаются по иерархии классов все выше и выше, в результате чего механизмы будут становиться доступными все большему числу подклассов. Интерфейсы предназначены для предотвращения этой проблемы. Они изолируют определение метода или набора методов от иерархии наследования. Поскольку иерархия интерфейсов не совпадает с иерархией классов, классы, никак не связанные между собой в иерархии классов, могут реализовать один и тот же интерфейс. Именно здесь возможности интерфейсов проявляются наиболее полно.

На заметку! *Интерфейсы добавляют большинство функциональных возможностей, требуемых многим приложениям, которым в обычных условиях в языках вроде C++ пришлось бы прибегать к использованию множественного наследования.*

Определение интерфейса

Определение интерфейса во многом подобно определению класса. Общая форма интерфейса имеет следующий вид:

```
доступ interface имя {
   возвращаемый_тип имя_метода1(список_параметров);
   возвращаемый_тип имя_метода2(список_параметров);
   тип имя_конечной_переменной1 = значение;
   тип имя_конечной_переменной2 = значение;
   // ...
   Возвращаемый_тип имя_методаN(список_параметров);
   тип имя_конечной_переменнойN = значение;
}
```

Если определение не содержит никакого спецификатора доступа, используется доступ по умолчанию, и интерфейс доступен только другим членам того пакета, в котором он объявлен. Если интерфейс объявлен как `public`, он может быть использован любым другим кодом. В этом случае интерфейс должен быть единственным общедоступным интерфейсом, объявленным в файле, и имя файла должно совпадать с именем интерфейса. *Имя* — имя интерфейса, которым может быть любой допустимый идентификатор. Обратите внимание, что объявляемые методы не содержат тел. Их объявления завершаются списком параметров, за которым следует символ точки с запятой. По сути, они представляют собой абстрактные методы. Ни один из указанных внутри интерфейса методов не может обладать никакой заданной по умолчанию реализацией. Каждый класс, который включает в себя интерфейс, должен реализовать все его методы.

Переменные могут быть объявлены внутри объявлений интерфейсов. Они неявно объявляются как `final` и `static` — т.е. реализующий класс не может их изменять. Кроме того, они должны быть также инициализированы. Все методы и переменные неявно объявляются как `public`.

Ниже приведен пример определения 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, его может реализовать один или более классов. Чтобы реализовать интерфейс, в определение класса потребуется включить конструкцию `implements`, а затем создать методы, определенные интерфейсом. Общая форма класса, который содержит выражение `implements`, имеет следующий вид:

```
доступ class имя_класса [extends суперкласс]
     [implements интерфейс [,интерфейс...]] {
  // тело_класса
}
```

Если класс реализует более одного интерфейса, имена интерфейсов разделяются запятыми. Если класс реализует два интерфейса, которые объявляют один и тот же метод, то один и тот же метод будет использоваться клиентами любого интерфейса. Методы, которые реализуют интерфейс, 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, который реализует приведенный ранее интерфейс `Callback`.

```
class Client implements Callback {
  // Реализует интерфейс Callback
  public void callback(int p) {
    System.out.println("Метод callback, вызванный со значением " + p);
  }
}
```

Обратите внимание, что метод `callback()` объявлен с использованием спецификатора доступа `public`.

Помните! *При реализации метода интерфейса он должен быть объявлен как* `public`.

Вполне допустима и достаточно распространена ситуация, когда классы, которые реализуют интерфейсы, определяют собственные дополнительные члены. Например, следующая версия класса `Client` реализует метод `callback()` и добавляет метод `nonInfaceMeth()`:

```
class Client implements Callback {
  // Реализует интерфейс Callback
  public void callback(int p) {
    System.out.println("Метод callback, вызванный со значением " + p);
  }
```

```
    void nonIfaceMeth() {
      System.out.println("Классы, которые реализуют интерфейсы" +
                         "могут определять также и другие члены.");
    }
}
```

Доступ к реализациям через ссылки на интерфейсы

Переменные можно объявлять как объектные ссылки, которые используют тип интерфейса, а не тип класса. Посредством такой переменной можно ссылаться на любой экземпляр любого класса, реализующего объявленный интерфейс. При вызове метода с помощью одной из таких ссылок выбор нужной версии будет производиться в зависимости от конкретного экземпляра интерфейса, на который выполняется ссылка. Это — одна из главных особенностей интерфейсов. Поиск выполняемого метода осуществляется динамически во время выполнения, что позволяет создавать классы позже, чем код, который вызывает методы по отношению к этим классам. Диспетчеризация кода может выполняться посредством интерфейса без необходимости наличия каких-либо сведений о "вызывающем". Этот процесс аналогичен использованию ссылки на суперкласс для доступа к объекту подкласса, описанному в главе 8.

Внимание! *Поскольку в системе Java динамический поиск методов во время выполнения сопряжен со значительными накладными расходами по сравнению с обычным вызовом методов, в коде, для которого важна производительность, интерфейсы следует использовать только тогда, когда это действительно необходимо.*

В следующем примере метод `callback()` вызывается через ссылочную переменную интерфейса:

```
class TestIface {
  public static void main(String args[]) {
    Callback c = new Client();
    c.callback(42);
  }
}
```

Эта программа создает следующий вывод:

```
Метод callback, вызванный со значением 42
```

Обратите внимание, что хотя переменная `c` объявлена с типом интерфейса `Callback`, ей был присвоен экземпляр класса `Client`. Хотя переменную `c` можно использовать для доступа к методу `callback()`, она не имеет доступа к каким-то другим членам класса `Client`. Ссылочная переменная интерфейса располагает сведениями только о тех методах, которые объявлены ее объявлением `interface`. Таким образом, переменная `c` не может применяться для доступа к методу `nonIfaceMeth()`, поскольку она объявлена классом `Client`, а не классом `Callback`.

Хотя приведенный пример формально показывает, как ссылочная переменная интерфейса может получать доступ к объекту реализации, он не демонстрирует полиморфные возможности такой ссылки. Чтобы продемонстрировать пример такого применения, вначале создадим вторую реализацию интерфейса `Callback`:

```
// Еще одна реализация интерфейса Callback.
class AnotherClient implements Callback {
  // Реализация интерфейса Callback
```

```
  public void callback(int p) {
    System.out.println("Еще одна версия callback");
    System.out.println("p в квадрате равно " + (p*p));
  }
}
```

Теперь проверим работу следующего класса:

```
class TestIface2 {
  public static void main(String args[]) {
    Callback c = new Client();
    AnotherClient ob = new AnotherClient();
    c.callback(42);
    c = ob; // теперь c ссылается на объект AnotherClient
    c.callback(42);
  }
}
```

Эта программа создает следующий вывод:

```
callback вызванный со значением 42
Еще одна версия callback
p в квадрате равно 1764
```

Как видите, вызываемая версия метода `callback()` определяется типом объекта, на который переменная `c` ссылается во время выполнения. Представленный пример очень прост, поэтому вскоре мы приведем еще один, более реальный пример.

Частичные реализации

Если класс содержит интерфейс, но не полностью реализует определенные им методы, он должен быть объявлен как `abstract` (абстрактный). Например:

```
abstract class Incomplete implements Callback {
  int a, b;
  void show() {
    System.out.println(a + " " + b);
  }
  // ...
}
```

В этом примере класс `Incomplete` не реализует метод `callback()` и должен быть объявлен как абстрактный. Любой класс, который наследует `Incomplete`, должен реализовать метод `callback()` либо 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`. Это отличает его от интерфейса верхнего уровня, который должен быть либо объявлен как `public`, либо, как уже было отмечено, должен использовать уровень доступа, заданный по умолчанию. Когда вложенный интерфейс используется вне содержащей его области определения, он должен определяться именем класса или интерфейса, членом которого он является. То есть вне класса или интерфейса, в котором объявлен вложенный интерфейс, его имя должно быть полностью определено.

В следующем примере демонстрируется применение вложенного интерфейса.

```
// Пример вложенного интерфейса.
// Этот класс содержит интерфейс-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[]) {
   // использует ссылку на вложенный интерфейс
   A.NestedIF nif = new B();
   if(nif.isNotNegative(10))
   System.out.println("10 не является отрицательным");
   if(nif.isNotNegative(-12))
   System.out.println("это не будет отображаться");
   }
}
```

Обратите внимание, что объект `A` определяет интерфейс-член `NestedIF`, который объявлен как `public`. Затем объект `B` реализует вложенный интерфейс путем указания:

```
implements A.NestedIF
```

Обратите также внимание, что имя интерфейса полностью определено и содержит имя класса. Внутри метода `main()` создается ссылка на `A.NestedIF`, получившая имя `nif`, которой присваивается ссылка на объект `B`. Поскольку объект `B` реализует `A.NestedIF`, это допустимо.

Использование интерфейсов

Чтобы возможности интерфейсов были понятны, рассмотрим более реальный пример. В предшествующих главах мы разработали класс `Stack`, который реализует простой стек фиксированного размера. Однако существует множество способов реализации стека. Например, стек может иметь фиксированный размер, либо быть "увеличивающимся". Стек может также храниться в массиве, связанном списке, бинарном дереве и т.п. Независимо от реализации стека его интерфейс остается неизменным. То есть методы `push()` и `pop()` определяют интерфейс стека независимо от нюансов реализации. Поскольку интерфейс стека отделен от его реализации, можно без труда определить интерфейс стека, предоставляя реализации определение специфичных особенностей. Рассмотрим два примера.

Вначале создадим интерфейс, который определяет целочисленный стек. Поместим его в файл `IntStack.java`. Этот интерфейс будет использоваться обеими реализациями стека.

```
// Определение интерфейса целочисленного стека.
interface IntStack {
   void push(int item);  // сохранение элемента
   int pop();            // извлечение элемента
}
```

Следующая программа создает класс `FixedStack`, который реализует версию целочисленного стека фиксированной длины:

```
// Реализация IntStack, использующая область хранения фиксированного размера.
class FixedStack implements IntStack {
  private int stck[];
  private int tos;

  // резервирование и инициализация стека
  FixedStack(int size) {
    stck = new int[size];
    tos = -1;
  }

  // заталкивание элемента в стек
  public void push(int item) {
    if(tos==stck.length-1) // использование члена длины стека
      System.out.println("Стек полон.");
    else
      stck[++tos] = item;
  }

  // выталкивание элемента из стека
  public int pop() {
    if(tos < 0) {
      System.out.println("Стек пуст.");
      return 0;
    }
    else
      return stck[tos--];
  }
}

class IFTest {
  public static void main(String args[]) {
    FixedStack mystack1 = new FixedStack(5);
    FixedStack mystack2 = new FixedStack(8);

    // заталкивание чисел в стек
    for(int i=0; i<5; i++) mystack1.push(i);
    for(int i=0; i<8; i++) mystack2.push(i);

    // выталкивание этих чисел 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());
  }
}
```

Теперь создадим еще одну реализацию `IntStack`, которая, используя то же самое определение `interface`, создает динамический стек. В этой реализации каждый стек создается с начальной длиной. При превышении этой начальной длины размер стека увеличивается. Каждый раз, когда возникает потребность в дополнительном месте, размер стека удваивается.

```
// Реализация "увеличивающегося" стека.
class DynStack implements IntStack {
  private int stck[];
  private int tos;
  // резервирование и инициализация стека
  DynStack(int size) {
    stck = new int[size];
    tos = -1;
  }
  // Заталкивание элемента в стек
  public void push(int item) {
    // если стек полон, резервирование стека большего размера
    if(tos==stck.length-1) {
      int temp[] = new int[stck.length * 2]; // удвоение размера
      for(int i=0; i<stck.length; i++) temp[i] = stck[i];
      stck = temp;
      stck[++tos] = item;
    }
    else
      stck[++tos] = item;
  }
  // Выталкивание элемента из стека
  public int pop() {
    if(tos < 0) {
      System.out.println("Стек пуст.");
      return 0;
    }
    else
      return stck[tos--];
  }
}
class IFTest2 {
  public static void main(String args[]) {
    DynStack mystack1 = new DynStack(5);
    DynStack mystack2 = new DynStack(8);
    // Эти циклы увеличивают размеры каждого из стеков
    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());
  }
}
```

Следующий класс использует обе реализации `FixedStack` и `DynStack`. Он выполняет это посредством ссылки на интерфейс. Это означает, что разрешение обращений к методам `push()` и `pop()` осуществляется во время выполнения, а не во время компиляции.

```
/* Создание переменной интерфейса и
   обращение к стекам через нее.
*/
class IFTest3 {
  public static void main(String args[]) {
```

```
    IntStack mystack;   // создание ссылочной переменной интерфейса
    DynStack ds = new DynStack(5);
    FixedStack fs = new FixedStack(8);
    mystack = ds;       // загрузка динамического стека
    // заталкивание чисел в стек
    for(int i=0; i<12; i++) mystack.push(i);
    mystack = fs;       // загрузка фиксированного стека
    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 `mystack` — ссылка на интерфейс `IntStack`. Таким образом, когда она ссылается на переменную `ds`, программа использует версии методов `push()` и `pop()`, определенные реализацией `DynStack`. Когда же она ссылается на переменную `fs`, программа использует версии методов `push()` и `pop()`, определенные реализацией `FixedStack`. Как уже было сказано, эти решения принимаются во время выполнения. Обращение к нескольким реализациям интерфейса через ссылочную переменную интерфейса — наиболее мощный метод поддержки полиморфизма времени выполнения Java.

Переменные в интерфейсах

Интерфейсы можно применять для импорта совместно используемых констант в несколько классов посредством простого объявления интерфейса, который содержит переменные, инициализированные нужными значениями. При включении интерфейса в класс (т.е. при "реализации" интерфейса) имена всех этих переменных будут помещены в область констант. (Это аналогично использованию в программе C/C++ заголовочного файла для создания большого числа констант типа `#define` или объявлений `const`.) Если интерфейс не содержит никаких методов, любой класс, который включает в себя такой интерфейс, в действительности ничего не реализует. Это равносильно тому, что класс импортировал бы постоянные поля в пространство имен класса в качестве переменных типа `final`. В следующем примере эта технология 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, что в этой программе использован один из стандартных Java-классов — `Random`. Этот класс генерирует псевдослучайные числа. Он содержит несколько методов, которые позволяют получать случайные числа в требуемой программой форме. В этом примере применяется метод `nextDouble()`, который возвращает случайные числа в диапазоне от 0,0 до 1,0.

В приведенном 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 внутри каждого класса ссылается на эти константы так, как если бы каждый класс определял или наследовал их непосредственно. Ниже показан вывод, полученный в результате тестового

выполнения этой программы. Обратите внимание, что при каждом запуске результаты выполнения программы будут различными.

```
Позднее
Вскоре
Нет
Да
```

Возможность расширения интерфейсов

Ключевое слово `extends` позволяет одному интерфейсу наследовать другой. Синтаксис определения такого наследования аналогичен синтаксису наследования классов. Когда класс реализует интерфейс, который наследует другой интерфейс, он должен предоставлять реализации всех методов, определенных внутри цепочки наследования интерфейса. Ниже показан пример.

```
// Один интерфейс может расширять другой.
interface A {
  void meth1();
  void meth2();
}
// Теперь B включает в себя meth1() и meth2() и добавляет meth3().
interface B extends A {
  void meth3();
}
// Этот класс должен реализовать все методы классов A и B
class MyClass implements B {
  public void meth1() {
    System.out.println("Реализация meth1().");
  }
  public void meth2() {
    System.out.println("Реализация meth2().");
  }
  public void meth3() {
    System.out.println("Реализация 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 времени компиляции. Как уже было сказано, любой класс, который реализует интерфейс, должен реализовать все определенные этим интерфейсом методы, в том числе любые методы, унаследованные от других интерфейсов.

Хотя в приведенных в этой книге примерах пакеты или интерфейсы используются не очень часто, оба эти средства являются важными составляющими среды программирования Java. Буквально все реальные программы, написанные на Java, будут храниться в пакетах. Вполне вероятно, что многие из них будут также реализовать интерфейсы. Поэтому важно освоиться с их применением.

10 ГЛАВА

Обработка исключений

В этой главе рассматривается механизм обработки исключений Java. *Исключение* — это ненормальная ситуация, возникающая во время выполнения последовательности кода. Другими словами, исключение — это ошибка времени выполнения. В языках программирования, которые не поддерживают обработки исключений, ошибки должны проверяться и обрабатываться "вручную" — обычно, путем использования кодов ошибок и тому подобного. Этот подход как обременителен, так и чреват проблемами. Обработка исключений Java позволяет избежать этих проблем, а, кроме того, переносит управление ошибками времени выполнения в объектно-ориентированный мир.

Основы обработки исключений

Исключение Java представляет собой объект, который описывает исключительную (то есть, ошибочную) ситуацию, возникающую в части программного кода. Когда такая исключительная ситуация возникает, создается объект, представляющий исключение, который *возбуждается* в методе, вызвавшем ошибку. Этот метод может либо обработать исключение самостоятельно, либо пропустить его. В обоих случаях, в некоторой точке исключение *перехватывается* и обрабатывается. Исключения могут генерироваться системой времени выполнения Java, либо они могут быть сгенерированы вручную вашим кодом. Исключения, которые возбуждает Java, имеют отношение к фундаментальным ошибкам, которые нарушают правила языка Java либо ограничения системы выполнения Java. Вручную сгенерированные исключения обычно применяются для того, чтобы сообщить о некоторых ошибочных ситуациях тому, кто вызвал данный метод.

Обработка исключений Java управляется пятью ключевыми словами: `try`, `catch`, `throw`, `throws` и `finally`. Если кратко, они работают следующим образом. Операторы программы, которые вы хотите отслеживать на предмет исключений, помещаются в блок `try`. Если исключение возникает в блоке `try`, оно возбуждается. Ваш код может перехватить исключение (используя `catch`) и обработать его некоторым осмысленным способом. Сгенерированные системой исключения автоматически возбуждаются системой времени выполнения Java. Чтобы вручную возбудить исключение, используется ключевое слово `throw`. Любое исключение, которое возбуждается внутри метода, должно быть специфицировано в его интерфейсе ключевым словом `throws`. Любой код, который в обязатель-

ном порядке должен быть выполнен после завершения блока try, помещается в блок finally. Ниже показана общая форма блока обработки исключений.

```
try {
  // блок кода, в котором отслеживаются ошибки
}
catch (Тип_исключения_1 exOb) {
  // обработчик исключений типа ExceptionType1
}
catch (Тип_исключения_2 exOb) {
  // обработчик исключений типа ExceptionType2
}
// ...
finally {
  // блок кода, который должен быть выполнен после завершения блока try
}
```

Здесь *Тип_исключения* — тип исключения, которое возникает. Остаток настоящей главы посвящен описанию применения этой программной структуры.

Типы исключений

Все типы исключений являются подклассами встроенного класса Trowable. То есть Trowable расположен на вершине иерархии классов исключений. Немедленно под Trowable в ней находятся два подкласса, которые разделяют все исключения на две отдельные ветви. Одну ветвь возглавляет Exception. Этот класс используется для исключительных условий, которые пользовательская программа должна перехватывать. Это также класс, от которого вы будете наследовать свои подклассы при создании ваших собственных типов исключений. У класса Exception имеется важный подкласс по имени RuntimeException. Исключения этого типа автоматически определяются для программ, которые вы пишете, и включают такие вещи, как деление на ноль и ошибочная индексация массивов.

Другая ветвь начинается с класса Error, определяющего исключения, вызов которых не ожидается при нормальном выполнении программы. Исключения типа Error используются системой времени выполнения Java для обозначения ошибок, происходящих внутри самой окружения. Примером такой ошибки может служить переполнение стека. В этой главе не рассматриваются исключения типа Error, поскольку они обычно создаются в ответ на катастрофические сбои, которые не могут быть обработаны вашей программой.

Необработанные исключения

Прежде чем вы узнаете, как обрабатывать исключения в своей программе, полезно будет посмотреть, что происходит, когда вы не обрабатываете их. Следующая небольшая программа представляет пример, который намеренно вызывает ошибку деления на ноль:

```
class Exc0 {
public static void main(String args[]) {
  int d = 0;
  int a = 42 / d;
}
}
```

Когда система времени выполнения Java обнаруживает попытку деления на ноль, она конструирует новый объект исключения и затем *возбуждает* исключение. Это прерывает выполнение `Exc0`, поскольку как только исключение возбуждено, оно должно быть *перехвачено* обработчиком исключений, а тот должен немедленно с ним что-то сделать. В данном примере мы не применили никакого собственного обработчика исключений, поэтому исключение перехватывается обработчиком по умолчанию, предоставленным системой времени выполнения Java. Любое исключение, которое не перехвачено вашей программой, в конечном итоге будет перехвачено и обработано этим обработчиком по умолчанию. Обработчик по умолчанию отображает строку, описывающую исключение, печатает трассировку стека от точки возникновения исключения и прерывает программу. Ниже приведен пример исключения, сгенерированного представленным выше кодом:

```
java.lang.ArithmeticException: / by zero
      at Exc0.main(Exc0.java:4)
```

Обратите внимание, что имя класса `Exc0`, имя метода `main`, имя файла `Exc0.java` и номер строки `4` включены в трассировку стека. Также нужно обратить внимание на то, что возбужденное исключение является подклассом `Exception`, называемым `ArithmeticException`, который более точно описывает тип возникшей ошибки. Как будет показано далее в настоящей главе, Java применяет несколько встроенных типов исключений, соответствующих разным типам ошибок времени выполнения, которые могут быть сгенерированы.

Трассировка стека всегда покажет последовательность вызовов методов, которая привела к ошибке. Например, вот другая версия предыдущей программы, представляющая ту же ошибку, но в методе, отдельном от `main()`:

```
class Exc1 {
static void subroutine() {
  int d = 0;
  int a = 10 / d;
}
public static void main(String args[]) {
  Exc1.subroutine();
}
}
```

Результирующая трассировка стека обработчика исключений по умолчанию показывает весь стек вызовов:

```
java.lang.ArithmeticException: / by zero
      at Exc1.subroutine(Exc1.java:4)
      at Exc1.main(Exc1.java:7)
```

Как видите, в нижней части стека находится строка 7 метода `main`, в которой расположен вызов `subroutine()`, породивший исключение в строке 4. Трассировка стека достаточно удобна для отладки, поскольку показывает всю последовательность вызовов, приведших к ошибке.

Использование `try` и `catch`

Хотя обработчик исключений по умолчанию, который предоставляет система времени выполнения Java, удобен для отладки, обычно вы захотите обрабатывать исключения самостоятельно. Это дает два существенных преимущества. Во-первых, вы получаете

возможность исправить ошибку. Во-вторых, предотвращается автоматическое прерывание выполнения программы. Большинство пользователей будут недовольны (и это как минимум), если ваша программа будет останавливаться и распечатывать трассировку стека всякий раз при возникновении ошибки. К счастью, предотвратить это достаточно просто.

Чтобы противостоять и обрабатывать ошибки времени выполнения, нужно просто поместить код, который вы хотите наблюдать, внутрь блока `try`. Непосредственно за блоком `try` следует включить конструкцию `catch`, которая специфицирует тип перехватываемого исключения. Чтобы проиллюстрировать, насколько это просто делается, в следующую программу включен блок `try` с конструкцией `catch`, который обрабатывает исключение `ArithmeticException`, генерируемое в результате попытки деления на ноль.

```
class Exc2 {
public static void main(String args[]) {
   int d, a;
   try { // Мониторинг блока кода.
     d = 0;
     a = 42 / d;
     System.out.println("Это не будет напечатано.");
   } catch (ArithmeticException e) { // перехват ошибки деления на ноль
     System.out.println("Деление на ноль.");
   }
   System.out.println("После оператора catch.");
}
}
```

Эта программа создает следующий вывод:

```
Деление на ноль.
После оператора catch.
```

Обратите внимание, что вызов `println()` внутри блока `try` никогда не будет выполняться. Как только исключение возбуждено, управление передается из блока `try` в блок `catch`. То есть строка “Это не будет напечатано” не отображается. После того, как блок `catch` будет выполнен, управление передается на строку программы, следующую за всем механизмом `try/catch`.

Операторы `try` и `catch` составляют единый узел. Область действия `catch` не распространяется на те операторы, которые идут перед оператором `try`. Оператор `catch` не может перехватить исключение, возбужденное другим оператором `try` (кроме случаев вложенных конструкций `try`, которые будут описаны ниже). Операторы, которые защищены блоком `try`, должны быть заключены в фигурные скобки (т.е. они должны находиться внутри блока). Вы не можете применить `try` к отдельному оператору программы.

Целью правильно построенных операторов `catch` является разрешение исключительных ситуаций и продолжение работы, как если бы ошибки вообще не случались. Например, в следующей программе каждая итерация цикла `for` получает два случайных числа. Эти два числа делятся одно на другое, а результат используется для деления значения 12345. Окончательный результат помещается в `a`. Если какая-либо из операций деления вызывает ошибку деления на ноль, эта ошибка перехватывается, значение `a` устанавливается равным 0 и выполнение программы продолжается.

```
// Обработка исключения с продолжением работы.
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);
  }
}
}
```

Отображение описания исключения

`Throwable` переопределяет метод `toString()` (определенный в `Object`) таким образом, что он возвращает строку, содержащую описание исключения. Вы можете отобразить это описание с помощью `println()`, просто передав исключение в виде аргумента. Например, блок `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 подставляется в программу и программа запускается, каждая попытка деления на ноль отобразит следующее сообщение:

```
Исключение: java.lang.ArithmeticException: / by zero
```

Хотя в данном контексте это не имеет особого значения, все же возможность отобразить описание исключения в некоторых случаях полезна — в частности, когда вы экспериментируете с исключениями или занимаетесь отладкой.

Множественные операторы `catch`

В некоторых случаях один фрагмент кода может инициировать более одного исключения. Чтобы справиться с такой ситуацией, вы можете специфицировать два или более операторов `catch`, каждый для перехвата своего типа исключений. Когда возбуждается исключение, каждый оператор `catch` проверяется по порядку, и первый из них, чей тип соответствует исключению, выполняется. После того, как выполнится один из операторов `catch`, все остальные пропускаются, и выполнение программы продолжается с места, следующего за блоком `try/catch`. В следующем примере кода перехватываются два разных типа исключений.

```
// Демонстрация применения множественных операторов catch.
class MultiCatch {
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 0: " + e);
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Ошибка индекса массива: " + e);
    }
    System.out.println("После блока try/catch.");
  }
}
```

Эта программа вызовет исключение деления на ноль, если будет запущена без аргументов командной строки, поскольку в этом случае `a` будет равно 0. Она выполнит деление, если будут передан аргумент командной строки, устанавливающий `a` равным значению больше нуля. Но в этом случае будет сгенерировано исключение `ArrayIndexOutOfBoundsException`, так как длина массива целых чисел `c` равна 1, в то время как программа пытается присвоить значение элементу массива `c[42]`.

Вот результаты запуска этой программы обоими способами:

```
C:\>java MultiCatch
a = 0
Деление на 0: java.lang.ArithmeticException: / by zero
После блока try/catch.
C:\>java MultiCatch TestArg
a = 1
Ошибка индекса массива: java.lang.ArrayIndexOutOfBoundsException:42
После блока try/catch.
```

Когда используются множественные операторы `catch`, важно помнить, что подклассы исключений должны следовать перед любыми их суперклассами. Это потому, что оператор `catch`, который использует суперкласс, будет перехватывать все исключения этого суперкласса плюс всех его подклассов. То есть подкласс исключения никогда не будет обработан, если вы попытаетесь его перехватить после его суперкласса. Более того, в Java недостижимый код является ошибкой. Например, рассмотрим следующую программу:

```
/* Эта программа содержит ошибку.
   Подкласс должен идти перед его суперклассом в
   последовательности операторов catch. В противном случае
   будет создан недоступный код, что приведет к ошибке при компиляции.
*/
class SuperSubCatch {
public static void main(String args[]) {
  try {
    int a = 0;
    int b = 42 / a;
  } catch(Exception e) {
    System.out.println("Общий перехват Exception.");
  }
  /* Этот catch никогда не будет достигнут, потому что
     ArithmeticException — это подкласс Exception. */
  catch(ArithmeticException e) { // Ошибка — недостижимый код
    System.out.println("Это никогда не выполнится.");
  }
}
}
```

Если вы попытаетесь скомпилировать эту программу, то получите сообщение об ошибке, говорящее о том, что второй оператор `catch` недостижим, потому что исключение уже перехвачено. Поскольку `ArithmeticException` — подкласс `Exception`, первый оператор `catch` обработает все ошибки, основанные на `Exception`, включая `ArithmeticException`. Это означает, что второй оператор `catch` не будет никогда выполнен. Чтобы исправить эту проблему, потребуется изменить порядок следования операторов `catch`.

Вложенные операторы `try`

Операторы `try` могут быть вложенными. То есть оператор `try` может находиться внутри блока другого `try`. Всякий раз, когда управление попадает в блок `try`, контекст этого исключения заталкивается в стек. Если вложенный оператор `try` не имеет обработчика `catch` для определенного исключения, стек "раскручивается" и проверяются на соответствие обработчики `catch` следующего (внешнего) блока `try`. Это продолжается до тех пор, пока не будет найден подходящий оператор `catch` либо пока не будут проверены все уровни вложенных `try`. Если подходящий оператор `catch` не будет найден, то исключение обработает система времени выполнения Java. Ниже приведен пример, в котором используются вложенные операторы `try`.

```
// Пример вложенных операторов try.
class NestTry {
public static void main(String args[]) {
  try {
     int a = args.length;
     /* Если не указаны параметры командной строки,
        следующий оператор сгенерирует
        исключение деления на ноль. */
     int b = 42 / a;
     System.out.println("a = " + a);
     try { // вложенный блок try
       /* Если используется один аргумент командной строки,
          то исключение деления на ноль
          будет сгенерировано следующим кодом. */
       if(a==1) a = a/(a-a); // деление на ноль
       /* Если используется два аргумента командной строки,
          то генерируется исключение выхода за пределы массива. */
       if(a==2) {
           int c[] = { 1 };
           c[42] = 99; // генерируется исключение выхода за пределы массива
       }
     } catch(ArrayIndexOutOfBoundsException e) {
       System.out.println("Индекс за пределами массива: " + e);
     }
  } catch(ArithmeticException e) {
     System.out.println("Деление на 0: " + e);
  }
}
}
```

Как видите, в этой программе один блок `try` вложен внутрь другого. Программа работает следующим образом. Когда вы запускаете ее без аргументов командной строки, внешним блоком `try` генерируется исключение деления на ноль. Запуск программы с одним аргументом вызывает генерацию исключения деления на ноль во вложенном блоке `try`.

Поскольку вложенный блок не обрабатывает это исключение, оно передается внешнему блоку try, который обрабатывает его. Если программе передается два аргумента командной строки, то генерируется исключение выхода индекса за границы массива во внутреннем блоке try. Вот примеры запуска этой программы, иллюстрирующие каждый случай:

```
C:\>java NestTry
Деление на 0: java.lang.ArithmeticException: / by zero
C:\>java NestTry One
a = 1
Деление на 0: 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, если в процессе выполняются вызовы методов. Например, вы можете в пределах блока try вызывать метод, а внутри этого метода иметь еще один блок try. В этом случае try в теле метода находится внутри внешнего блока try, который вызывает этот метод. Ниже представлена версия предыдущей программы с блоком try, перемещенным внутрь метода nesttry().

```
/* Операторы try могут быть неявно вложены в вызовах методов. */
class MethNestTry {
static void nesttry(int a) {
  try { // вложенный блок try
       /* Если используется один аргумент командной строки,
       то исключение деления на ноль
       будет сгенерировано следующим кодом. */
     if(a==1) a = a/(a-a); // деление на ноль
       /* Если используется два аргумента командной строки,
         то генерируется исключение выхода за пределы массива. */
     if(a==2) {
       int c[] = { 1 };
       c[42] = 99; // генерируется исключение выхода 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 параметры командной строки,
       следующий оператор сгенерирует
       исключение деления на ноль. */
     int b = 42 / a;
     System.out.println("a = " + a);
     nesttry(a);
  } catch(ArithmeticException e) {
     System.out.println("Деление на 0: " + e);
  }
}
}
```

Вывод этой программы идентичен предыдущему примеру.

throw

До сих пор мы перехватывали только исключения, которые возбуждала система времени выполнения Java. Однако существует возможность возбуждать исключения из ваших программ явным образом, используя оператор `throw`. Его общая форма показана ниже:

```
throw экземпляр_Throwable;
```

Здесь *экземпляр_Throwable* должен быть объектом типа `Throwable` либо подклассом `Throwable`. Примитивные типы — такие, как `int` или `char`, как и классы, отличные от `Throwable`, например `String` и `Object`, не могут быть использованы для исключений.

Существуют два способа получить объект `Throwable`: с использованием параметра в операторе `catch` либо созданием объекта оператором `new`.

Поток выполнения останавливается непосредственно после оператора `throw` — любые последующие операторы не выполняются. Обнаруживается ближайший закрытый блок `try`, имеющий оператор `catch` соответствующего исключению типа. Если соответствие найдено, управление передается этому оператору. Если же нет, проверяется следующий внешний блок `try` и т.д. Если не находится подходящего по типу оператора `catch`, то обработчик исключений по умолчанию прерывает программу и печатает трассировку стека.

Ниже приведен пример программы, создающей и возбуждающей исключение. Обработчик, который перехватывает его, повторно возбуждает его для внешнего обработчика.

```
// Демонстрация применения throw.
class ThrowDemo {
static void demoproc() {
  try {
     throw new NullPointerException("demo");
  } catch(NullPointerException e) {
     System.out.println("Перехвачено внутри demoproc.");
     throw e; // повторно возбудить исключение
  }
}
public static void main(String args[]) {
  try {
     demoproc();
  } catch(NullPointerException e) {
     System.out.println("Повторный перехват: " + e);
  }
}
}
```

Эта программа получает два шанса обработки одной и той же ошибки. Во-первых, `main()` устанавливает контекст исключения, затем вызывает `demoproc()`. Метод `demoproc()` устанавливает другой контекст обработки исключения и немедленно возбуждает новый экземпляр исключения `NullPointerException`, который перехватывается в следующей строке. Затем исключение возбуждается повторно. Ниже показан результирующий вывод.

```
Перехвачено внутри demoproc.
Повторный перехват: java.lang.NullPointerException: demo
```

Эта программа также демонстрирует, как создавать собственные объекты стандартных исключений Java. Обратите внимание на строку:

```
throw new NullPointerException("demo");
```

Здесь операция new используется для конструирования экземпляра NullPointerException. Многие их встроенных исключений времени выполнения Java имеют, по меньшей мере, два конструктора: один без параметров и один со строковым параметром. Когда применяется вторая форма, аргумент указывает строку, описывающую исключение. Эта строка отображается, когда объект используется в качестве аргумента print() или println(). Она также может быть получена вызовом метода getMessage(), который определен в Throwable.

throws

Если метод может породить исключение, которое он сам не обрабатывает, он должен специфицировать это поведение так, чтобы вызывающий его код мог позаботиться об этом исключении. Это делается добавлением к объявлению метода конструкции throws. Конструкция throws перечисляет типы исключений, которые метод может возбуждать. Это необходимо для всех исключений, кроме имеющих тип Error, RuntimeException либо их подклассов. Все остальные исключения, которые может возбуждать метод, должны быть объявлены в конструкции throws. Если этого не сделать, получится ошибка во время компиляции.

Вот общая форма объявления метода, которая включает оператор throws:

```
тип имя_метода(список_параметров) throws список_исключений
{
  // тело метода
}
```

Здесь *список_исключений* — это разделенный запятыми список исключений, которые метод может возбуждать.

Ниже представлен пример неправильной программы, пытающейся возбудить исключение, которое сама она не перехватывает. Поскольку в программе не указан оператор throws для отображения этого факта, такая программа не скомпилируется.

```
// Эта программа содержит ошибку и потому не компилируется.
class ThrowsDemo {
static void throwOne() {
  System.out.println("Внутри throwOne.");
  throw new IllegalAccessException("демо");
}
public static void main(String args[]) {
  throwOne();
}
}
```

Чтобы скомпилировать этот пример, нужно внести в него два изменения. Во-первых, вы должны объявить, что throwOne() возбуждает исключение IllegalAccessException. Во-вторых, main() должен определять блок try/catch, который перехватит это исключение.

Исправленный пример выглядит следующим образом:

```
// Теперь код корректен.
class ThrowsDemo {
static void throwOne() throws IllegalAccessException {
  System.out.println("Внутри throwOne.");
  throw new IllegalAccessException("demo");
}
```

```
public static void main(String args[]) {
  try {
    throwOne();
  } catch (IllegalAccessException e) {
     System.out.println("Перехвачено " + e);
  }
}
}
```

Вот результат, полученный при запуске этой программы:

```
Внутри throwOne
Перехвачено java.lang.IllegalAccessException: demo
```

finally

Когда исключение возбуждено, выполнение метода направляется по нелинейному пути, изменяющему нормальный поток управления внутри метода. В зависимости от того, как закодирован метод, существует даже возможность преждевременного возврата управления. В некоторых методах это может служить причиной серьезных проблем. Например, если метод при входе открывает файл и закрывает его при выходе, вероятно, вы не захотите, чтобы выполнение кода, закрывающего файл, было пропущено из-за применения механизма обработки исключений. Ключевое слово `finally` предназначено для того, чтобы справиться с такой ситуацией.

`finally` создает блок кода, который будет выполнен после завершения блока `try/catch`, но перед кодом, следующим за `try/catch`. Блок `finally` выполняется независимо от того, возбуждено исключение или нет. Если исключение возбуждено, блок `finally` выполняется, даже если ни один оператор `catch` этому исключению не соответствует. В любой момент, когда метод собирается возвратить управление вызывающему коду изнутри блока `try/catch` — из-за необработанного исключения, или явным применением оператора `return` — блок `finally` будет выполнен перед возвратом управления из метода. Это может быть удобно для закрытия файловых дескрипторов либо освобождения других ресурсов, которые были получены в начале метода и должны быть освобождены перед возвратом. Оператор `finally` необязателен. Однако каждый оператор `try` требует наличия, по крайней мере, одного оператора `catch` или `finally`. Ниже приведен пример программы, которая показывает три метода, возвращающих управление разными способами, но ни один из них не пропускает выполнения блока `finally`.

```
class FinallyDemo {
// Возбуждает исключение из метода.
static void procA() {
     try {
         System.out.println("внутри procA");
         throw new RuntimeException("демо");
     } finally {
         System.out.println("блок finally procA");
     }
}
// Возврат управления в блоке try.
static void procB() {
     try {
         System.out.println("внутри procB");
         return;
```

```
        } finally {
            System.out.println("блок finally procB");
        }
    }

    // Нормальное выполнение блока try.
    static void procC() {
        try {
            System.out.println("внутри procC");
        } finally {
            System.out.println("блок finally procC");
        }
    }

    public static void main(String args[]) {
        try {
            procA();
        } catch (Exception e) {
            System.out.println("Исключение перехвачено");
        }
        procB();
        procC();
    }
    }
```

В этом примере `procA()` преждевременно прерывает выполнение в блоке `try`, возбуждая исключение. Блок `finally` все равно выполняется. В `procB()` возврат управления осуществляется в блоке `try` оператором `return`. Блок `finally` выполняется перед возвратом из `procB()`. В `procC()` блок `try` выполняется нормально, без ошибок. Однако блок `finally` выполняется все равно.

Помните! *Если блок `finally` ассоциируется с `try`, то `finally` будет выполнен по завершении `try`.*

Вот результат, сгенерированный предыдущей программой:

```
внутри procA
блок finally procA
Исключение перехвачено
внутри procB
блок finally procB
внутри procC
блок finally procC
```

Встроенные исключения Java

Внутри стандартного пакета `java.lang` определено несколько классов исключений. Некоторые из них использовались в предшествующих примерах. Наиболее общие из этих исключений являются подклассами стандартного типа `RuntimeException`. Как уже объяснялось ранее, эти исключения не нужно включать в список `throws` метода — они называются *непроверяемыми исключениями*, поскольку компилятор не проверяет факт обработки или возбуждения методом таких исключений. Непроверяемые исключения, определенные в `java.lang`, описаны в табл. 10.1. В табл. 10.2 перечислены те из исключений, определенных в `java.lang`, которые должны быть включены в списки `throws`

методов, которые могут их генерировать и не обрабатывают самостоятельно. Они называются *проверяемыми исключениями*. В Java также определено несколько других типов исключений, имеющих отношение к библиотекам классов.

Таблица 10.1. Непроверяемые подклассы `RunTimeException`, определенные в `java.lang`

Исключение	Описание
`ArithmeticException`	Арифметическая ошибка, такая как деление на ноль.
`ArrayIndexOutOfBoundsException`	Выход индекса за границу массива.
`ArrayStoreException`	Присваивание элементу массива объекта несовместимого типа.
`ClassCastException`	Неверное приведение.
`EnumConstantNotPresentException`	Попытка использования неопределенного значения перечисления.
`IllegalArgumentException`	Неверный аргумент использован при вызове метода.
`IllegalMonitorStateException`	Неверная операция мониторинга, такая как ожидание незаблокированного потока.
`IllegalStateException`	Окружение или приложение в некорректном состоянии.
`IllegalThreadStateException`	Запрошенная операция несовместима с текущим состоянием потока.
`IndexOutOfBoundsException`	Некоторый тип индекса вышел за допустимые пределы.
`NegativeArraySizeException`	Создан массив отрицательного размера.
`NullPointerException`	Неверное использование нулевой ссылки.
`NumberFormatException`	Неверное преобразование строки в числовой формат.
`SecurityException`	Попытка нарушения безопасности.
`StringIndexOutOfBounds`	Попытка использования индекса за пределами строки.
`TypeNotPresentException`	Тип не найден (добавлено в J2SE 5).
`UnsupportedOperationException`	Обнаружена неподдерживаемая операция.

Таблица 10.2. Проверяемые исключения, определенные в `java.lang`

Исключение	Описание
`ClassNotFoundException`	Класс не найден.
`CloneNotSupportedException`	Попытка клонировать объект, который не реализует интерфейс `Cloneable`.
`IllegalAccessException`	Доступ к классу не разрешен.
`InstantiationException`	Попытка создать объект абстрактного класса или интерфейса.
`InterruptedException`	Один поток прерван другим потоком.
`NoSuchFieldException`	Запрошенное поле не существует.
`NoSuchMethodException`	Запрошенный метод не существует.

Создание собственных подклассов исключений

Хотя встроенные исключения Java обрабатывают большинство частых ошибок, вероятно, вам потребуется создать ваши собственные типы исключений для обработки ситуаций, специфичных для ваших приложений. Это достаточно просто сделать: просто определите подкласс `Exception` (который, разумеется, является подклассом `Throwable`). Ваши подклассы не обязаны реализовывать что-либо — важно само их присутствие в системе типов, которое позволит использовать их как исключения.

Класс `Exception` не определяет никаких собственных методов. Естественно, он наследует методы, представленные в `Throwable`. Таким образом, всем исключениям, включая те, что вы создадите сами, доступны методы, определенные в `Throwable`. Они все перечислены в табл. 10.3.

Таблица 10.3. Методы, определенные в `Throwable`

Метод	Описание
`Throwable fillInStackTrace()`	Возвращает объект `Throwable`, содержащий полную трассировку стека. Этот объект может быть возбужден повторно.
`Throwable getCause()`	Возвращает исключение, лежащее под текущим исключением. Если такого нет, возвращается `null`.
`String getLocalizedMessage()`	Возвращает локализованное описание исключения.
`String getMessage()`	Возвращает описание исключения
`StackTraceElement[] getStackTrace()`	Возвращает массив, содержащий трассировку стека и состоящий из элементов типа `StackTraceElement`. Метод в верхушке стека — это метод, который был вызван непосредственно перед тем, как было возбуждено исключение. Этот метод содержится в первом элементе массива. Класс `StackTraceElement` дает вашей программе доступ к информации о каждом элементе в трассировке, такой как имя его метода.
`Throwable initCause(` `Throwable` *исключение*`)`	Ассоциирует *исключение* с вызывающим исключением, как причиной этого вызывающего исключения. Возвращает ссылку на исключение.
`void printStackTrace()`	Отображает трассировку стека.
`void printStackTrace(PrintStream` *поток*`)`	Посылает трассировку стека в заданный поток.
`void printStackTrace(PrintWriter` *поток*`)`	Посылает трассировку стека в заданный поток.
`void setStackTrace(` `StackTraceElement` *элементы*`[])`	Посылает трассировку стека в элементы, переданные в *элементы*. Этот метод предназначен для специализированных приложений, а не для нормального применения.
`String toString()`	Возвращает объект `String`, содержащий описание исключения. Этот метод вызывается из `println()` при выводе объекта `Throwable`.

Вы можете также переопределить один или более из этих методов в собственных классах исключений.

`Exception` определяет четыре конструктора. Два были добавлены в JDK 1.4 для поддержки цепочек исключений, о которых будет сказано в следующем разделе. Другие два показаны здесь:

```
Exception()
Exception(String msg)
```

Первая форма создает исключение, не имеющее описания. Вторая — позволяет специфицировать описание исключения.

Хотя указание такого описания часто полезно при создании исключения, иногда все же лучше переопределить `toString()`, и вот почему. Версия `toString()`, определенная `Throwable` (от которого наследуется `Exception`) сначала отображает имя исключения, за которым следует двоеточие, а после него — ваше описание. Переопределив `toString()`, вы можете предотвратить отображение имени исключения и двоеточия. Это проясняет вывод, что весьма желательно в некоторых случаях.

В следующем примере объявляется новый подкласс `Exception`, который затем используется для сигнализации об ошибочной ситуации в методе. Он переопределяет метод `toString()`, позволяя отобразить тщательно настроенное описание исключения.

```
// Эта программа создает пользовательский 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 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 " + e);
  }
}
}
```

В этом примере определен подкласс `Exception` с именем `MyException`. Этот подкласс достаточно прост: он имеет только конструктор и перегруженный метод `toString()`, отображающий значение исключения. Класс `ExceptionDemo` определяет метод под названием `compute()`, который возбуждает объект `MyException`. Это исключение возбуждается, когда целочисленный параметр `compute()` принимает значение больше 10.

Метод `main()` устанавливает обработчик исключений `MyException`, затем вызывает `compute()` с правильным параметром (меньше 10), и с неправильным, чтобы продемонстрировать оба пути выполнения кода. Ниже показан результат.

```
Вызван compute(1)
Нормальное завершение
Вызван compute(20)
Перехвачено MyException[20]
```

Сцепленные исключения

Начиная с J2SE 1.4, в подсистему исключений было добавлено новое средство: *сцепленные исключения* (chained exceptions). Это средство позволяет ассоциировать с одним исключением другое исключение. Второе исключение описывает причину появления первого. Например, представьте ситуацию, когда метод возбуждает исключение `ArithmeticException`, поскольку была предпринята попытка деления на ноль. Однако реальная причина проблемы заключается в ошибке ввода-вывода, что приводит к неправильному делению. И хотя метод должен возбуждать `ArithmeticException`, так как произошла именно эта ошибка, вы можете также позволить вызывающему коду узнать о том, что в основе лежит ошибка ввода-вывода. Сцепленные исключения позволяют справиться с этой, а также с любой другой ситуацией, в которой присутствуют уровни исключений.

Чтобы разрешить сцепленные исключения, были добавлены два конструктора и два метода к `Throwable`.

Ниже показаны конструкторы.

```
Throwable(Throwable causeExc)
Throwable(String msg, Throwable causeExc)
```

В первой форме *`causeExc`* — это исключение, послужившее причиной текущего исключения. Если такого нет, возвращается `null`. Метод `initCause()` ассоциирует *`causeExc`* с вызывающим исключением и возвращает ссылку на исключение. То есть вы можете вызвать `initCause()` только однажды для каждого объекта-исключения. Более того, если исключение-причина было установлено конструктором, то вы не можете установить его заново методом `initCause()`.

Вообще говоря, `initCause()` используется для установки причины унаследованного класса исключения, которое не поддерживает эти два дополнительных конструктора.

Вот пример, демонстрирующий применение механизма сцепленных исключений:

```
// Демонстрация сцепленных исключений.
class ChainExcDemo {
static void demoproc() {
   // создать исключение
   NullPointerException e = new NullPointerException("верхний уровень");
   // добавить причину
   e.initCause(new ArithmeticException("причина"));
   throw e;
}

public static void main(String args[]) {
  try {
     demoproc();
  } catch(NullPointerException e) {
```

```
    // отобразить исключение верхнего уровня
    System.out.println("Перехвачено: " + e);

    // отобразить исключение-причину
    System.out.println("Исходная причина: " + e.getCause());
  }
}
}
```

Эта программа создает следующий вывод:

```
Перехвачено: java.lang.NullPointerException: верхний уровень
Исходная причина: java.lang.ArithmeticException: причина
```

В этом примере исключением верхнего уровня является `NullPointerException`. К нему добавлено исключение-причина — `ArithmeticException`. Когда исключение возбуждается из `demoproc()`, оно перехватывается в `main()`. Затем исключение верхнего уровня отображается, а за ним следует исключение, лежащее в основе, которое извлекается методом `getCause()`.

Сцепленные исключения могут вкладываться на любую глубину. То есть исключение-причина может иметь собственную причину. Но имейте в виду, что слишком длинные цепочки сцепленных исключений, скорее всего, свидетельствуют о плохом дизайне.

Сцепленные исключения не являются вещью, совершенно необходимой в каждой программе. Однако в случаях, когда информация об исключении-причине таки нужна, они представляют собой элегантное решение.

Использование исключений

Обработка исключений представляет собой мощный механизм для управления сложными программами, которые имеют много динамических характеристик времени выполнения. Важно думать о `try`, `throws` и `catch`, как о чистом способе обработки ошибок и необычных краевых условиях в вашей программной логике. В отличие от ряда других языков, в которых для индикации сбоев используются коды ошибок, в Java применяются исключения. То есть, когда метод может завершиться сбоем, он возбуждает исключение. Это более чистый способ справиться со сбойными ситуациями.

Одно последнее замечание: операторы управления исключениями Java не должны рассматриваться как общий способ нелокального ветвления. Если вы будете это делать, это только запутает ваш код и усложнит его сопровождение.

11 ГЛАВА

Многопоточное программирование

В отличие от многих других языков программирования, Java предлагает встроенную поддержку *многопоточного программирования*. Многопоточная программа содержит две или более частей, которые могут выполняться одновременно. Каждая часть такой программы называется *потоком* (thread), и каждый поток задает отдельный путь выполнения. То есть, многопоточность — это специализированная форма многозадачности.

Вы почти наверняка знакомы с многозадачностью, поскольку она поддерживается практически всеми современными операционными системами. Однако существуют два отдельных типа многозадачности: многозадачность, основанная на процессах, и многозадачность, основанная на потоках. Важно понимать разницу между ними. Большинству читателей многозадачность, основанная на процессах, является более знакомой формой. *Процесс* по сути своей — это выполняющаяся программа. То есть многозадачность, основанная на процессах, представляет собой средство, которое позволяет вашему компьютеру одновременно выполнять две или более программ. Так, например, процессная многозадачность позволяет запускать компилятор Java в то самое время, когда вы используете текстовый редактор. В многозадачности, основанной на процессах, программа представляет собой наименьший элемент кода, которым может управлять планировщик операционной системы.

В среде *поточной многозадачности* наименьшим элементом управляемого кода является поток. Это означает, что одна программа может выполнять две или более задач одновременно. Например, текстовый редактор может форматировать текст в то же время, когда выполняется его печать — до тех пор, пока эти два действия выполняются двумя отдельными потоками. То есть многозадачность на основе процессов имеет дело с "картиной в целом", а потоковая многозадачность справляется с деталями.

Многозадачные потоки требуют меньше накладных расходов, чем многозадачные процессы. Процессы — это тяжеловесные задачи, каждая из которых требует своего собственного адресного пространства. Межпроцессные коммуникации дорогостоящи и ограничены. Переключение контекста от одного процесса к другому также обходится дорого. С другой стороны, потоки являются облегченными. Они разделяют одно и тоже адресное пространство и совместно используют один и тот же тяжеловесный процесс.

Коммуникации между потоками являются экономными, а переключения контекста между потоками характеризуется низкой стоимостью. Хотя Java-программы используются в средах процессной многозадачности, многозадачность, основанная на процессах, средствами Java не управляется. А вот многопоточная многозадачность средствами Java управляется.

Многопоточность позволяет вам писать очень эффективные программы, которые по максимуму используют центральный процессор, поскольку время ожидания может быть сведено к минимуму. Это особенно важно для интерактивных сетевых сред, в которых работает Java, так как в них наличие ожидания и простоев — обычное явление. Например, скорость передачи данных по сети намного ниже, чем скорость, с которой компьютер может их обрабатывать. Даже ресурсы локальной файловой системы читаются и пишутся намного медленнее, чем темп их обработки в процессоре. И, конечно, ввод пользователя намного медленнее, чем компьютер. В однопоточных средах ваша программа вынуждена ожидать окончания таких задач, прежде чем переходить к следующей — даже если центральный процессор бо́льшую часть времени простаивает. Многопоточность позволяет получить доступ к этому времени ожидания и использовать его рациональным образом.

Если вы программировали для таких операционных систем, как Windows, это значит, что вы уже знакомы с многопоточным программированием. Однако тот факт, что Java управляет потоками, делает многопоточность особенно удобной, поскольку многие детали подконтрольны вам как программисту.

Модель потоков Java

Система времени выполнения Java зависит от потоков во многих отношениях, и все библиотеки классов спроектированы с учетом многопоточности. Фактически Java использует потоки для того, чтобы обеспечить асинхронность всей среде выполнения. Это позволяет снизить неэффективность за счет предотвращения холостой растраты циклов центрального процессора.

Значение многопоточной среды лучше понимается при сравнении с ее противоположностью. Однопоточные системы используют подход, называемый *циклом событий с опросом*. В этой модели единственный поток управления выполняется в бесконечном цикле, опрашивая единственную очередь событий, чтобы принять решение о том, что делать дальше. Как только этот механизм опроса возвращает, скажем, сигнал о том, что сетевой файл готов к чтению, цикл событий передает управление соответствующему обработчику событий. До тех пор, пока тот не вернет управление, в системе ничего не может произойти. Это расходует время процессора. Это также может привести к тому, что одна часть программы будет доминировать над другими и не давать возможности обрабатывать любые другие события. Вообще говоря, в однопоточном окружении, когда поток блокируется (то есть приостанавливает выполнение) по причине ожидания некоторого ресурса, выполнение всей программы приостанавливается.

Выгода от многопоточности состоит в том, что основной механизм циклического опроса исключается. Один поток может быть приостановлен без остановки других частей программы. Например, время ожидания при чтении данных из сети либо ожидание пользовательского ввода может быть утилизировано где угодно. Многопоточность позволяет циклам анимации “засыпать” на секунду между показом соседних кадров, не приостанавливая работы всей системы. Когда поток блокируется в программе Java, то останавливается только один-единственный заблокированный поток. Все остальные потоки продолжают выполняться.

Потоки существуют в нескольких состояниях. Поток может *выполняться*. Он может быть *готов к выполнению*, как только получит время центрального процессора. Работающий поток может быть *приостановлен*, что временно прекращает его активность. Выполнение приостановленного потока может быть *возобновлено*, позволяя ему продолжить работу с того места, где он был приостановлен. Поток может быть *заблокирован*, когда ожидает какого-то ресурса. В любой момент поток может быть *прерван*, что немедленно останавливает его выполнение. Однажды прерванный поток уже не может быть возобновлен.

Приоритеты потоков

Java присваивает каждому потоку приоритет, который определяет поведение данного потока по отношению к другим. Приоритеты потоков задаются целыми числами, определяющими относительный приоритет одного потока по сравнению к другими. Значение приоритета само по себе никакого смысла не имеет — более высокоприоритетный поток не выполняется быстрее, чем низкоприоритетный, когда он является единственным исполняемым потоком в данный момент. Вместо этого приоритет потока используется для принятия решения при переключении от одного выполняющегося потока к другому. Это называется *переключением контекста*. Правила, которые определяют, когда должно происходить переключение контекста, достаточно просты.

- *Поток может добровольно уступить управление*. Это делается явным уступанием очереди выполнения, приостановкой или блокированием ожидания ввода-вывода. При таком сценарии все прочие потоки проверяются, и ресурсы процессора передаются потоку с максимальным приоритетом, который готов к выполнению.
- *Поток может быть прерван другим, более приоритетным потоком*. В этом случае низкоприоритетный поток, который не занимает процессор, просто приостанавливается высокоприоритетным потоком, независимо от того, что он делает. В основном, высокоприоритетный поток выполняется, как только он этого "захочет". Это называется *вытесняющей многозадачностью* (или *многозадачностью с приоритетами*).

В случае, когда два потока, имеющие одинаковый приоритет, претендуют на цикл процессора, ситуация усложняется. Для таких операционных систем, как Windows, потоки с одинаковым приоритетом разделяют время в циклическом режиме. Для операционных систем других типов потоки с одинаковым приоритетом должны принудительно передавать управление своим "родственникам". Если они этого не делают, другие потоки не запускаются.

Внимание! *Из-за разницы в способах переключения операционными системами потоковых контекстов могут возникать проблемы переносимости.*

Синхронизация

Поскольку многопоточность дает вашим программам возможность асинхронного поведения, должен существовать способ обеспечить синхронизацию, когда в этом возникает необходимость. Например, если вы хотите, чтобы два потока взаимодействовали и разделяли сложную структуру данных, такую как связный список, то вы нуждаетесь в каком-то способе исключения конфликтов между ними. То есть вы должны предотвратить запись данных в одном потоке, когда другой занимается их чтением. Для этой цели в Java реализован элегантный трюк из старой модели межпроцессной синхронизации, а именно — *монитор*. Монитор — это управляющий механизм, впервые реализованный

Чарльзом Энтони Ричардом Хоаром. Вы можете воспринимать монитор как очень маленький ящик, который принимает только один поток в единицу времени. Как только поток вошел в монитор, все другие потоки должны ждать, пока тот не покинет его. Таким образом, монитор может быть использован для защиты разделяемых ресурсов от одновременного использования более чем одним потоком.

Большинство многопоточных систем применяют мониторы как объекты, которые ваша программа может получить и которыми она может манипулировать. Java предлагает более чистое решение. Не существует отдельного класса монитора вроде "Monitor". Вместо этого каждый объект имеет свой собственный неявный монитор, вход в который осуществляется автоматически, когда вызывается синхронизированный метод объекта. Когда поток находится внутри синхронизированного метода, ни один другой поток не может вызвать никакого синхронизированного метода этого объекта. Это позволяет вам писать очень ясный и краткий многопоточный код, поскольку поддержка синхронизации встроена в язык.

Обмен сообщениями

После того, как вы разделите вашу программу на отдельные потоки, вам нужно определить, как они будут общаться друг с другом. При программировании на большинстве других языков для установки взаимодействия между потоками вы должны зависеть от операционной системы. То есть, конечно же, появляются накладные расходы. В отличие от них, Java предоставляет ясный и экономичный способ общения двух или более потоков между собой — посредством вызова предопределенных методов, которыми обладают объекты. Система сообщений Java позволяет потоку войти в синхронизированный метод объекта и затем ожидать, пока какой-то другой поток явно не уведомит его о прибытии.

Класс `Thread` и интерфейс `Runnable`

Многопоточная система Java встроена в класс `Thread`, его методы и дополняющий его интерфейс `Runnable`. `Thread` инкапсулирует поток выполнения. Поскольку вы не можете напрямую обратиться к нематериальному состоянию работающего потока, вы имеете дело с его заместителем (proxy) — экземпляром класса `Thread`, который породил его. Чтобы создать новый поток, ваша программа должна либо расширить `Thread`, либо реализовать интерфейс `Runnable`.

Класс `Thread` определяет несколько методов, которые помогают управлять потоками. Некоторые из них, которые будут упомянуты в настоящей главе, перечислены в табл. 11.1.

Таблица 11.1. Методы управления потоками класса `Thread`

Метод	Назначение
`getName`	Получить имя потока.
`getPriority`	Получить приоритет потока.
`isAlive`	Определить, выполняется ли поток.
`join`	Ожидать завершения потока.
`run`	Входная точка потока.
`sleep`	Приостановить выполнение потока на заданное время.
`start`	Запустить поток вызовом его метода `run`.

До сих пор все примеры в нашей книге использовали единственный поток управления. В остатке этой главы объясняется, как применять `Thread` и `Runnable` для создания и управления потоками, начиная с потока, который есть в каждой программе Java — главного.

Главный поток

Когда Java-программа стартует, немедленно начинает выполняться один поток. Обычно его называют *главным потоком* программы, потому что это тот поток, который запускается вместе с вашей программой. Главный поток важен по двум причинам.

- Это поток, от которого порождаются все "дочерние" потоки.
- Часто он должен быть последним потоком, завершающим выполнение, так как он предпринимает различные завершающие действия.

Несмотря на то что главный поток создается автоматически при запуске программы, им можно управлять через объект `Thread`. Чтобы делать это, вы должны получить ссылку на него вызовом метода `currentThread()`, который является общедоступным статическим (`public static`) методом класса `Thread`. Его общая форма выглядит следующим образом:

```
static Thread currentThread()
```

Этот метод возвращает ссылку на поток, из которого он был вызван. Получив ссылку на главный поток, вы можете управлять им точно так же, как любым другим.

Рассмотрим следующий пример:

```
// Управление главным потоком.
class CurrentThreadDemo {
public static void main(String args[]) {
  Thread t = Thread.currentThread();
  System.out.println("Текущий поток: " + t);
  // изменить имя потока
  t.setName("Мой Thread");
  System.out.println("После изменения имени: " + t);
  try {
     for(int n = 5; n > 0; n--) {
       System.out.println(n);
       Thread.sleep(1000);
     }
  } catch (InterruptedException e) {
     System.out.println("Главный поток прерван");
  }
}
}
```

В этой программе ссылка на текущий поток (в данном случае — главный) получается вызовом `currentThread()`, и эта ссылка сохраняется в локальной переменной `t`. Далее программа отображает информацию о потоке. Программа вызывает `setName()` для изменения внутреннего имени потока. После этого информация о потоке отображается заново. Далее в цикле печатается обратный отсчет с задержкой на 1 секунду после каждой строки. Пауза организуется вызовом метода `sleep()`. Аргумент `sleep()` задает период задержки в миллисекундах. Обратите внимание на блок `try/catch` вокруг цикла. Метод

sleep() в Thread может возбудить InterruptException. Это может произойти, если некоторый другой поток захочет прервать выполнение этого спящего потока. Этот пример просто печатает сообщение, если поток прерывается. В реальных программах вы будете обрабатывать подобную ситуацию иначе. Ниже показан вывод, генерируемый этой программой.

```
Текущий поток: Thread[main,5,main]
После изменения имени: Thread[My Thread,5,main]
5
4
3
2
1
```

Обратите внимание, что вывод генерируется, когда t используется в качестве аргумента для println(). Он отображает по порядку имя потока, его приоритет и имя его группы. По умолчанию имя главного потока — main. Его приоритет равен 5, что является значением по умолчанию, а main — также имя группы потоков, к которой относится данный. *Группа потоков* — это структура данных, которая управляет состоянием набором потоков в целом. После того как имя потока изменено, t распечатывается вновь. На этот раз отображается новое имя потока.

Давайте поближе взглянем на методы, определенные в Thread, которые используются в программе. Метод sleep() заставляет поток, из которого он был вызван, приостановить выполнение на указанное количество миллисекунд. Его общая форма выглядит так:

```
static void sleep(long миллисекунды) throws InterruptedException
```

Количество миллисекунд, на которое нужно приостановить выполнение, передается в параметре *миллисекунды*. Этот метод может возбуждать исключение InterruptedException.

Метод sleep() имеет также вторую форму, показанную ниже, который позволяет задать период в миллисекундах и наносекундах:

```
static void sleep(long миллисекунды, long наносекунды) throws InterruptedException
```

Вторая форма может применяться только в средах, которые предусматривают задание временных периодов в наносекундах.

Как показано в предыдущей программе, вы можете установить имя потока, используя setName(). Получить имя потока можно вызовом getName() (эта процедура в программе не показана). Эти методы являются членами класса Thread и объявлены следующим образом:

```
final void setName(String имя_потока)
final String getName()
```

Здесь *имя_потока* указывает имя потока.

Создание потока

В наиболее общем смысле вы создаете поток, реализуя объект класса Thread. В Java определены два способа, какими это можно сделать.

- Реализуя интерфейс Runnable.
- Расширяя класс Thread.

В следующих разделах рассматриваются эти способы по очереди.

Реализация Runnable

Самый простой способ создания потока — это объявление класса, реализующего интерфейс Runnable. Runnable абстрагирует единицу исполняемого кода. Вы можете конструировать поток из любого объекта, реализующего интерфейс Runnable. Чтобы реализовать Runnable, класс должен объявить единственный метод run():

```
public void run()
```

Внутри run() вы определяете код, который, собственно, составляет новый поток. Важно понимать, что run() может вызывать другие методы, использовать другие классы, объявлять переменные — точно так же, как это делает главный поток. Единственным отличием является то, что run() устанавливает точку входа для другого, параллельного потока внутри вашей программы. Этот поток завершится, когда run() вернет управление.

После того как будет объявлен класс, реализующий интерфейс Runnable, вы создадите объект типа Thread из этого класса. В Thread определено несколько конструкторов. Тот, который должен использоваться в данном случае, выглядит следующим образом:

```
Thread(Runnable объект_потока, String имя_потока)
```

В этом конструкторе *объект_потока* — это экземпляр класса, реализующего интерфейс Runnable. Он определяет, где начнется выполнение потока. Имя нового потока передается в *имя_потока*.

После того, как новый поток будет создан, он не запускается до тех пор, пока вы не вызовете метод start(), объявленный в классе Thread. По сути, start() выполняет вызов run(). Метод start() показан ниже:

```
void start()
```

Рассмотрим пример, создающий новый поток и запускающий его выполнение:

```
// Создание второго потока.
class NewThread implements Runnable {
Thread t;
NewThread() {
  // Создать новый, второй поток
  t = new Thread(this, "Демонстрационный поток");
  System.out.println("Дочерний поток создан: " + t);
  t.start(); // Запустить поток
}
// Точка входа второго потока.
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.");
}
}
```

Внутри конструктора `NewThread` в следующем операторе создается новый объект `Thread`:

```
t = new Thread(this, "Демонстрационный поток");
```

Передача `this` в первом аргументе означает, что вы хотите, чтобы новый поток вызвал `run()` метод объекта `this`. Далее вызывается `start()`, чем запускается выполнение потока, начиная с метода `run()`. Это запускает цикл `for` дочернего потока. После вызова `start()` конструктор `NewThread` возвращает управление `main()`. Когда главный поток продолжает свою работу, он входит в свой цикл `for`. После этого оба потока выполняются параллельно, разделяя ресурсы центрального процессора, вплоть до завершения своих циклов. Вывод, генерируемый этой программой, показан ниже (ваш вывод может варьироваться, в зависимости от скорости процессора и загрузки).

```
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 уже упоминалось ранее, в многопоточной программе часто главный поток должен завершать выполнение последним. Фактически, для некоторых старых виртуальных машин Java (JVM), если главный поток завершается до завершения дочерних потоков, то исполняющая система Java может “зависнуть”. Предыдущая программа гарантирует, что главный поток завершится последним, поскольку главный поток “спит” 1000 миллисекунд между итерациями цикла, а дочерний поток “спит” только 500 миллисекунд. Это заставляет дочерний поток завершиться раньше главного. Но далее вы узнаете лучший способ ожидания завершения потоков.

Расширение Thread

Второй способ создания потока — это объявить класс, расширяющий Thread, а затем создать экземпляр этого класса. Расширяющий класс обязан переопределить метод run(), который является точкой входа для нового потока. Он также должен вызвать start() для запуска выполнения нового потока. Ниже приведен пример предыдущей программы, переписанной с использованием расширения Thread.

```
// Создание второго потока расширением Thread
class NewThread extends Thread {
NewThread() {
  // Создать новый второй поток
  super("Демонстрационный поток");
  System.out.println("Дочерний поток: " + this);
  start(); // Запустить поток
}

// Точка входа второго потока.
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
  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 программа генерирует точно такой же вывод, что и предыдущая версия. Как вы можете видеть, дочерний поток создается при конструировании объекта NewThread, который наследуется от Thread.

Обратите внимание на super() внутри NewThread. Он вызывает следующую форму конструктора Thread:

```
public Thread(String имя_потока)
```

Здесь *имя_потока* указывает имя потока.

Выбор подхода

В данный момент вы можете спросить, почему Java предлагает два способа создания дочерних потоков, и какой из этих подходов лучше. Ответы на эти вопросы взаимосвязаны. Класс `Thread` определяет несколько методов, которые могут быть переопределены в классах-наследниках. Из этих методов только один *должен* быть переопределен в обязательном порядке — это метод `run()`. То есть, конечно, этот же метод нужен, когда вы реализуете интерфейс `Runnable`. Многие программисты Java считают, что классы следует расширять только в случаях, когда они должны быть усовершенствованы или некоторым образом модифицированы. Поэтому если вы не переопределяете никаких других методов `Thread`, то вероятно, лучше просто реализовать интерфейс `Runnable`. Конечно, все остается на ваше усмотрение. Тем не менее, в оставшейся части настоящей главы мы будем создавать потоки, используя классы, реализующие интерфейс `Runnable`.

Создание множества потоков

До сих пор вы использовали только два потока: главный и один дочерний. Однако ваша программа может порождать столько потоков, сколько необходимо. Например, в следующей программе создаются три дочерних потока.

```
// Создание множества потоков.
class NewThread implements Runnable {
String name; // имя потока
Thread t;
NewThread(String threadname) {
  name = threadname;
  t = new Thread(this, name);
  System.out.println("Новый поток: " + t);
  t.start(); // запустить поток
}
// Входная точка потока.
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"); // запуск потоков
  new NewThread("Два");
  new NewThread("Три");
  try {
     // ожидание завершения других потоков
     Thread.sleep(10000);
  } catch (InterruptedException e) {
     System.out.println("Главный поток прерван");
  }
```

```
  System.out.println("Главный поток завершен.");
}
}
```

Вывод этой программы показан ниже:

```
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, будучи запущенными, все три дочерних потока разделяют ресурс центрального процессора. Обратите внимание на вызов `sleep(10000)` в `main()`. Это заставляет главный поток "уснуть" на 10 секунд и гарантирует, что он будет завершен последним.

Использование `isAlive()` и `join()`

Как упоминалось, часто необходимо, чтобы главный поток завершался последним. В предыдущих примерах это обеспечивается вызовом `sleep()` из `main()` с задержкой, достаточной для того, чтобы гарантировать, что все дочерние потоки завершатся раньше главного. Однако это неудовлетворительное решение, которое вызывает серьезный вопрос: как один поток может знать о том, что другой завершился? К счастью, `Thread` предлагает средство, которое дает ответ на этот вопрос.

Существуют два способа определить, что поток был завершен. Во-первых, вы можете вызвать метод `isAlive()` для этого потока. Этот метод определен в классе `Thread` и его общая форма такова:

```
final Boolean isAlive()
```

Метод `isAlive()` возвращает `true`, если поток, для которого он вызван, еще выполняется. В противном случае он возвращает `false`.

В то время как `isAlive()` применяется изредка, существует метод, который вы будете использовать чаще, чтобы дождаться завершения потока, а именно — `join()`, показанный ниже:

```
final void join() throws InetrruptedException
```

Этот метод ожидает завершения потока, для которого он вызван. Его имя отражает концепцию, что вызывающий поток ожидает, когда указанный поток присоединиться к нему. Дополнительные формы `join()` позволяют указывать максимальный период времени, которое вы будете ожидать завершения указанного потока.

Ниже приведена усовершенствованная версия предыдущего примера, использующая метод `join()` для обеспечения того, чтобы главный поток завершился последним. Здесь также демонстрируется применение метода `isAlive()`.

```
// Применение join() для ожидания завершения потоков.
class NewThread implements Runnable {
String name;   // имя потока
Thread t;
NewThread(String threadname) {
  name = threadname;
  t = new Thread(this, name);
  System.out.println("Новый поток: " + t);
  t.start();   // Запуск потока
}
// Входная точка потока.
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 ob1 = new NewThread("Одни");
  NewThread ob2 = new NewThread("Два");
  NewThread ob3 = new NewThread("Три");
  System.out.println("Поток Один запущен: "
                     + ob1.t.isAlive());
  System.out.println("Поток Два запущен: "
                     + ob2.t.isAlive());
  System.out.println("Поток Три запущен: "
                     + ob3.t.isAlive());
  // ожидать завершения потоков
  try {
     System.out.println("Ожидание завершения потоков.");
     ob1.t.join();
     ob2.t.join();
     ob3.t.join();
  } catch (InterruptedException e) {
     System.out.println("Главный поток прерван");
  }
  System.out.println("Поток Один запущен: "
                     + ob1.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.");
  }
  }
```

Пример вывода этой программы показан ниже (ваш вывод может отличаться в зависимости от скорости и загрузки процессора).

```
Новый поток: Thread[Одни,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, после того как вызовы `join()` вернут управление, потоки прекращают работу.

Приоритеты потоков

Приоритеты потоков используются планировщиком потоков для принятия решений о том, когда каждому из потоков будет разрешено работать. Теоретически высокоприоритетные потоки получают больше времени процессора, чем низкоприоритетные. Практически объем времени процессора, который получает поток, часто зависит от нескольких факторов помимо его приоритета. (Например, то, как операционная система реализует многозадачность, может влиять на относительную доступность времени процессора). Высокоприоритетный поток может также выгружать низкоприоритетный. Например, когда низкоприоритетный поток работает, а высокоприоритетный собирается продолжить свою прерванную работу (в связи с приостановкой или ожиданием завершения операции ввода-вывода), то последний выгружает низкоприоритетный поток.

Теоретически потоки с равным приоритетом должны получать равный доступ к центральному процессору. Но вы должны быть осторожны. Помните, что Java спроектирована для работы в широком спектре сред. Некоторые из этих сред реализуют многозадачность принципиально отлично от других. В целях безопасности потоки, которые разделяют один и тот же приоритет, должны получать управление в равной степени. Это гарантирует, что все потоки получат возможность выполняться в среде операционных систем с не вытесняющей многозадачностью. На практике, даже в средах с не вытесняющей многозадачностью большинство потоков все-таки имеют шанс выполняться, поскольку большинство потоков неизбежно сталкиваются с блокирующими ситуациями, такими как ожидание ввода-вывода. Когда подобное случается, заблокированный поток приостанавливается, и остальные потоки могут работать. Но если вы хотите добиться гладкой многопоточной работы, то не должны полагаться на это. К тому же некоторые типы задач интенсивно нагружают процессор. Такие потоки захватывают процессор. Потокам такого типа вы должны передавать управление от случая к случаю, чтобы дать возможность выполняться другим.

Чтобы установить приоритет потока, используйте метод `setPriority()`, который является членом класса `Thread`. Так выглядит его общая форма:

```
final void setPriority(int уровень)
```

Здесь *уровень* специфицирует новый уровень приоритета для вызывающего потока. Значение *уровень* должно быть в пределах диапазона от `MIN_PRIORITY` до `MAX_PRIORITY`. В настоящее время эти значения равны соответственно 1 и 10. Чтобы вернуть потоку приоритет по умолчанию, укажите `NORM_PRIORITY`, который в настоящее время равен 5. Эти приоритеты определены как статические финальные (`static final`) переменные в классе `Thread`.

Вы можете получить текущее значение приоритета потока, вызвав метод `getPriority()` класса `Thread`, как показано ниже:

```
final int getPriority()
```

Реализации Java могут иметь принципиально разное поведение в том, что касается планирования потоков. Версия для Windows XP/98/NT/2000 работает более-менее ожидаемым образом. Однако другие версии могут работать несколько иначе. Большинство несовпадений возникают, когда вы полагаетесь на вытесняющую многозадачность вместо совместного использования времени процессора. Наиболее безопасный способ получить предсказуемое межплатформенное поведение Java — это использовать потоки, которые принудительно осуществляют управление центральным процессором.

В следующем примере демонстрируются два потока с разными приоритетами, которые выполняются на платформе без вытесняющей многозадачности иначе, чем на платформе с упомянутой многозадачностью. Один поток получает приоритет на два уровня выше нормального, как определено `Thread.NORM_PRIORITY`, а другой — на два уровня ниже нормального. Потоки стартуют и готовы к выполнению в течение 10 секунд. Каждый поток выполняет цикл, подсчитывающий количество итераций. Через 10 секунд главный поток останавливает оба потока. Затем количество итераций цикла, которое успел выполнить каждый поток, отображается.

```
// Демонстрация приоритетов потоков.
class clicker implements Runnable {
long click = 0;
Thread t;
private volatile boolean running = true;
```

```
public clicker(int p) {
  t = new Thread(this);
  t.setPriority(p);
}
public void run() {
  while (running) {
    click++;
  }
}
public void stop() {
  running = false;
}
public void start() {
  t.start();
}
}
class HiLoPri {
public static void main(String args[]) {
  Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
  clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
  clicker lo = new clicker(Thread.NORM_PRIORITY - 2);
  lo.start();
  hi.start();
  try {
    Thread.sleep(10000);
  } catch (InterruptedException e) {
    System.out.println("Главный поток прерван.");
  }
  lo.stop();
  hi.stop();
  // Ожидание 10 секунд до прерывания.
  try {
    hi.t.join();
    lo.t.join();
  } catch (InterruptedException e) {
    System.out.println("Перехвачено исключение InterruptedException");
  }
  System.out.println("Низкоприоритетный поток: " + lo.click);
  System.out.println("Высокоприоритетный поток: " + hi.click);
}
}
```

Вывод этой программы при запуске под Windows показывает, что потоки осуществляли переключение контекста, хотя не было никакого принудительного захвата процессора и никаких блокирующих операций ввода-вывода. Высокоприоритетный поток получил бо́льшую часть времени процессора.

```
Низкоприоритетный поток:  4408112
Высокоприоритетный поток:  589626904
```

Конечно, точный вывод, порождаемый этой программой, зависит от скорости вашего процессора и количества задач, выполняемых в системе. Когда та же программа запускается в среде с не вытесняющей многозадачностью, получается другой результат.

Еще одно замечание относительно предыдущей программы. Обратите внимание, что переменной `running` предшествует слово `volatile`. Хотя `volatile` более подробно объ-

ясняется в главе 13, оно используется здесь, чтобы гарантировать, что значение `running` будет проверяться на каждом шаге итераций цикла:

```
while(running) {
  click++;
}
```

Без указания `volatile` Java имеет возможность оптимизировать цикл таким образом, что будет создана локальная копия `running`. Применение `volatile` предотвращает эту оптимизацию, сообщая Java, что `running` может изменяться неявным для кода образом.

Синхронизация

Когда два или более потоков имеют доступ к одному разделенному ресурсу, они нуждаются в обеспечении того, что ресурс будет использован только одним потоком одновременно. Процесс, с помощью которого это достигается, называется *синхронизацией*. Как вы увидите, Java предлагает ее уникальную поддержку на уровне языка.

Ключом к синхронизации является концепция монитора (также называемого *семафором*). *Монитор* — это объект, который используется, как взаимное исключение (mutually exclusive lock — mutex), или *мьютекс*. Только один поток одновременно может *владеть* монитором. Когда поток запрашивает блокировку, говорят, что он *входит* в монитор. Все другие потоки, которые пытаются войти в заблокированный монитор, будут приостановлены до тех пор, пока первый поток не *выйдет* из монитора. Обо всех этих прочих потоках говорят, что они *ожидают* монитора. Поток, который владеет монитором, может повторно войти в него, если пожелает.

Если вы имели дело с синхронизацией в других языках, таких как C или C++, то знаете, что использовать ее не просто. Это потому, что эти языки сами по себе не поддерживают синхронизацию. Вместо этого, чтобы синхронизировать потоки, ваша программа должна использовать примитивы операционной системы. К счастью, поскольку Java реализует синхронизацию через языковые элементы, большая часть сложности, ассоциированная с синхронизацией, исчезает.

Вы можете синхронизировать ваш код двумя способами. Оба предусматривают использование ключевого слова `synchronized`, и оба способа мы здесь рассмотрим.

Использование синхронизированных методов

Синхронизация в Java проста, поскольку объекты имеют собственные, ассоциированные с ними неявные мониторы. Чтобы войти в монитор объекта, следует просто вызвать метод, модифицированный ключевым словом `synchronized`. Когда поток находится внутри синхронизированного метода, все другие потоки, которые пытаются вызвать его (или любые другие синхронизированные методы) на том же экземпляре, должны ожидать. Чтобы выйти из монитора и передать управление объектом другому ожидающему потоку, владелец монитора просто возвращает управление из синхронизированного метода.

Чтобы понять необходимость синхронизации, давайте начнем с простого примера, который не использует ее, хотя и должен. Следующая программа содержит три простых класса. Первый из них, `Callme`, имеет единственный метод — `call()`. Этот метод принимает параметр типа `String` по имени `msg`. Этот метод пытается напечатать строку `msg` внутри квадратных скобок. Интересно отметить, что после того, как `call()` печатает открывающую скобку и строку `msg`, он вызывает `Thread.sleep(1000)`, который приостанавливает текущий поток на одну секунду.

Конструктор следующего класса, `Caller`, принимает ссылку на экземпляр класса `Callme` и `String`, которые сохраняются соответственно в `target` и `msg`. Конструктор также создает новый поток, который вызовет метод `run()` объекта. Поток стартует немедленно. Метод `run()` класса `Caller` вызывает метод `call()` на экземпляре `target` класса `Callme`, передавая ему строку `msg`. Наконец, класс `Synch` начинает с создания единственного экземпляра `Callme` и трех экземпляров `Caller`, каждый с уникальной строкой сообщения. Один экземпляр `Callme` передается каждому `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!");
  // wait for threads to end
  try {
     ob1.t.join();
     ob2.t.join();
     ob3.t.join();
  } catch(InterruptedException e) {
     System.out.println("Прервано");
  }
}
}
```

Вот вывод, сгенерированный этой программой:

```
Добро пожаловать[в синхронизированный[мир!]
]
]
```

Как видите, вызывая `sleep()`, метод `call()` позволяет переключиться на выполнение другого потока. Это приводит к смешанному выводу трех строк сообщений. В этой программе нет ничего, что предотвращает вызов потоками одного и того же метода на одном и том же объекте в одно и то же время. Это называется *состоянием гонок*, поскольку три потока соревнуются друг с другом в окончании выполнения метода. Этот пример использует `sleep()`, чтобы сделать эффект повторяемым и наглядным. В большинстве ситуаций этот эффект более неуловим и менее предсказуем, поскольку вы не можете предвидеть, когда произойдет переключение контекста. Это может привести к тому, что программа один раз отработает правильно, а другой раз — нет.

Чтобы исправить эту программу, вы должны *сериализировать* доступ к `call()`. То есть вы должны разрешить доступ к этому методу одновременно только одному потоку. Чтобы сделать это, вам нужно просто предварить объявление `call()` ключевым словом `synchronized`, как показано ниже:

```
class Callme {
  synchronized void call(String msg) {
  ...
```

Это предотвратит доступ другим потокам к `call()`, когда один из них уже использует его. После того как слово `synchronized` добавлено к `call()`, результат работы программы будет выглядеть следующим образом:

```
[Добро пожаловать]
[в синхронизированный]
[мир!]
```

Всякий раз, когда у вас есть метод, или группа методов, которые манипулируют внутренним состоянием объекта в многопоточной среде, вы должны использовать ключевое слово `synchronized`, чтобы исключить ситуацию с гонками. Помните, что как только поток входит в любой синхронизированный метод на экземпляре, ни один другой поток не может войти ни в один синхронизированный метод на том же экземпляре. Однако не синхронизированные методы экземпляра по-прежнему остаются доступными для вызова.

Оператор `synchronized`

Хотя создание `synchronized` методов в ваших классах — простой и эффективный способ достижения синхронизации, все же он работает не во всех случаях. Чтобы понять, почему, рассмотрим следующее. Предположим, что вы хотите синхронизировать доступ к объектам классов, которые не были предназначены для многопоточного доступа. То есть класс не использует методов `synchronized`. Более того, класс был написан не вами, а независимым разработчиком, и у вас нет доступа к его исходному коду. Значит, вы не можете добавить слово `synchronized` к объявлению соответствующих методов класса. Как может быть синхронизирован доступ к объектам такого класса? К счастью, существует довольно простое решение этой проблемы: вы просто заключаете вызовы методов этого класса в блок `synchronized`.

Вот общая форма оператора `synchronized`:

```
synchronized(объект) {
  // операторы, подлежащие синхронизации
}
```

Здесь *`объект`* — это ссылка на синхронизируемый объект. Блок `synchronized` гарантирует, что вызов метода-члена *`объекта`* произойдет только тогда, когда текущий поток успешно войдет в монитор *`объекта`*.

Ниже показана альтернативная версия предыдущего примера с использованием синхронизированного блока внутри метода `run()`.

```
// Эта программа использует синхронизированный блок.
class Callme {
void call(String msg) {
  System.out.print("[" + msg);
  try {
     Thread.sleep(1000);
  } catch (InterruptedException e) {
     System.out.println("Interrupted");
  }
  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 call()
public void run() {
  synchronized(target) { // синхронизированный блок
     target.call(msg);
  }
}
}

class Synch1 {
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!");
  // wait for threads to end
  try {
     ob1.t.join();
     ob2.t.join();
     ob3.t.join();
  } catch(InterruptedException e) {
     System.out.println("Прервано");
  }
}
}
```

Здесь метод `call()` не модифицирован словом `synchronized`. Вместо этого используется оператор `synchronized` внутри метода `run()` класса `Caller`. Это позволяет получить тот же корректный результат, что и предыдущий пример, поскольку каждый поток ожидает окончания выполнения своего предшественника.

Межпотоковые коммуникации

Предыдущие примеры, безусловно, блокировали другие потоки от асинхронного доступа к некоторым методам. Это использование неявных мониторов объектов Java является мощным средством, но вы можете достичь более тонкого уровня контроля посредством межпроцессных коммуникаций. Как вы увидите, это особенно просто в Java.

Как обсуждалось ранее, многопоточность заменила программирование на основе циклов событий за счет разделения ваших задач на дискретные, логически обособленные единицы. Потоки также предоставляют вторичную выгоду: они исключают опрос. Опрос обычно реализуется в виде цикла, используемого для периодической проверки некоторого условия. Как только условие истинно, выполняется определенное действие. Это расходует время процессора. Например, рассмотрим классическую проблему, когда один поток генерирует некоторые данные, а другой принимает их. Чтобы сделать проблему более интересной, предположим, что поставщик данных должен ожидать, когда потребитель завершит, прежде чем поставщик сгенерирует новые данных. В системах с опросом потребитель данных тратит много циклов процессора на ожидание данных от поставщика. Как только поставщик завершает, он должен начать опрос, расходующий циклы процессора в ожидании завершения работы потребителя данных, и так далее. Понятно, что такая ситуация нежелательна.

Чтобы избежать опроса, Java включает элегантный механизм межпроцессных коммуникаций посредством методов `wait()`, `notify()` и `notifyAll()`. Эти методы реализованы как `final` в классе `Object`, поэтому они доступны всем классам. Все три метода могут быть вызваны только из `synchronized`-контекста. Хотя с точки зрения компьютерной науки они концептуально сложны, правила применения этих методов достаточно просты.

- `wait()` принуждает вызывающий поток отдать монитор и приостановить выполнение до тех пор, пока какой-нибудь другой поток не войдет в тот же монитор и не вызовет `notify()`.
- `notify()` возобновляет работу потока, который вызвал `wait()` на том же самом объекте.
- `notifyAll()` возобновляет работу всех протоков, который вызвали `wait()` на том же самом объекте. Одному из потоков дается доступ.

Эти методы объявлены в `Object`, как показано ниже:

```
final void wait() throws InterruptedException
final void notify()
final void notifyAll()
```

Существуют дополнительные формы `wait()`, позволяющие указать время ожидания. Прежде чем рассматривать пример, демонстрирующий межпотоковое взаимодействие, необходимо сделать одно важное замечание. Хотя `wait()` обычно ожидает до тех пор, пока не будет вызван `notify()` или `notifyAll()`, существует вероятность, что в очень редких случаях ожидающий поток может быть разбужен поддельным сигналом. При этом ожидающий поток возобновляется без вызова `notify()` или `notifyAll()`. (По сути, поток возобновляется без явных причин.) Из-за этой маловероятной возможности Sun рекомендует выполнять вызовы `wait()` внутри цикла, проверяющего условие, по которому поток ожидает. В приведенном ниже примере показан такой подход.

А пока рассмотрим пример, использующий `wait()` и `notify()`. Для начала проанализируем следующий простой пример программы, некорректно реализующий задачу "поставщик/потребитель". Она состоит из четырех классов: `Q` — очередь, которую нужно синхронизировать, `Producer` — объект-поток, который генерирует элементы очереди, `Consumer` — объект-поток, принимающий элементы очереди, и `PC` — крошечный класс, который создает объекты `Q`, `Producer` и `Consumer`.

```
// 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 останова нажмите Control-C.");
}
}
```

Несмотря на то что методы `put()` и `get()` в `Q` синхронизированы, ничто не остановит переполнение потребителя поставщиком, как и ничто не помешает потребителю извлечь один и тот же компонент очереди дважды. То есть вы получите неверный результат, показанный ниже (точная последовательность может быть другой, в зависимости от скорости процессора и загрузки).

```
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 отправляет 1, запускается потребитель и получает это же значение 1 пять раз подряд. Затем поставщик продолжает работу и поставляет значения от 2 до 7, не давая возможности потребителю получить их.

Правильный способ написания этой программы на Java заключается в том, чтобы применить `wait()` и `notify()`, чтобы передавать сигналы в обоих направлениях, как показано ниже:

```
// 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("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("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 останова нажмите Control-C.");
}
}
```

Внутри `get()` вызывается `wait()`. Это приостанавливает работу потока до тех пор, пока `Producer` не известит вас о том, что данные прочитаны. Когда это случается, выполнение внутри `get()` продолжается. После получения данных `get()` вызывает `notify()`. Это сообщает `Producer`, что все в порядке, и можно помещать в очередь следующий элемент данных. Внутри `put()` метод `wait()` приостанавливает выполнение до тех пор, пока `Consumer` не извлечет элемент из очереди. Когда выполнение возобновится, следующий элемент данных помещается в очередь и вызывается `notify()`. Это сообщает `Consumer`, что он теперь может извлечь его.

Ниже приведен вывод программы, который доказывает, что теперь синхронизация работает корректно.

```
Отправлено: 1
Получено: 1
Отправлено: 2
Получено: 2
Отправлено: 3
Получено: 3
Отправлено: 4
Получено: 4
Отправлено: 5
Получено: 5
```

Взаимная блокировка

Особый тип ошибок, которого следует избегать, имеющий отношение к многозадачности — это взаимная блокировка (deadlock), которая происходит, когда потоки имеют циклическую зависимость от пары синхронизированных объектов. Например, предположим, что один поток входит в монитор объекта X, а другой — в монитор объекта Y. Если поток в X попытается вызвать любой синхронизированный метод Y, он заблокируется, как и ожидалось. Однако если поток Y, в свою очередь, попытается вызвать любой синхронизированный метод X, то поток будет ожидать вечно, потому что для получения доступа к X он должен снять свой собственный блок на Y, чтобы первый поток мог отработать. Взаимная блокировка является ошибкой, которую трудно отладить, по двум описанным ниже причинам.

- В общем, она случается довольно редко, когда выполнение двух потоков точно совпадает по времени.
- Она может происходить, когда в этом участвует более двух потоков и двух синхронизированных объектов. (То есть взаимная блокировка может случиться в результате более сложной последовательности событий, чем в приведенном примере.)

Чтобы полностью разобраться с этим явлением, лучше рассмотреть его в действии. Следующий пример создает два класса — `A` и `B`, с методами `foo()` и `bar()` соответственно, которые приостанавливаются непосредственно перед попыткой вызова метода другого класса. Главный класс, названный `Deadlock`, создает экземпляры `A` и `B`, затем запускает второй поток, устанавливающий состояние взаимной блокировки. Методы `foo()` и `bar()` используют `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 A.foo");
  try {
     Thread.sleep(1000);
  } catch(Exception e) {
     System.out.println("A прерван");
  }
  System.out.println(name + " пытается вызвать B.last()");
  b.last();
}
synchronized void last() {
  System.out.println("внутри A.last");
}
}
class B {
synchronized void bar(A a) {
  String name = Thread.currentThread().getName();
  System.out.println(name + " вошел в B.bar");
  try {
     Thread.sleep(1000);
  } catch(Exception e) {
     System.out.println("B прерван");
  }
  System.out.println(name + " пытается вызвать A.last()");
  a.last();
}
```

```
synchronized void last() {
  System.out.println("внутри A.last");
}
}
class Deadlock implements Runnable {
A a = new A();
B b = new B();
Deadlock() {
  Thread.currentThread().setName("MainThread");
  Thread t = new Thread(this, "RacingThread");
  t.start();
  a.foo(b); // получить блокировку внутри этого потока.
  System.out.println("Назад в главный поток");
}
public void run() {
  b.bar(a); // получить блокировку b в другом потоке.
  System.out.println("Назад в другой поток");
}
public static void main(String args[]) {
  new Deadlock();
}
}
```

Когда вы запустите эту программы, то увидите следующий результат:

```
MainThread вошел в A.foo
RacingThread вошел в B.bar
MainThread пытается вызвать B.last()
RacingThread пытается вызвать A.last()
```

Поскольку эта программа заблокирована, вам придется нажать <CTRL+C> для завершения программы. Вы можете видеть весь поток и дамп кэша монитора, нажав <CTRL+BREAK>. Вы увидите, что `RacingThread` владеет монитором на `b`, в то время как последний ожидает монитора на `a`. В то же время `MainThread` владеет `a` и ожидает `b`. Эта программа никогда не завершится. Как иллюстрирует этот пример, если ваша многопоточная программа неожиданно зависла, то первое, что вы должны проверить — возможность взаимной блокировки.

Приостановка, возобновление и останов потоков

Иногда возникает необходимость в приостановке выполнения потоков. Например, отдельный поток может использоваться для отображения времени дня. Если пользователю не нужны часы, то этот поток можно приостановить. В любом случае приостановка потока — простая вещь. Выполнение приостановленного потока может быть легко возобновлено.

Механизм для временной либо окончательной остановки потока, а также его возобновления отличался в ранних версиях Java, таких как Java 1.0, от современных версий, начиная с Java 2. Хотя при написании нового кода вы должны придерживаться нового подхода, вы по-прежнему должны понимать, как эти операции были реализованы в ранних версиях среды Java. Например, может возникнуть необходимость в поддержке или обновлении старого, унаследованного кода. Вам также может понадобиться понять, по-

чему в этот механизм были внесены изменения. По этим причинам в следующем разделе описан изначальный способ управления выполнением потоков, а за ним следует раздел, описывающий, как это реализовано в новых версиях.

Приостановка, возобновление и останов потоков в Java 1.1 и более ранних версиях

До версии Java 2 программы использовали `suspend()` и `resume()`, которые были методами, определенными в `Thread` для приостановки и возобновления потоков. Они имеют следующую форму:

```
final void suspend()
final void resume()
```

В следующей программе демонстрируется применение этих методов:

```
// Использование suspend() и resume().
class NewThread implements Runnable {
String name; // имя потока
Thread t;
NewThread(String threadname) {
  name = threadname;
  t = new Thread(this, name);
  System.out.println("Новый поток: " + t);
  t.start(); // запуск потока
}
// Точка входа потока.
public void run() {
  try {
     for(int i = 15; i > 0; i--) {
       System.out.println(name + ": " + i);
       Thread.sleep(200);
     }
  } catch (InterruptedException e) {
     System.out.println(name + " прерван.");
  }
  System.out.println(name + " завершен.");
}
}
class SuspendResume {
public static void main(String args[]) {
  NewThread ob1 = new NewThread("Один");
  NewThread ob2 = new NewThread("Два");
  try {
     Thread.sleep(1000);
     ob1.t.suspend();
     System.out.println("Приостановка потока Один");
     Thread.sleep(1000);
     ob1.t.resume();
     System.out.println("Возобновление потока Один");
     ob2.t.suspend();
     System.out.println("Приостановка потока Два");
     Thread.sleep(1000);
     ob2.t.resume();
     System.out.println("Возобновление потока Два");
  } catch (InterruptedException e) {
```

```
      System.out.println("Главный поток прерван");
    }
    // Ожидание завершения потоков
    try {
      System.out.println("Ожидание завершения потоков.");
      ob1.t.join();
      ob2.t.join();
    } catch (InterruptedException e) {
      System.out.println("Главный поток прерван");
    }
    System.out.println("Главный поток завершен.");
  }
  }
```

Пример вывода этой программы показан ниже (в вашем конкретном случае он может отличаться, в зависимости от скорости и загрузки процессора).

```
Новый поток: Thread[Один,5,main]
Один: 15
Новый поток: Thread[Два,5,main]
Два: 15
Один: 14
Два: 14
Один: 13
Два: 13
Один: 12
Два: 12
Один: 11
Два: 11
Приостановка потока Один
Два: 10
Два: 9
Два: 8
Два: 7
Два: 6
Возобновление потока Один
Приостановка потока Два
Один: 10
Один: 9
Один: 8
Один: 7
Один: 6
Возобновление потока Two
Ожидание завершения потоков.
Два: 5
Один: 5
Два: 4
Один: 4
Два: 3
Один: 3
Два: 2
Один: 2
Два: 1
Один: 1
Два завершен.
Один завершен.
Главный поток завершен.
```

Класс Thread также определяет метод stop(), который останавливает поток. Его сигнатура такова:

```
final void stop()
```

Остановленный поток уже не может быть возобновлен с помощью resume().

Современный способ приостановки, возобновления и остановки потоков

Хотя применение методов класса Thread по именам suspend(), resume() и stop() выглядит как исключительно разумный и удобный подход к управлению выполнением потоков, они не должны использоваться в новых Java-программах. И вот почему. Метод suspend() класса Thread несколько лет назад был объявлен нежелательным в Java 2. Это было сделано потому, что иногда он способен порождать серьезные системные сбои. Предположим, что поток пытается получить блокировки на критичных структурах данных. Если поток приостановит в этот момент, блокировки не будут установлены. Другие потоки, которые могут ожидать этих ресурсов, могут оказаться взаимно заблокированными.

Метод resume() также нежелателен. Он не вызовет проблем, но не может быть использован без метода suspend() как своего дополнения.

Метод stop() класса Thread также объявлен устаревшим в Java 2. Это было сделано потому, что он также иногда может послужить причиной серьезных системных сбоев. Предположим, что поток выполняет запись в критически важную структуру данных, и успел выполнить только частичное обновление. Если его остановить в этот момент, структура данных может оказаться в поврежденном состоянии.

Поскольку вы не можете теперь использовать методы suspend(), resume() или stop() для управления потоками, то можете подумать, что теперь вообще нет способа приостановить, возобновить или прервать поток. К счастью, это не так. Вместо этого поток должен быть проектирован так, чтобы метод run() периодически проверял, должно ли выполнение потока быть приостановлено, возобновлено или прервано. Обычно это достигается использованием переменной-флага, указывающей состояние потока. До тех пор, пока этот флаг имеет значение "запущен", метод run() должен продолжать выполнение. Если переменная имеет значение "прерван", поток должен приостановиться. Если флаг получает значение "стоп", то поток должен завершиться. Конечно, существует множество способов написать такой код, но основной принцип остается неизменным для всех программ.

В следующем примере показано как методы wait() и notify(), унаследованные от Object, могут применяться для управления выполнением потока. Этот пример похож на программу из предыдущего раздела. Однако вызовы устаревших методов в ней исключены. Рассмотрим работу этой программы.

Класс NewThread содержит переменную экземпляра типа boolean по имени suspendFlag, используемую для управления выполнением потока. Конструктор инициализирует ее значением false. Метод run() содержит блок synchronized, который проверяет состояние suspendFlag. Если ее значение равно true, вызывается метод wait() для приостановки выполнения потока. Метод mysuspend() устанавливает значение suspendFlag в true. Метод myresime() устанавливает suspendFlag в false и вызывает notify() для того, чтобы "разбудить" поток. И, наконец, метод main() модифицирован для вызова методов mysuspend() и myresime().

```
// Приостановка и возобновление потока современным способом.
class NewThread implements Runnable {
   String name; // имя потока
   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
}
// Точка входа потока.
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.");
}
void mysuspend() {
  suspendFlag = true;
}
synchronized void myresume() {
  suspendFlag = false;
  notify();
}
}
class SuspendResume {
public static void main(String args[]) {
  NewThread ob1 = new NewThread("Один");
  NewThread ob2 = new NewThread("Два");
  try {
     Thread.sleep(1000);
     ob1.mysuspend();
     System.out.println("Приостановка потока Один");
     Thread.sleep(1000);
     ob1.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");
  }
```

```
    // ожидание завершения потоков
    try {
      System.out.println("Ожидание завершения потоков.");
      ob1.t.join();
      ob2.t.join();
    } catch (InterruptedException e) {
      System.out.println("Главный поток прерван");
    }
    System.out.println("Главный поток завершен");
  }
  }
```

Вывод этой программы идентичен приведенному в предыдущем разделе. Чуть позднее в этой книге вы найдете еще примеры, в которых используется современный механизм управления потоками. Хотя этот метод не так “чист”, как старый, его следует придерживаться, дабы избежать ошибок времени выполнения. Это подход, который *должен* применяться во всем новом коде.

Использование многопоточности

Ключ к эффективному использованию многопоточный средств Java лежит в том, чтобы думать параллельно вместо того, чтоб думать последовательно. Например, когда вы имеете две подсистемы в программе, которые могут выполняться одновременно, оформите их в виде отдельных потоков. При взвешенном применении многопоточности вы будете писать очень эффективные программы. Однако следует проявлять осторожность. Если вы создадите слишком много потоков, вы можете даже снизить производительность всей программы вместо того, чтобы повысить ее. Помните, что переключение контекстов между потоками требует определенных накладных расходов. Если вы создадите очень много потоков, больше времени процессора будет затрачено на переключение контекста, нежели на само выполнение программы!

12 ГЛАВА

Перечисления, автоупаковка и аннотации (метаданные)

В настоящей главе рассматриваются три последних дополнения к языку Java: перечисления, автоупаковка и аннотации (называемые также метаданными). Каждое их них увеличивает мощь языка, предлагая изящный подход к решению часто возникающих задач программирования. В главе также обсуждаются оболочки типов Java и рефлексия.

Перечисления

Версиям, предшествовавшим JDK 5, недоставало одного средства, необходимость в котором чувствовали многие программисты: перечисления. В простейшей форме *перечисление* — это список именованных констант. Хотя Java включает и другие средства, имеющие похожую функциональность, такие как переменные `final`, многим программистам все же не хватало концептуальной чистоты перечислений — в особенности потому, что они применяются во многих других языках программирования. Начиная с JDK 5, перечисления были добавлены к языку Java и, наконец, стали доступны программистам на Java.

В простейшей форме перечисления Java подобны перечислениям в других языках. Однако это сходство поверхностно. В языках вроде C++ перечисления просто представляют собой списки целочисленных констант. В Java перечисления определяют тип класса. За счет реализации перечислений в виде классов сама концепция перечисления значительно расширяется. Например, в Java перечисления могут иметь конструкторы, методы и переменные-экземпляры. Таким образом, хотя воплощения перечислений пришлось ждать несколько лет, реализация их в Java стоила того.

Основные понятия о перечислениях

Перечисления создаются использованием ключевого слова `enum`. Например, ниже показано простое перечисление сортов яблок:

```
// Перечисление сортов яблок.
enum Apple {
   Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
```

Идентификаторы Jonathan, GoldenDel и так далее называются *константами перечисления*. Каждая из них явно объявлена как общедоступный статический финальный член класса Apple. Более того, их тип — это тип перечисления, в котором они объявлены — в данном случае это Apple. То есть в языке Java эти константы называются *самотипизированными*, причем "*само*" ссылается на окружающее перечисление.

Объявив перечисление, вы можете создавать переменные этого типа. Однако даже несмотря на то, что перечисления определяют тип класса, вы не можете создавать объекты этого типа с помощью операции new. Вместо этого вы объявляете и используете переменную перечисления почти таким же образом, как это делается с элементарными типами. Например, ниже объявляется ap как переменная перечислимого типа Apple:

```
Apple ap;
```

Поскольку ap имеет тип Apple, присвоить ей можно только те значения, которые определены в перечислении. Например, здесь присваивается ap значение RedDel:

```
ap = Apple.RedDel;
```

Обратите внимание, что RedDel предшествует Apple.

Две перечислимых константы можно проверять на равенство с помощью операции отношения ==. Например, следующий оператор сравнивает ap с константой Apple.GoldenDel:

```
if (ap == Apple.GoldenDel) // ...
```

Перечислимые значения также могут быть использованы в управляющей конструкции switch. Конечно же, все операторы case должны использовать константы из того же enum, что и выражение switch. Например, следующий switch абсолютно корректен:

```
// Использование enum для управления switch.
switch(ap) {
case Jonathan:
  // ...
case Winesap:
  // ...
```

Обратите внимание, что в операторах case имена перечислимых констант используются без квалифицированного имени их типа перечисления. То есть применяется Winesap, а не Apple.Winesap. Дело в том, что тип перечисления в операторе switch уже неявно задает тип enum для операторов case. Нет необходимости квалифицировать константы в операторах case именем их типа enum. Фактически попытка сделать это приведет к ошибке компиляции.

Когда константа перечисления отображается, скажем, методом println(), выводится ее имя. Например, следующая строка кода:

```
System.out.println(Apple.Winesapp)
```

отобразит имя Winesapp.

Приведенная ниже программа собирает все вместе и демонстрирует 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;
   // Вывод значения enum.
   System.out.println("Значение ap: " + ap);
   System.out.println();
   ap = Apple.GoldenDel;
   // Сравнение двух перечислимых значений.
   if(ap == Apple.GoldenDel)
     System.out.println("ap содержит GoldenDel.\n");
   // Применение enum для управления оператором switch.
   switch(ap) {
     case Jonathan:
        System.out.println("Jonathan красный.");
        break;
     case GoldenDel:
        System.out.println("Golden Delicious желтый.");
        break;
     case RedDel:
        System.out.println("Red Delicious красный.");
        break;
     case Winesap:
        System.out.println("Winesap красный.");
        break;
     case Cortland:
        System.out.println("Cortland красный.");
        break;
   }
}
}
```

Эта программа создает следующий вывод:

```
Значение ap: RedDel

ap содержит GoldenDel.

Golden Delicious желтый.
```

Методы `values()` и `valueOf()`

Перечисления автоматически включают два предопределенных метода: `values()` и `valueOf()`.

Их общая форма выглядит так:

```
public static тип_enum[] values()
public static тип_enum valueOf(String строка)
```

Метод `values()` возвращает массив, содержащий список констант перечисления. Метод `valueOf()` возвращает константу перечисления, чье значение соответствует строке, переданной в аргументе *строка*. В обоих случаях *тип_enum* — это тип перечисления. Например, в случае с перечислением `Apple`, показанным выше, типом возврата `Apple.valueOf("Winesapp")` будет `Winesapp`.

В следующей программе демонстрируется применение методов `values()` и `valueOf()`.

```
// Использование встроенных методов перечислений.
// Перечисление сортов яблок.
enum Apple {
   Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
class EnumDemo2 {
public static void main(String args[])
{
  Apple ap;
  System.out.println("Константы Apple:");
  // применение values()
  Apple allapples[] = Apple.values();
  for(Apple a : allapples)
     System.out.println(a);
  System.out.println();
  // применение valueOf()
  ap = Apple.valueOf("Winesap");
  System.out.println("ap содержит " + ap);
}
}
```

Вывод этой программы:

```
Константы Apple:
Jonathan
GoldenDel
RedDel
Winesap
Cortland

ap содержит Winesap
```

Обратите внимание, что программа использует стиль "for-each" цикла `for` для прохода по массиву констант, возвращенных `values()`. В целях демонстрации создается переменная `allapples` и ей присваивается ссылка на массив перечислимых значений. Однако этот шаг не является необходимым, поскольку `for` можно написать, как показано ниже, избежав необходимости в переменной `allapples`:

```
for(Apple a : Apple.values())
  System.out.println(a);
```

Также обратите внимание, как значение, соответствующее имени `Winesapp` получается вызовом метода `valueOf()`:

```
ap = Apple.valueOf("Winesap");
```

Как объяснялось ранее, `valueOf()` возвращает перечислимое значение, ассоциированное с именем константы, переданным в строке.

На заметку! *Программисты на C/C++ обратят внимание на то, что в Java значительно упрощено преобразование между читабельной для человека формой константы перечисления и его бинарным значением по сравнению с другими языками. Это существенное преимущество подхода к перечислениям языка Java.*

Перечисления в Java являются типами классов

Как уже объяснялось, перечисление в Java — это тип класса. Хотя вы не можете создать экземпляр enum с помощью операции new, в остальном перечисление обладает всеми возможностями, которые имеются у других классов. Тот факт, что enum определяет класс, придает такую мощь перечислениям Java, которой лишены перечисления в других языках. Например, вы можете предоставлять им конструкторы, добавлять переменные экземпляров и методы, и даже реализовывать интерфейсы.

Важно понимать, что каждая константа перечисления является объектом его типа перечисления. То есть, когда вы определяете конструктор для enum, он вызывается при каждом создании константы перечисления. Также каждая константа перечисления имеет свою собственную копию переменных экземпляра, объявленных перечислением. Например, рассмотрим следующую версию Apple.

```
// Использование конструктора enum, переменной экземпляра и метода.
enum Apple {
   Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);
   private int price; // цена каждого яблока
   // Конструктор
   Apple(int p) { price = p; }
   int getPrice() { return price; }
}

class EnumDemo3 {
public static void main(String args[])
{
  Apple ap;
  // Отобразить цену Winesap.
  System.out.println("Winesap стоит " +
                     Apple.Winesap.getPrice() +
                     " центов.\n");
  // Отобразить цены всех сортов яблок.
  System.out.println("Все цены яблок:");
  for(Apple a : Apple.values())
  System.out.println(a + " стоит " + a.getPrice() +
                     " центов.");
}
}
```

Ниже показан вывод программы.

```
Winesap стоит 15 центов.
Все цены яблок:
Jonathan стоит 10 центов.
GoldenDel стоит 9 центов.
RedDel стоит 12 центов.
Winesap стоит 15 центов.
Cortland стоит 8 центов.
```

Данная версия Apple добавляет три вещи. Первая — это переменная экземпляра price, которая применяется для хранения цены каждого из сортов яблок. Вторая — конструктор Apple, которому передается цена яблок. Третий — метод getPrice(), возвращающий значение цены.

Когда в `main()` объявляется переменная `ap`, конструктор `Apple` вызывается однажды для каждой объявленной константы. Следует отметить, что аргументы конструктору передаются помещением их в скобки после каждой константы, как показано ниже:

```
Jonathan(10), GoldenDel(9), RedDel(12), Winesap(15), Cortland(8);
```

Эти переменные передаются параметру `p` конструктора `Apple()`, который затем присваивает их `price`. Опять же конструктор вызывается однажды для каждой из констант.

Поскольку каждая из констант перечисления имеет свою собственную копию `price`, вы можете получить цену определенного сорта яблок вызовом `getPrice()`. Например, в `main()` цена сорта `Winesap` получается следующим вызовом:

```
Apple.Winesap.getPrice()
```

Цены всех сортов получаются в циклическом проходе по перечислению посредством цикла `for`. Поскольку существует копия `price` для каждой перечислимой константы, значение, ассоциированное одной константой, отделено и отличается от значения, ассоциированного с другой константой. Это мощная концепция, которая доступна только в случае реализации перечислений в виде классов, как это сделано в Java.

Хотя предыдущий пример содержит только один конструктор, `enum` может представлять две или более перегруженных формы, как это может делать любой другой класс. Например, приведенная ниже версия `Apple` предлагает конструктор по умолчанию, инициализирующий цену значением -1, означающим, что цена не указана.

```
// Использование конструкторов enum.
enum Apple {
   Jonathan(10), GoldenDel(9), RedDel, Winesap(15), Cortland(8);
   private int price; // цена каждого яблока
   // Конструктор
   Apple(int p) { price = p; }
   // Перегруженный конструктор
   Apple() { price = -1; }
   int getPrice() { return price; }
}
```

Обратите внимание, что в этой версии `RedDel` не передается аргумент. Это означает, что вызывается конструктор по умолчанию и переменная цены `RedDel` устанавливается равной `-1`.

Здесь есть два ограничения относительно перечислений. Во-первых, перечисление не может наследоваться от другого класса. Во-вторых, `enum` не может быть суперклассом. Это значит, что `enum` не может быть расширен. Во всем остальном `enum` ведет себя как любой другой тип класса. Ключевой момент — помнить, что каждая константа перечисления является объектом класса, в котором она определена.

Перечисления наследуются от `Enum`

Хотя вы не можете наследовать суперкласс при объявлении `enum`, все перечисления автоматически наследуют `java.lang.Enum`. Этот класс определяет несколько методов, доступных к использованию только всеми перечислениями. Класс `Enum` детально рассматривается во второй части книги, но три из его методов требуют описания прямо сейчас.

Вы можете получить значение, которое указывает позицию константы в списке констант перечисления. Это называется *порядковым значением* (ordinal value) и извлекается с помощью вызова метода `ordinal()`, показанного ниже:

```
final int ordinal()
```

Он возвращает порядковое значение вызывающей константы. Порядковые значения начинаются с нуля. То есть в перечислении `Apple` `Johnatan` имеет порядковое значение нуль, `GoldenDel` имеет порядковое значение 1, `RedDel` — 2 и так далее.

Вы можете сравнить порядковые значения двух констант одного и того же перечисления с помощью метода `compareTo()`. Он имеет следующую общую форму:

```
final int compareTo(тип_enum e)
```

Здесь *тип_enum* — тип перечисления, а *e* — константа, которую нужно сравнить с вызывающей константой. Помните, что вызывающая константа и *e* должны относиться к одному перечислению. Если вызывающая константа имеет порядковое значение меньше чем *e*, то `compareTo()` возвращает отрицательное значение. Если два порядковых значения одинаковы, возвращается ноль. Если вызывающая константа имеет порядковое значение больше чем *e*, то возвращается положительное значение.

Вы можете сравнить на эквивалентность перечислимую константу с любым другим объектом, используя `equals()` — переопределенный метод `equals()` класса `Object`. Хотя `equals()` может сравнивать перечислимые константы с любым другим объектом, эти два объекта будут эквивалентны только в случае, если оба они являются ссылкой на одну и ту же константу из одного и того же перечисления. Простое совпадение порядковых значений не заставит `equals()` вернуть `true`, если две константы принадлежат разным перечислениям.

Помните, что вы можете сравнивать две ссылки перечислений на эквивалентность, используя операцию ==.

В следующей программе демонстрируется применение методов `ordinal()`, `compareTo()` и `equals()`.

```
// Демонстрация 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 ordinal().
  System.out.println("Вот все константы " +
                     " 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();
  // Демонстрация compareTo() и equals()
  if(ap.compareTo(ap2) < 0)
     System.out.println(ap + " идет перед " + ap2);
  if(ap.compareTo(ap2) > 0)
     System.out.println(ap2 + " идет перед " + ap);
  if(ap.compareTo(ap3) == 0)
     System.out.println(ap + " эквивалентно " + ap3);
  System.out.println();
```

```
  if(ap.equals(ap2))
    System.out.println("Error!");
  if(ap.equals(ap3))
    System.out.println(ap + " equals " + ap3);
  if(ap == ap3)
    System.out.println(ap + " == " + ap3);
}
}
```

Ниже показан вывод этой программы.

```
Вот все константы и их порядковые значения:
Jonathan 0
GoldenDel 1
RedDel 2
Winesap 3
Cortland 4

GoldenDel идет перед RedDel
RedDel эквивалентно RedDel

RedDel эквивалентно RedDel
RedDel == RedDel
```

Еще один пример перечисления

Прежде чем двигаться дальше, рассмотрим еще один пример применения `enum`. В главе 9 создавалась программа для автоматического принятия решений. В этой версии переменные, называемые `NO`, `YES`, `MAYBE`, `LATER`, `SOON` и `NEVER`, были объявлены в интерфейсе и использованы для представления возможных ответов. Хотя в таком подходе нет ничего технологически неверного, применение перечислений — более подходящее решение. Здесь представлена усовершенствованная версия этой программы, которая использует `enum` по имени `Answers` для представления ответов. Вы должны сравнить эту версию с оригинальной из главы 9.

```
// Усовершенствованная версия программы принятия решений
// из главы 9. В этой версии для представления
// используется enum, 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 вы знаете, в Java для хранения базовых типов данных, поддерживаемых языком, используются примитивные типы (также называемые простыми типами), такие как `int` или `double`. Примитивные типы, в отличие от объектов, используются для таких значений из соображений производительности. Применение объектов для этих значений добавляет нежелательные накладные расходы, даже в случае простейших вычислений. Поэтому примитивные типы не являются частью иерархии объектов и не наследуются от `Object`.

Несмотря на то что примитивные типы обеспечивают выигрыш производительности, бывают случаи, когда вам может понадобиться объектное представление. Например, вы не можете передать в метод примитивный тип по ссылке. Кроме того, многие из стандартных структур данных, реализованных в Java, оперируют с объектами, что означает, что вы не можете применять эти структуры данных для сохранения примитивных типов. Чтобы справиться с такими (и подобными) ситуациями, Java предлагает *оболочки типов*,

которые представляют собой классы, скрывающие примитивный тип в объект. Классы-оболочки типов детально описаны во второй части книги, но также представлены здесь, поскольку имеют непосредственное отношение к автоупаковке Java.

Оболочки типов — это `Double`, `Float`, `Long`, `Integer`, `Short`, `Byte`, `Character` и `Boolean`. Эти классы предоставляют широкий диапазон методов, позволяющий в полной мере интегрировать примитивные типы в иерархию объектных типов Java. Каждый из них кратко рассматривается далее.

Character

`Character` — это оболочка вокруг `char`. Конструктор для `Character` выглядит следующим образом:

```
Character(char ch)
```

Здесь *ch* указывает символ, который будет помещен в оболочку при создании объекта `Character`.

Чтобы получить значение `char`, содержащееся в объекте `Character`, вызывайте метод `charValue()`, показанный ниже:

```
char charValue()
```

Он возвращает инкапсулированный символ.

Boolean

`Boolean` — оболочка вокруг значений `boolean`. В ней определены следующие конструкторы:

```
Boolean(boolean boolValue)
Boolean(String boolString)
```

В первой версии *boolValue* должно быть либо `true`, либо `false`. Во второй версии, если *boolString* содержит значение строку “true” (в верхнем или нижнем регистре), то новый объект `Boolean` будет равен `true`. В противном случае он будет равен `false`.

Чтобы получить значение `boolean` из объекта `Boolean`, используйте метод `booleanValue()`, показанный ниже:

```
boolean booleanValue()
```

Он возвращает `boolean`-эквивалент вызывающего объекта.

Оболочки числовых типов

До сих пор наиболее часто используемые оболочки типов — это те, что представляют числовые значения. Это `Byte`, `Short`, `Integer`, `Long`, `Float` и `Double`. Все оболочки числовых типов наследуют абстрактный тип `Number`. `Number` объявляет методы, которые возвращают значение объекта в каждом из различных числовых форматов. Вот эти методы:

```
byte byteValue()
double doubleValue()
float floatValue()
int intValue()
long longValue()
short shortValue()
```

Например, `doubleValue()` возвращает значение объекта как `double`, `floatValue()` возвращает значение как `float` и так далее. Эти методы реализованы каждой из оболочек числовых типов.

Все числовые оболочки типов определяют конструкторы, которые позволяют конструировать объекты из заданного значения или строкового представления этого значения. Например, вот как выглядят конструкторы для `Integer`:

```
Integer(int num)
Integer(String str)
```

Если *str* не содержит числового значения, то бросается исключение `NumberFormatException`.

Все типы-оболочки переопределяют `toString()`. Этот метод возвращает читабельную для человека форму значения, содержащегося в оболочке. Это позволяет выводить значение, передавая объект оболочки типа методу `println()`, например, без необходимости преобразования его в примитивный тип. В следующей программе показано, как использовать оболочку числового типа для инкапсуляции значения и последующего его извлечения.

```
// Демонстрация оболочки типа.
class Wrap {
public static void main(String args[]) {
  Integer iOb = new Integer(100);
  int i = iOb.intValue();
  System.out.println(i + " " + iOb); // отображает 100 100
}
}
```

Эта программа помещает целое значение 100 внутрь объекта `Integer` по имени `iOb`. Эта программа затем получает значение вызовом `intValue()` и помещает результат в `i`.

Процесс инкапсуляции значения в объект называется *упаковкой* (boxing). То есть следующая строка программы упаковывает значение 100 в `Integer`:

```
Integer iOb = new Integer(100);
```

Процесс извлечения значения из оболочки типа называется *распаковкой* (unboxing). Например, приведенная строка программы распаковывает значение `iOb`:

```
int i = iOb.intValue();
```

Та же самая общая процедура, что используется в предыдущей программе для упаковки и распаковки значений, применялась и в исходной версии Java. Однако в версию Java J2SE 5 были внесены фундаментальные усовершенствования за счет добавления автоматической упаковки, описанной ниже.

Автоупаковка

Начиная с JDK 5, к языку Java добавлены два важных средства: *автоупаковка* (autoboxing) и *автораспаковка* (autounboxing). Автоупаковка — это процесс, посредством которого примитивный тип автоматически инкапсулируется (упаковывается) в эквивалентную ему оболочку типа — всякий раз, когда требуется объект этого типа. Нет необходимости явного конструирования объекта. Автораспаковка — это процесс, с помощью которого значение упакованного объекта автоматически извлекается (распаковывается)

из оболочки типа, когда нужно получить его значение. Нет необходимости вызывать методы вроде `intValue()` или `doubleValue()`.

Добавление автоматической упаковки и распаковки значительно упрощает кодирование некоторых алгоритмов, исключая необходимость в ручной упаковке и распаковке значений. Это также помогает предотвратить ошибки. Более того, это очень важно для средства обобщения классов и алгоритмов, которые оперируют только объектами. И, наконец, автоупаковка существенно облегчает работу с каркасом коллекций (Collection Framework), описанным во второй части книги.

С автоупаковкой больше нет необходимости в ручном конструировании объектов для оболочки примитивных типов. Вам нужно только присвоить значение ссылке оболочки типов. Java автоматически конструирует эти объекты для вас. Например, вот современный способ конструировать объект `Integer`, который содержит значение 100:

```
Integer iOb = 100; // автоупаковка int
```

Обратите внимание, что никакого объекта явно операцией `new` не создается. Java делает это для вас автоматически.

Чтобы распаковать объект, просто присваивайте ссылку на объект переменной примитивного типа. Например, чтобы распаковать `iOb`, вы должны использовать следующую строку:

```
int i = iOb; // автораспаковка
```

Java справляется с деталями за вас.

Вот предыдущая программа, переписанная для использования автоупаковки/автораспаковки:

```
// Демонстрация автоупаковки/автораспаковки.
class AutoBox {
public static void main(String args[]) {
  Integer iOb = 100; // автоупаковка int
  int i = iOb; // автораспаковка
  System.out.println(i + " " + iOb); // отображает 100 100
}
}
```

Автоупаковка и методы

В дополнение к простым случаям присвоения 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/автораспаковка может случиться, когда аргумент передается методу либо когда значение возвращается из метода. Рассмотрим пример.

```
// Автоупаковка/автораспаковка происходит
// с методами параметров и возвращаемыми значениями.

class AutoBox2 {
   // принять параметр Integer и вернуть
   // значение int;
   static int m(Integer v) {
      return v ; // автораспаковка int
   }
```

```
  public static void main(String args[]) {
    // Передача int методу m() и присвоение возвращаемого значения
    // объекту Integer. Здесь аргумент 100 автоматически упаковывается
    // в Integer. Возвращаемое значение также упаковывается в Integer.
    Integer iOb = m(100);
    System.out.println(iOb);
  }
}
```

Эта программа отображает следующий результат:

```
100
```

Обратите внимание, что в этой программе `m()` специфицирует параметр типа `Integer` и возвращает результат типа `int`. Внутри `main()` методу `m()` передается значение 100. Поскольку `m()` ожидает `Integer`, это значение автоматически упаковывается. Затем `m()` возвращает `int`-эквивалент аргумента. Это заставляет автоматически упаковаться `v`. Далее это `int`-значение присваивается `iOb` в `main()`, что вызывает автоматическую упаковку результата типа `int`.

Автоупаковка/распаковка происходит в выражениях

Вообще автоупаковка и распаковка происходят всякий раз, когда требуется преобразование в объект или из объекта. Это касается выражений. Внутри выражения числовой объект автоматически распаковывается. Выходной результат выражения при необходимости упаковывается заново. Например, рассмотрим 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);

  // Следующее автоматически распаковывает iOb,
  // выполняет его приращение, затем повторно
  // упаковывает результат обратно в iOb.
  ++iOb;
  System.out.println("После ++iOb: " + iOb);

  // Здесь iOb распаковано, выражение вычисляется,
  // а результат снова упаковывается и
  // присваивается iOb2.
  iOb2 = iOb + (iOb / 3);
  System.out.println("iOb2 после выражения: " + iOb2);

  // Вычисляется то же самое выражение,
  // но результат не упаковывается.
  i = iOb + (iOb / 3);
  System.out.println("i после выражения: " + i);
}
}
```

Вывод показан ниже:

```
Исходное значение iOb: 100
После ++iOb: 101
iOb2 после выражения: 134
i после выражения: 134
```

Обратите особое внимание на следующую строку программы:

```
++iOb;
```

Это вызывает увеличение на 1 значения `iOb`. Оно работает следующим образом: `iOb` распаковывается, значение увеличивается и результат упаковывается вновь.

Автоматическая распаковка также позволяет смешивать разные типы числовых объектов в одном выражении. Как только значение распаковано, применяются стандартные правила повышения типов и преобразования. Например, следующая программа абсолютно корректна:

```
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);
}
}
```

Результат показан ниже:

```
dOb после выражения: 198.6
```

Как видите, оба объекта — и `Double dOb`, и `Integer iOb` — участвуют в сложении, и результат повторно упаковывается и сохраняется в `dOb`.

Благодаря автоупаковке, можно применять целочисленные объекты для управления оператором `switch`. Например, рассмотрим 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 выражение `switch`, `iOb` распаковывается и получается его значение типа `int`.

Как показывают примеры программ, благодаря автоупаковке/распаковке, применение числовых объектов в выражениях интуитивно понятно и просто. В прошлом такой код требовал применения приведений и вызовов методов вроде `intValue()`.

Автоупаковка/распаковка значений `Boolean` и `Character`

Как описывалось ранее, Java также поддерживает оболочки для `boolean` и `char` — соответственно `Boolean` и `Character`. Автоупаковка/распаковка также применима к этим типам. Например, рассмотрим следующую программу:

```
// Автоупаковка/распаковка Boolean и Character.
class AutoBox5 {
public static void main(String args[]) {
  // Автоупаковка/распаковка boolean.
  Boolean b = true;
  // Ниже, b автоматически распаковывается
  // при использовании в условном выражении if.
  if(b) System.out.println("b равна true");
  // Автоупаковка/распаковка char.
  Character ch = 'x'; // упаковка char
  char ch2 = ch; // распаковка char
  System.out.println("ch2 равна " + ch2);
}
}
```

Результат этой программы:

```
b равна true
ch2 равна x
```

Наиболее важный момент в этой программе, о котором стоит упомянуть — это автоматическая распаковка b внутри условного выражения if. Как вы должны помнить, условное выражение, которое управляет if, должно при вычислении возвращать boolean. Благодаря автораспаковке, значение boolean, содержащееся в b, автоматически распаковывается при вычислении условного выражения. То есть с появлением автоупаковки/распаковки стало возможным применять объекты Boolean для управления в операторе if.

Благодаря автоупаковке/распаковке, объект Boolean теперь также может применяться для управления всеми циклическими конструкциями Java. Когда Boolean применяется в качестве условного выражения в while, for или do/while, оно автоматически распаковывается в свой boolean-эквивалент. Например, вот новый допустимый код:

```
Boolean b;
// ...
while(b) { // ...
```

Автоупаковка/распаковка помогает предотвратить ошибки

В дополнение к удобству, которое оно предоставляет, автоупаковка/распаковка может также помочь избежать ошибок. Например, рассмотрим следующую программу:

```
// Ошибка, порожденная "ручной" распаковкой.
class UnboxingError {
public static void main(String args[]) {
  Integer iOb = 1000;         // автоупаковка значения 1000
  int i = iOb.byteValue();    // ручная распаковка, как byte !!!
  System.out.println(i);      // не отображает 1000 !
}
}
```

Эта программа отображает не ожидаемое значения 1000, а –24! Причина состоит в том, что значение внутри iOb распаковано вручную вызовом byteVal(), что привело к усечению значения 1000, хранящегося в iOb. В результате получилось “мусорное” значение –24, которое было присвоено i. Автораспаковка предотвращает этот тип ошибок, поскольку значение iOb всегда будет автоматически распаковываться в значение, совместимое с int.

В общем, поскольку автоупаковка всегда создает правильный объект, а автораспаковка всегда порождает правильное значение, то нет опасности получить неверное значение или неверный объект. В тех редких случаях, когда вам нужно получить значение типа, отличающегося от того, который генерируется автоматически, вы можете вручную упаковывать и распаковывать значения. Конечно, выгоды от автоупаковки/автораспаковки в этом случае теряются. В принципе, новый код должен использовать автоупаковку/автораспаковку. Это — правильный способ написания нового кода на Java.

Предостережения

Теперь, когда Java включает средства автоматической упаковки/распаковки, некоторые могут подумать, что соблазнительно применять исключительно объекты вроде `Integer` или `Double`, исключая использование примитивных типов. Например, теперь можно написать код вроде следующего:

```
// Плохое применение автоупаковки/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 этом примере объекты типа `Double` хранят значения, которые используются для вычисления гипотенузы прямоугольного треугольника. Несмотря на то что этот код технически корректен и работает правильно, все же это очень плохое использование автоупаковки/автораспаковки. Он намного менее эффективен, чем эквивалентный код, использующий примитивный тип `double`. Причина в том, что автоупаковка и автораспаковка добавляют накладные расходы, которые отсутствуют в случае применения примитивных типов.

Вообще говоря, вы должны ограничивать использование оболочек типов только теми случаями, когда требуется объектное представление примитивных типов. Автоупаковка/автораспаковка была добавлена в Java вовсе не в качестве обходного маневра с целью исключения примитивных типов.

Аннотации (метаданные)

Начиная с JDK 5, в Java появилось новое мощное средство, которое позволяет встроить информацию поддержки в исходные файлы. Эта информация, называемая также *аннотацией*, не меняет действия программы. То есть аннотация сохраняет семантику программ неизменной. Однако эта информация может быть использована различными инструментальными средствами, как во время разработки, так и в период развертывания. Например, аннотация может обрабатываться генераторами исходного кода. Термин *метаданные* также используется для именования этого средства, но более описательный термин *средства аннотирования программ* применяется более широко.

Основы аннотирования

Аннотации создаются посредством механизма, основанного на интерфейсе. Начнем с примера. Вот объявление аннотации под названием `MyAnno`:

```
// Простой тип аннотации.
@interface MyAnno {
  String str();
  int val();
}
```

Во-первых, обратите внимание на символ `@`, предшествующий ключевому слову `interface`. Это говорит компилятору, что объявлен тип аннотации. Далее отметим два метода-члена — `str()` и `val()`. Все аннотации состоят только из объявлений методов. Однако вы не определяете тел этих методов. Вместо этого их реализует Java. Более того, методы ведут себя в большей степени подобно полям, как вы вскоре убедитесь.

Аннотация не может включать слова `extends`. Однако все аннотации автоматически расширяют интерфейс `Annotation`. То есть `Annotation` является суперинтерфейсом для всех аннотаций. Он объявлен в пакете `java.lang.annotation`. В `Annotation` переопределены методы `hashCode()`, `equals()` и `toString()`, которые определены в `Object`. В нем также специфицирован метод `annotationType()`,возвращающий объект `Class`, который представляет вызывающую аннотацию.

Объявив аннотацию, вы можете использовать ее для аннотирования объявления. Любой тип объявления может иметь аннотацию, ассоциированную с ним. Например, аннотироваться могут классы, методы, поля, параметры и константы `enum`. Аннотированной может быть даже сама аннотация. Во всех случаях аннотация предшествует остальной части объявления.

Когда вы применяете аннотацию, то присваиваете значения ее членам. Например, ниже показан вариант применения `MyAnno` к методу:

```
// Аннотирование метода.
@MyAnno(str = "Пример аннотации", val = 100)
public static void myMeth() { // ...
```

Эта аннотация связана с методом `myMeth()`. Посмотрите внимательно на синтаксис аннотации. За именем аннотации, которому предшествует `@`, следует взятый в скобки список инициализаторов членов. Чтобы присвоить члену значение, значение присваивается имени члена. Таким образом, в этом примере строка “Пример аннотации” присваивается члену `str MyAnno`. Обратите внимание, что в этом присваивании никаких скобок за `str` не следует. Когда члену аннотации дается значение, используется только его имя. То есть члены аннотации в данном контексте выглядят как поля.

Спецификация политики удержания

Прежде чем объяснять аннотации дальше, необходимо обсудить *политики удержания аннотаций* (annotation retention policies). Политика удержания определяет, в какой точке аннотация отбрасывается. Java определяет три таких политики, которые инкапсулированы в перечислении `java.lang.annotation.RetentionPolicy`. Это `SOURCE`, `CLASS` и `RUNTIME`.

Аннотации с политикой удержания `SOURCE` удерживаются только в исходном файле и отбрасываются при компиляции.

Аннотации с политикой удержания `CLASS` сохраняются в файле `.class` во время компиляции. Однако они недоступны JVM во время выполнения.

Аннотации с политикой удержания `RUNTIME` сохраняются в файле `.class` во время компиляции и остаются доступными JVM во время выполнения. То есть политика `RUNTIME` представляет аннотации наиболее высокой степени постоянства.

Политика удержания для аннотации задается с помощью одной из встроенных аннотаций Java: `@Retention`. Ее общая форма показана ниже:

```
@Retention(политика_удержания)
```

Здесь *политика_удержания* должна быть одной из описанных ранее констант. Если для аннотации не указано никакой политики сохранения, используется `CLASS`.

В следующей версии `MyAnno` с помощью `@Retention` указывается политика `RUNTIME`. То есть `MyAnno` будет доступна JVM во время выполнения программы.

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
   String str();
   int val();
}
```

Получение аннотаций во время выполнения с использованием рефлексии

Хотя аннотации спроектированы в основном для использования инструментами разработки и развертывания, если они специфицируют политику удержания `RUNTIME`, то могут быть опрошены во время выполнения любой Java-программой за счет использования рефлексии. Рефлексия — это средство, позволяющее получить информацию о классе во время выполнения программы. Программный интерфейс (API) рефлексии содержится в пакете `java.lang.reflect`. Существует множество способов применения рефлексии, и мы не будем здесь обсуждать их все. Тем не менее, мы пройдемся по нескольким примерам, имеющим отношение к аннотациям.

Первый шаг в использовании рефлексии — это получение объекта `Class`, представляющего класс, чью аннотацию нужно получить. `Class` — это один из встроенных классов Java, определенный в пакете `java.lang`. Он детально рассматривается во второй части книги. Есть разные способы получения объекта `Class`. Один из простейших — вызвать метод `getClass()`,определенный в `Object`. Его общая форма показана ниже:

```
final Class getClass()
```

Он возвращает объект `Class`, который представляет вызывающий объект. (`getClass()` и несколько других связанных с рефлексией методов, используют средство обобщения. Однако поскольку обобщения не обсуждаются вплоть до главы 14, эти методы показаны и используются здесь в их "сырой" форме. В результате при компиляции следующих программ вы увидите предупреждающие сообщения. В главе 14 обобщения будут описаны со всех подробностях.)

После того, как вы получите объект `Class`, вы должны использовать его методы для получения информации о различных элементах, объявленных в классе, включая его аннотацию. Если вы хотите получить аннотации, ассоциированные с определенным элементом класса, вы должны сначала получить объект, представляющий этот элемент. Например, `Class` представляет (помимо прочих) методы `getMethod()`, `getField()` и `getConstructor()`, которые получают информацию о методе, поле и конструкторе соответственно. Эти методы возвращают объекты типов `Method`, `Field` и `Constructor`.

Чтобы понять этот процесс, рассмотрим пример, который получает аннотации, ассоциированные с методом. Чтобы сделать это, вы сначала получаете объект `Class`, представляющий класс, затем вызываете метод `getMethod()` этого объекта, указав имя метода. `getMethod()` имеет следующую общую форму:

```
Method getMethod(String methName, Class ... paramTypes)
```

Имя метода передается в *methName*. Если метод принимает аргументы, то объекты Class, представляющие их типы, также должны быть указаны в *paramTypes*. Обратите внимание, что *paramTypes* — это список аргументов переменной длины (varargs). Это означает, что вы можете специфицировать столько типов параметров, сколько нужно, включая ноль. getMethod() возвращает объект Method, который представляет метод. Если метод не может быть найден, возбуждается исключение NoSuchMethodException.

От объектов Class, Method, Field или Constructor вы можете получить специфические аннотации, ассоциированные с этим объектом, обратившись к getAnnotation(). Его общая форма представлена ниже:

```
Annotation getAnnotation(Class annoType)
```

Здесь *annoType* — это объект Class, представляющий аннотацию, в которой вы заинтересованы. Метод возвращает ссылку на аннотацию. Используя эту ссылку, вы можете получить значения, ассоциированные с членами аннотации.

Метод возвращает null, если аннотация не найдена; это случай, когда аннотация не имеет @Retention, установленную в RUNTIME.

Ниже показана программа, которая собирает все описанные части вместе и использует рефлексию для отображения аннотации, ассоциированной с 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 отобразить значения членов.
  try {
     // Для начала получить Class,
     // представляющий класс.
     Class c = ob.getClass();
     // Теперь получить объект Method,
     // представляющий этот метод.
     Method m = c.getMethod("myMeth");
     // Далее получить аннотацию класса.
     MyAnno anno = m.getAnnotation(MyAnno.class);
     // Наконец, отобразить аннотацию.
     System.out.println(anno.str() + " " + anno.val());
  } catch (NoSuchMethodException exc) {
     System.out.println("Метод не найден.");
  }
}

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

Вот как выглядит результат работы этой программы:

```
Пример аннотации 100
```

Эта программа использует рефлексию, как описано, чтобы получить и отобразить значения `str` и `val` в аннотации `MyAnno`, ассоциированной с `myMeth()` в классе `Meta`. Есть две вещи, на которые следует обратить особое внимание. Первая — выражение `MyAnno.class` в строке:

```
MyAnno anno = m.getAnnotation(MyAnno.class);
```

Это выражение вычисляется как объект `Class` типа `MyAnno` — аннотация. Это называется *литералом класса*. Вы можете использовать выражения этого типа всякий раз, когда требуется объект `Class` известного класса. Например, следующий оператор служит для получения объекта `Class` для `Meta`:

```
Class c = Meta.class;
```

Конечно, такой подход работает, только когда вы знаете имя класса объекта заранее, что не всегда возможно. Вообще вы можете получать литерал класса для классов, интерфейсов, примитивных типов и массивов.

Второй интересный момент — это способ, которым значения, ассоциированные с `str` и `val`, получаются, когда они выводятся в следующей строке:

```
System.out.println(anno.str() + " " + anno.val());
```

Обратите внимание, что они вызываются с применением синтаксиса вызова методов. Тот же подход используется всякий раз, когда требуется получить член аннотации.

Второй пример применения рефлексии

В предыдущем примере `myMeth()` не имел параметров. То есть когда вызывался `getMethod()`, передавалось только имя `myMeth`. Однако для того, чтобы получить метод, который имеет параметры, вы должны специфицировать объекты класса, представляющие типы этих параметров, в виде аргументов `getMethod()`. Например, ниже показана слегка измененная версия предыдущей программы.

```
import java.lang.annotation.*;
import java.lang.reflect.*;
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
   String str();
   int val();
}
class Meta {
// myMeth теперь имеет 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);
  }
  }
```

Результат работы этой версии будет таким:

```
Два параметра 19
```

В этой версии `myMeth()` принимает параметры `String` и `int`. Чтобы получить информацию об этом методе, `getMethod()` должен быть вызван следующим образом:

```
Method m = c.getMethod("myMeth", String.class, int.class);
```

Здесь объекты `Class`, представляющие `String` и `int`, передаются в виде дополнительных аргументов.

Получение всех аннотаций

Вы можете получить сразу все аннотации, имеющие `@Retention`, равную `RUNTIME`, которые ассоциированы с позицией, вызвав `getAnnotations()` для этой позиции. Общая форма этого метода выглядит так:

```
Annotation[] getAnnotations()
```

Он возвращает массив аннотаций. `getAnnotations()` может быть вызван для объектов типа `Class`, `Method`, `Constructor` и `Field`.

Вот еще один пример с рефлексией, который показывает, как получить все аннотации, ассоциированные с классом и методом. Он объявляет две аннотации. Затем он использует их 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();
     // Отобразить все аннотации для Meta2.
     System.out.println("Все аннотации для Meta2:");
```

```
      for(Annotation a : annos)
        System.out.println(a);
      System.out.println();
      // Отобразить все аннотации для myMeth.
      Method m = ob.getClass( ).getMethod("myMeth");
      annos = m.getAnnotations();
      System.out.println("Все аннотации для 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 показан результат работы этой программы:

```
Все аннотации для Meta2:
@What(description=Аннотация тестового класса)
@MyAnno(str=Meta2, val=99)

Все аннотации для myMeth:
@What(description=Аннотация тестового метода)
@MyAnno(str=Testing, val=100)
```

Эта программа использует `getAnnotations()` для получения массива всех аннотаций, ассоциированных с классом `Meta2` и методом `myMeth()`. Как объяснялось, `getAnnotations()` возвращает массив объектов `Annotation`. Вспомните, что `Annotation` — это суперинтерфейс для всех интерфейсов аннотаций, и что он переопределяет `toString()` из класса `Object`. То есть когда выводится ссылка на `Annotation`, вызывается его метод `toString()` для генерации строки, описывающей аннотацию, что и демонстрирует предыдущий пример.

Интерфейс `AnnotatedElement`

Методы `getAnnotation()` и `getAnnotations()`, использованные в предыдущем примере, определены интерфейсом `AnnotatedElement`, который определен в `java.lang.reflect`. Этот интерфейс поддерживает рефлексию для аннотации и реализован классами `Method`, `Field`, `Constructor`, `Class` и `Package`.

В дополнение к `getAnnotation()` и `getAnnotations()` интерфейс `AnnotatedElement` определяет два других метода. Первый из них — `getDeclaredAnnotations()`, который имеет следующую общую форму:

```
Annotation[] getDeclaredAnnotations()
```

Он возвращает не унаследованные аннотации, представленные в вызывающем объекте. Второй — это `isAnnotationPresent()`, имеющий такую форму:

```
boolean isAnnotationPresent(Class annoType)
```

Он возвращает `true`, если аннотация, специфицированная в *`annoType`*, ассоциирована с вызывающим объектом. В противном случае возвращает `false`.

На заметку! *Методы* `getAnnotation()` *и* `isAnnotationPresent()` *используют новое средство обобщений (generics) для гарантии безопасности типов. Поскольку обобщения не обсуждаются вплоть до главы 14, их сигнатура представлена в настоящей главе в черновой форме.*

Использование значений по умолчанию

Вы можете дать членам аннотации значения по умолчанию, которые будут использоваться, когда применяется аннотация. Значение по умолчанию указывается добавлением слова `default` к объявлению члена. Это выглядит следующим образом:

```
type member() default value;
```

Здесь *value* должно иметь тип, совместимый с *type*.

Вот как выглядит `@MyAnno`, переписанная с использованием значений по умолчанию:

```
// Объявление типа аннотации, включающее значения по умолчанию.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
   String str() default "Тестирование";
   int val() default 9000;
}
```

Это объявление определяет значение по умолчанию `"Тестирование"` для члена `str` и `9000` — для `val`. Это означает, что ни одно их значений не обязательно указывать при использовании `@MyAnno`. Однако любому из них или обоим сразу можно присвоить значение при необходимости. Таким образом, существуют четыре способа применения `@MyAnno`:

```
@MyAnno() // значения str и val принимаются по умолчанию
@MyAnno(str = "некоторая строка")           // val — по умолчанию
@MyAnno(val = 100)                          // str — по умолчанию
@MyAnno(str = "Тестирование", val = 100)    // нет умолчаний
```

В следующей программе демонстрируется использование значений по умолчанию в аннотациях.

```
import java.lang.annotation.*;
import java.lang.reflect.*;
// Объявление типа аннотаций с включением значений по умолчанию.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
   String str() default "Тестирование";
   int val() default 9000;
}
class Meta3 {
// Аннотирование метода с использованием значений по умолчанию.
@MyAnno()
public static void myMeth() {
  Meta3 ob = new Meta3();
  // Получить аннотацию к методу
  // и отобразить 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();
  }
  }
```

Вывод будет таким:

```
Тестирование 9000
```

Аннотация-маркер

Аннотация-*маркер* — это специальный вид аннотаций, который не содержит членов. Его единственное назначение — пометить (маркировать) объявление. То есть его присутствие как аннотации существенно. Лучший способ определить, присутствует ли аннотация-маркер — воспользоваться методом `isAnnotationPresent()`, который определен в интерфейсе `AnnotatedElement`.

Рассмотрим пример использования аннотации-маркера. Поскольку такая аннотация не имеет членов, важно просто определить — присутствует она или нет.

```
import java.lang.annotation.*;
import java.lang.reflect.*;
// Аннотация-маркер.
@Retention(RetentionPolicy.RUNTIME)
@interface MyMarker { }
class Marker {
// Аннотирование метода с помощью маркера.
// Обратите внимание на необходимость скобок ().
@MyMarker
public static void myMeth() {
  Marker ob = new Marker();
  try {
     Method m = ob.getClass().getMethod("myMeth");
     // Определение наличия аннотации.
     if(m.isAnnotationPresent(MyMarker.class))
       System.out.println("MyMarker присутствует.");
  } catch (NoSuchMethodException exc) {
     System.out.println("Метод не найден.");
  }
}
public static void main(String args[]) {
  myMeth();
}
}
```

Показанный ниже вывод подтверждает наличие `@MyMarker`:

```
MyMarker присутствует.
```

Обратите внимание, что нет необходимости после `@MyMarker` указывать скобки. То есть `@MyMarker` применяется просто использованием ее имени:

```
@MyMarker
```

Не будет ошибкой указать пустые скобки, однако в этом нет необходимости.

Одночленные аннотации

Одночленная аннотация содержит, как должно быть понятно, только один член. Она работает подобно обычной аннотации за исключением того, что допускает сокращенную форму спецификации значения члена. Когда присутствует только один член, вы можете просто специфицировать его значение, когда аннотация применяется, вы не обязаны указывать имя члена. Однако для того, чтобы использовать это сокращение, член должен иметь имя `value`.

Ниже показан пример создания и использование одночленной аннотации.

```
import java.lang.annotation.*;
import java.lang.reflect.*;
// Одночленная аннотация.
@Retention(RetentionPolicy.RUNTIME)
@interface MySingle {
   int value(); // именем переменной должно быть value
}
class Single {
// Аннотирование метода 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()); // отображает 100
  } catch (NoSuchMethodException exc) {
    System.out.println("Метод не найден.");
  }
}
public static void main(String args[]) {
  myMeth();
}
}
```

Как и ожидалось, эта программа отображает значение 100. Здесь `@MySingle` применяется для аннотирования `myMeth()`, как показано ниже:

```
@MySingle(100)
```

Обратите внимание, что `value =` не должно быть указано.

Вы можете применять синтаксис одночленных аннотаций и при использовании аннотаций с другими членами, но все остальные члены должны иметь значения по умолчанию. Например, ниже добавляется член `xyz` со значением по умолчанию, равным 0.

```
@interface SomeAnno {
   int value();
   int xyz() default 0;
}
```

В случаях, когда вы хотите использовать значение по умолчанию для `xyz`, вы можете применить `@SomeAnno`, как показано ниже, просто указав значение для `value` с использованием синтаксиса одночленных аннотаций:

```
@SomeAnno(88)
```

В этом случае xyz по умолчанию принимает значение 0, а value — 88. Конечно, чтобы специфицировать другое значение для xyz, необходимо, чтобы оба члена были инициированы явно:

```
@SomeAnno(value = 88, xyz = 99)
```

Помните, что когда вы применяете одночленные аннотации, именем члена должно быть value.

Встроенные аннотации

В Java определено очень много встроенных аннотаций. Большинство из них специализированы, но семь имеют общее назначение. Четыре из них импортируются из java.lang.annotation: @Retention, @Documented, @Target и @Inherited. Три других — @Override, @Deprecated и @SuppressWarnings — включены в java.lang. Каждая из них описана ниже.

@Retention

@Retention предназначена для применения только в качестве аннотации к другим аннотациям. Определяет политику удержания, как было описано в настоящей главе.

@Documented

@Documented — это маркер-интерфейс, который сообщает инструменту, что аннотация должна быть документирована. Он предназначен для использования только в качестве аннотации к объявлению аннотации.

@Target

@Target — аннотация, специфицирующая типы объявлений, к которым может быть применима аннотация. Предназначена для использования только в качестве аннотации к другим аннотациям. @Target принимает один аргумент, который должен быть константой из перечисления ElementType. Этот аргумент специфицирует типы объявлений, к которым может быть применена аннотация. Эти константы описаны в табл. 12.1 вместе с типами объявлений, к которым они относятся.

Таблица 12.1. Константы из перечисления ElementType

Целевая константа	Аннотация может быть приложена
ANNOTATION_TYPE	Другая аннотация
CONSTRUCTOR	Конструктор
FIELD	Поле
LOCAL_VARIABLE	Локальная переменная
METHOD	Метод
PACKAGE	Пакет
PARAMETER	Параметр
TYPE	Класс, интерфейс или перечисление

Вы можете специфицировать одно или более из этих значений в аннотации `@Target`. Чтобы указать множественные значения, вы должны поместить их внутрь ограниченного фигурными скобками списка. Например, чтобы специфицировать, что аннотация применима только к полям и локальным переменным, нужно использовать следующую аннотацию `@Target`:

```
@Target( { ElementType.FIELD, ElementType.LOCAL_VARIABLE } )
```

@Inherited

`@Inherited` — аннотация-маркер, которая может применяться в другом объявлении аннотации. Более того, она касается только тех аннотаций, что будут использоваться в объявлениях классов. `@Inherited` позволяет аннотации суперкласса быть унаследованной в подклассе. Таким образом, когда осуществляется запрос к подклассу на предмет специфической аннотации, то если у этой аннотации в подклассе нет, проверяется суперкласс. Если запрошенная аннотация присутствует у суперкласса и она аннотирована как `@Inherited`, то эта аннотация будет возвращена.

@Override

`@Override` — аннотация-маркер, которая может применяться только в методах. Метод, аннотированный как `@Override`, должен переопределять метод суперкласса. Если он этого не делает, в результате возникает ошибка времени компиляции. Она используется для обеспечения того, что метод суперкласса будет действительно переопределен, а не просто перегружен.

@Deprecated

`@Deprecated` — аннотация-маркер. Она указывает, что объявление устарело и должно быть заменено более новой формой.

@SuppressWarnings

`@SuppressWarnings` — указывает, что одно или более предупреждений, которые могут быть выданы компилятором, следует подавить. Подавляемые предупреждения специфицируются именами в строковой форме. Эта аннотация может быть применима к объявлениям любого типа.

Некоторые ограничения

Существует некоторое количество ограничений, касающихся объявления аннотаций. Во-первых, одна аннотация не может наследовать другую. Во-вторых, все методы, объявленные в аннотации, должны не принимать параметров. Более того, они должны возвращать один из перечисленных ниже типов:

- примитивный тип, такой как `int` или `double`;
- объект типа `String` или `Class`;
- тип `enum`;
- тип другой аннотации;
- массив одного из предыдущих типов.

Аннотации не могут быть обобщенными. Другими словами, они не могут принимать параметры-типы. (Обобщения рассматриваются в главе 14.) И, наконец, в методах аннотации не может быть указана конструкция `throws`.

13
ГЛАВА

Ввод-вывод, аплеты и другие темы

Настоящая глава посвящена двум из наиболее важных пакетов Java: `io` и `applet`. Пакет `io` поддерживает базовую систему ввода-вывода Java, включая файловый ввод-вывод. Пакет `applet` поддерживает аплеты. Поддержка ввода-вывода и аплетов осуществляется ядром библиотек программного интерфейса (API), а не ключевыми словами языка. По этой причине углубленное обсуждение этих тем содержится во второй части книги, где рассматриваются классы API. В этой главе описаны основы этих двух подсистем, чтобы вы смогли увидеть, как они интегрированы в язык Java и как они встроены в общий контекст программирования на Java и ее исполняющей системы. В этой главе также рассматриваются последние из ключевых слов Java: `transient`, `volatile`, `instanceof`, `native`, `strictfp` и `assert`. Завершает главу описание статического импорта и применения ключевого слова `this`.

Основы ввода-вывода

Как вы могли заметить, читая предыдущую главу 12, до сих пор в примерах программ было задействовано не так много операций ввода-вывода. Фактически помимо `print()` и `println()`, никаких методов ввода-вывода в общем-то и не применялось. Причина этого проста: большинство реальных Java-приложений не являются текст-ориентированными консольными программами. Вместо этого они являются графически-ориентированными программами, которые основаны на Abstract Window Toolkit (AWT) или Swing для взаимодействия с пользователем. Хотя текст-ориентированные программы великолепны в качестве учебных примеров, они не занимают сколько-нибудь значительную часть в мире реальных программ. К тому же поддержка консольного ввода-вывода в Java ограничена и не слишком удобна в использовании — даже для простейших программ. Текстовый консольный ввод-вывод не является важным при программировании на Java.

Тем не менее, Java обеспечивает мощную и гибкую поддержку ввода-вывода, когда это касается файлов и сетей. Система ввода-вывода Java целостна и последовательна. Фактически, если однажды разобраться с ее базовыми принципами, все остальное для профессионала становится простым.

Потоки

Java-программы создают потоки ввода-вывода. *Поток* (stream) — это абстракция, которая либо порождает, либо принимает информацию. Поток связан с физическим устройством с помощью системы ввода-вывода Java. Все потоки ведут себя на один манер, даже несмотря на то, что реальные физические устройства, к которым они подключены, отличаются друг от друга. Таким образом, одни и те же классы и методы ввода-вывода применимы к устройствам разного типа. Это означает, что абстракция входного потока может охватить разные типы ввода: из дискового файла, клавиатуры или сетевого сокета. Аналогично выходной поток может ссылаться на консоль, дисковый файл или сетевое подключение. Потоки — это ясный способ обращения с вводом-выводом без необходимости для вашего кода разбираться с разницей, например, между клавиатурой и сетью. Java реализует потоки внутри иерархии классов, определенных в пакете `java.io`.

Байтовые и символьные потоки

Java определяет два типа потоков: байтовые и символьные. *Байтовые потоки* предоставляют удобные средства для управления вводом и выводом байтов. Байтовые потоки используются, например, при чтении и записи бинарных данных. *Символьные потоки* предлагают удобные возможности управления вводом и выводом символов. Они используют кодировку Unicode и, таким образом, могут быть интернационализированы. Кроме того, в некоторых случаях символьные потоки более эффективны, чем байтовые.

Исходная версия Java (Java 1.0) не включала символьных потоков, и потому весь ввод-вывод был байт-ориентированным. Символьные потоки были добавлены в Java 1.1, и при этом некоторые байт-ориентированные классы и методы устарели. Вот почему старый код, к котором не используются символьные потоки, должен быть, где возможно, обновлен, чтобы воспользоваться их преимуществами.

Еще один момент: на самом низком уровне весь ввод-вывод по-прежнему байт-ориентирован. Символьные потоки просто предлагают удобные и эффективные средства управления символами.

Обзор байт-ориентированных и символ-ориентированных потоков представлен в следующих разделах.

Классы байтовых потоков

Байтовые потоки определены в двух иерархиях классов. На вершине находятся абстрактные классы `InputStream` и `OutputStream`. Каждый из этих абстрактных классов имеет несколько реальных подклассов, которые управляют различиями между различными устройствами, такими как дисковые файлы, сетевые подключения и даже буферы памяти. Классы байтовых потоков перечислены в табл. 13.1. Некоторые из этих классов описываются ниже в разделе, а другие — во второй части. Помните, что для использования потоковых классов необходимо импортировать `java.io`.

Абстрактные классы `InputStream` и `OutputStream` определяют несколько ключевых методов, которые реализуют другие потоковые классы. Два наиболее важных — это `read()` и `write()`, которые, соответственно, читают и пишут байты данных. Оба метода объявлены как абстрактные внутри `InputStream` и `OutputStream`. В классах-наследниках они переопределяются.

Классы символьных потоков

Символьные потоки также определены в двух иерархиях классов. На их вершине находятся два абстрактных класса: `Reader` и `Writer`. Эти абстрактные классы управляют потоками символов Unicode. В Java предусмотрено несколько конкретных подклассов для каждого из них. Классы символьных потоков перечислены в табл. 13.2.

Абстрактные классы `Reader` и `Writer` определяют несколько ключевых методов, которые реализуют другие потоковые классы. Два наиболее важных — это `read()` и `write()`, которые, соответственно читают и пишут символьные данные. Эти методы переопределяются в потоковых классах-наследниках.

Таблица 13.1. Классы байтовых потоков

Потоковый класс	Назначение
`BufferedInputStream`	Буферизированный входной поток.
`BufferedOutputStream`	Буферизированный выходной поток.
`ByteArrayInputStream`	Входной поток, читающий из массива байт.
`ByteArrayOutputStream`	Выходной поток, записывающий в массив байт.
`DataInputStream`	Входной поток, включающий методы для чтения стандартных типов данных Java.
`DataOutputStream`	Выходной поток, включающий методы для записи стандартных типов данных Java.
`FileInputStream`	Входной поток, читающий из файла.
`FileOutputStream`	Выходной поток, записывающий в файл.
`FilterInputStream`	Реализация `InputStream`.
`FilterOutputStream`	Реализация `OutputStream`.
`InputStream`	Абстрактный класс, описывающий поток ввода.
`ObjectInputStream`	Входной поток для объектов.
`ObjectOutputStream`	Выходной поток для объектов.
`OutputStream`	Абстрактный класс, описывающий поток вывода.
`PipedInputStream`	Входной канал (например, межпрограммный).
`PipedOutputStream`	Выходной канал.
`PrintStream`	Выходной поток, включающий `print()` и `println()`.
`PushbackInputStream`	Входной поток, поддерживающий однобайтовый возврат во входной поток.
`RandomAccessFile`	Поддерживает файловый ввод-вывод с произвольным доступом.
`SequenceInputStream`	Входной поток, представляющий собой комбинацию двух и более входных потоков, которые читаются совместно — один после другого.

Таблица 13.2. Классы символьных потоков

Потоковый класс	Назначение
BufferedReader	Буферизованный входной символьный поток.
BufferedWriter	Буферизованный выходной символьный поток.
CharArrayReader	Входной поток, который читает из символьного массива.
CharArrayWriter	Выходной поток, который пишет в символьный массив.
FileReader	Входной поток, читающий файл.
FileWriter	Выходной поток, пишущий в файл.
FilterReader	Фильтрующий читатель.
FilterWriter	Фильтрующий писатель.
InputStreamReader	Входной поток, транслирующий байты в символы.
LineNumberReader	Входной поток, подсчитывающий строки.
OutputStreamWriter	Выходной поток, транслирующий байты в символы.
PipedReader	Входной канал.
PipedWriter	Выходной канал.
PrintWriter	Выходной поток, включающий `print()` и `println()`.
PushbackReader	Входной поток, позволяющий возвращать символы обратно в поток.
Reader	Абстрактный класс, описывающий символьный ввод.
StringReader	Входной поток, читающий из строки.
StringWriter	Выходной поток, пишущий в строку.
Writer	Абстрактный класс, описывающий символьный вывод.

Предопределенные потоки

Как вы знаете, все Java-программы автоматически импортируют пакет `java.lang`. В этом пакете определен класс `System`, инкапсулирующий некоторые аспекты среды времени выполнения. Например, используя некоторые из его методов, можно получить текущее время и настройки различных параметров, ассоциированных с системой. `System` также содержит три предопределенных потоковых переменных: `in`, `out` и `err`. Эти переменные объявлены как `public`, `static` и `final` в классе `System`. Это значит, что они могут быть использованы любой другой частью вашей программы без обращения к специфическому объекту `System`.

`System.out` ссылается на стандартный выходной поток. По умолчанию это консоль. `System.in` ссылается на стандартный входной поток, который также по умолчанию является консолью. `System.err` ссылается на стандартный поток ошибок, который также по умолчанию связан с консолью. Однако эти потоки могут быть перенаправлены на любое совместимое устройство ввода-вывода.

`System.in` — это объект типа `InputStream`, `System.out` и `System.err` — объекты типа `PrintStream`. Это байтовые потоки, хотя обычно они используются для чтения и записи символов с консоли и на консоль. Как вы увидите, при необходимости их можно поместить в оболочки символьных потоков.

В примерах, приведенных в предыдущих главах, использовался поток `System.out`. Вы можете почти таким же образом применять `System.err`. Как будет показано в следующем разделе, использование `System.in` немного сложнее.

Чтение консольного ввода

В Java 1.0 единственным способом выполнения консольного ввода было использование байтового потока, и существует большой объем старого кода, в котором применяется этот подход. Сегодня применение байтового потока для чтения консольного ввода по-прежнему технически возможно, но поступать так не рекомендуется. Предпочтительный метод чтения консольного ввода — это использовать символ-ориентированный поток, что значительно упрощает возможности интернационализации и поддержки разрабатываемых программ.

В Java консольный ввод выполняется чтением `System.in`. Чтобы получить символьный поток, присоединенный к консоли, вы должны поместить `System.in` в оболочку объекта `BufferedReader`. `BufferedReader` поддерживает буферизованный входной поток. Наиболее часто используемый его конструктор выглядит так:

```
BufferedReader(Reader inputReader)
```

Здесь *`inputReader`* — это поток, который связывается с создаваемым экземпляром `BufferedReader`. `Reader` — абстрактный класс. Одним из его конкретных наследников является `InputStreamReader`, который преобразует байты в символы. Для получения объекта `InputStreamReader`, который присоединен к `System.in`, служит следующий конструктор:

```
InputStreamReader(InputStream inputStream)
```

Поскольку `System.in` ссылается на объект типа `InputStream`, он должен быть использован как параметр *`inputStream`*. Собрав все вместе, получим следующую строку кода, которая создает `BufferedReader`, соединенный с клавиатурой:

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

После выполнения этого оператора `br` представляет собой основанный на символах поток, подключенный к консоли через `System.in`.

Чтение символов

Для чтения символа из `BufferedReader` применяется `read()`. Ниже показана версия `read()`, которая будет использоваться:

```
int read() throws IOException
```

Каждый раз, когда вызывается метод `read()`, он читает символ из входного потока и возвращает его как целое значение. При достижении конца потока возвращается `-1`. Как видите, метод может возбудить исключение `IOException`.

В следующей программе демонстрируется применение `read()`, читая символы с консоли до тех пор, пока не пользователь не введет “q”. Обратите внимание, что любые исключения ввода-вывода, которые могут быть сгенерированы, просто передаются в `main()`. Такой подход распространен при чтении с консоли, но при желании вы можете обработать ошибки такого рода самостоятельно.

```
// Использование 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("Вводите символы, 'q' — для выхода.");

  // читать символы
  do {
     c = (char) br.read();
     System.out.println(c);
  } while(c != 'q');
}
}
```

Ниже показан пример запуска этой программы:

```
Вводите символы, 'q' — для выхода.
123abcq
1
2
3
a
b
c
q
```

Этот вывод может выглядеть немного не так, как вы ожидали, потому что `System.in` является строчно-буферизованным по умолчанию. Это значит, что никакого ввода в действительности программе не передается до тех пор, пока не будет нажата клавиша <ENTER>. Как можно предположить, это делает `read()` лишь отчасти применимым для интерактивного консольного ввода.

Чтение строк

Чтобы прочесть строку с клавиатуры, используйте версию метода `readLine()`, который является членом класса `BufferedReader`. Его общая форма такова:

```
String readLine() throws IOException
```

Как видите, он возвращает объект `String`.

Следующая программа демонстрирует `BufferedReader` и метод `readLine()`. Программа читает и отображает строки текста до тех пор, пока вы не введете слово “стоп”:

```
// Чтение строк с консоли с применением BufferedReader.
import java.io.*;

class BRReadLines {
public static void main(String args[]) throws IOException
{
  // Создать BufferedReader с использованием System.in
  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
  String str;
  System.out.println("Вводите строки текста.");
```

```
  System.out.println("Введите 'стоп' для завершения.");
  do {
     str = br.readLine();
     System.out.println(str);
  } while(!str.equals("стоп"));
}
}
```

В следующем примере создается крошечный текстовый редактор. В коде создается массив объектов `String` и затем читаются строки текста с сохранением каждой строки в виде элемента массива. Чтение производится до 100 строк или до того, как будет введено слово “стоп”. Для чтения с консоли используется `BufferedReader`.

```
// Крошечный редактор.
import java.io.*;
class TinyEdit {
public static void main(String args[]) throws IOException
{
  // Создать BufferedReader, используя System.in
  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
  String str[] = new String[100];
  System.out.println("Вводите 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("\nВот ваш файл:");

  // отобразить строки
  for(int i=0; i<100; i++) {
     if(str[i].equals("стоп")) break;
     System.out.println(str[i]);
  }
}
}
```

Ниже показан пример запуска этой программы:

```
Вводите строки текста.
Введите 'стоп' для завершения.
Это строка один.
Это строка два.
Java делает работу со строками простой.
Просто создайте объект String.
стоп
Вот ваш файл:
Это строка один.
Это строка два.
Java делает работу со строками простой.
Просто создайте объект String.
```

Запись консольного вывода

Консольный вывод проще всего осуществлять с помощью методов `print()` и `println()`, описанных ранее, которые используются в большинстве примеров этой книги. Эти методы определены в классе `PrintStream` (который является типом объекта `System.out`). Даже несмотря на то, что `System.out` — байтовый поток, применение его для вывода в простых программах вполне оправдано. Тем не менее, в следующем разделе описана символ-ориентированная альтернатива.

Поскольку `PrintStream` — выходной поток, унаследованный от `OutputStream`, он также реализует низкоуровневый метод `write()`. То есть `write()` может применяться для записи на консоль. Простейшая форма `write()`, определенного в `PrintStream`, показана ниже:

```
void write(int byteval)
```

Этот метод пишет в поток байт, переданный в *byteval*. Хотя параметр *byteval* объявлен как целочисленный, записываются только 8 его младших бит. Вот короткий пример, использующий `write()` для вывода буквы "A" с последующим переводом строки на экран:

```
// Демонстрация System.out.write().
class WriteDemo {
public static void main(String args[]) {
  int b;
  b = 'A';
  System.out.write(b);
  System.out.write('\n');
}
}
```

Вам не часто придется использовать `write()` для вывода на консоль (хотя в некоторых ситуациях это и удобно), поскольку значительно проще применять для этого `print()` и `println()`.

Класс `PrintWriter`

Хотя применение `System.out` для вывода на консоль допустимо, он рекомендуется в основном для целей отладки или для примеров программ вроде тех, что приводятся в настоящей книге. Для реальных программ рекомендуемым способом записи на консоль при использовании Java является поток `PrintWriter`. `PrintWriter` — это один из классов, основанных на символах. Применение такого класса для консольного вывода упрощает интернационализацию ваших программ.

`PrintWriter` определяет несколько конструкторов. Один из тех, которые мы будем использовать, показан ниже:

```
PrintWriter(OutputStream outputStream, boolean flushOnNewline)
```

Здесь *outputStream* — объект типа `OutputStream`, а *flushOnNewline* управляет тем, будет ли Java сбрасывать буфер в выходной поток каждый раз при вызове метода `println()`. Если *flushOnNewline* равно `true`, то происходит автоматический сброс буфера, если же `false`, то это автоматически не делается.

PrintWriter поддерживает методы print() и println() для всех типов, включая Object. То есть вы можете использовать эти методы таким же способом, как они применяются в System.out. Если аргумент не простого типа, то PrintWriter вызывает метод toString() и затем печатает результат.

Чтобы писать на консоль с помощью PrintWriter, специфицируйте System.out в качестве выходного потока и сбрасывайте поток после каждого символа новой строки. Например, следующая строка кода создает PrintWriter, который подключен к консольному выводу:

```
PrintWriter pw = new PrintWriter(System.out, true);
```

Показанное ниже приложение иллюстрирует применение PrintWriter для управления консольным выводом:

```
// Демонстрация 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);
   }
}
```

Вывод этой программы будет выглядеть следующим образом:

```
Это строка
-7
4.5E-7
```

Помните, что нет ничего неправильного в применении System.out для простого текстового вывода на консоль, когда вы изучаете Java или занимаетесь отладкой своих программ. Однако PrintWriter обеспечит возможность простой интернационализации для реальных программ. Поскольку никаких выгод от использования PrintWriter в простых программах нет, мы продолжим пользоваться System.out для вывода на консоль.

Чтение и запись файлов

Java предоставляет множество классов и методов, которые позволяют вам читать и записывать файлы. В Java все файлы байт-ориентированы, и Java предоставляет методы для чтения и записи байтов в файл. Однако Java позволяет также поместить байт-ориентированные файловые потоки в оболочки символ-ориентированных объектов. Такая техника описана во второй части книги. В настоящей же главе рассматриваются только основы файлового ввода-вывода.

Два из наиболее часто используемых потоковых класса — это FileInputStream и FileOutputStream, которые создают байтовые потоки, связанные с файлами. Чтобы открыть файл, вы просто создаете объект одного из этих классов, указав имя файла в качестве аргумента конструктора. Хотя оба класса имеют и дополнительные переопределенные конструкторы, мы будем использовать только следующие из них:

```
FileInputStream(String fileName) throws FileNotFoundException
FileOutputStream(String fileName) throws FileNotFoundException
```

Здесь *filename* — имя файла, который вы хотите открыть. Когда вы создаете входной поток, то если файл не существовал, возбуждается исключение FileNotFoundException. Для выходных потоков, если файл не может быть создан, также возбуждается исключение FileNotFoundException. Когда выходной файл открыт, любой ранее существовавший файл с тем же именем уничтожается.

Когда вы завершаете работу с файлом, вы должны закрыть его вызовом метода close(). Этот метод определен и в FileInputStream и в FileOutputStream, как показано ниже:

```
void close() throws IOException
```

Чтобы читать файл, вы можете применять версию метода read(), который определен в FileInputStream. Та, что мы будем использовать, выглядит так:

```
int read() throws IOException
```

Всякий раз, когда вызывается этот метод, он читает единственный байт из файла и возвращает его как целое число. read() возвращает -1, когда достигнут конец файла. Метод может возбуждать исключение IOException.

В следующей программе read() используется для ввода и отображения содержимого текстового файла, имя которого специфицировано в аргументе командной строки. Обратите внимание на блок try/catch, обрабатывающий две ошибки, которые могут возникнуть при работе программы — когда указанный файл не найден, либо когда пользователь забыл указать имя файла. Вы можете применять тот же подход всякий раз при использовании аргументов командной строки. Другие возможные исключения ввода-вывода просто передаются в main(), что вполне приемлемо для такого простого примера. Однако, работая с файлами, часто вы пожелаете обработать самостоятельно все исключения ввода-вывода.

```
/* Отображение текстового файла.
   Чтобы использовать эту программу, укажите
   имя файла, который хотите просмотреть.
   Например, чтобы просмотреть файл TEST.TXT,
   используйте следующую командную строку:

   java ShowFile TEST.TXT

*/

import java.io.*;

class ShowFile {
public static void main(String args[])
throws IOException
{
  int i;
  FileInputStream fin;

  try {
     fin = new FileInputStream(args[0]);
  } catch(FileNotFoundException e) {
     System.out.println("Файл не найден");
     return;
  } catch(ArrayIndexOutOfBoundsException e) {
     System.out.println("Использование: ShowFile Файл");
     return;
  }
```

```
    // читать символы до получения символа EOF (конец файла)
    do {
      i = fin.read();
      if(i != -1) System.out.print((char) i);
    } while(i != -1);
    fin.close();
  }
  }
```

Для записи в файл вы будете использовать метод `write()`, определенный в `FileOutputStream`. Его простейшая форма выглядит так:

```
void write(int byteval) throws IOException
```

Этот метод пишет в файл байт, переданный в *`byteval`*. Хотя *`byteval`* объявлен как целочисленный, в файл записываются только его младшие восемь бит. Если при записи произойдет ошибка, возбуждается исключение `IOException`. В следующем примере `write()` используется для копирования текстового файла:

```
/* Копирование текстового файла.
   Для использования этой программы укажите
   имена исходного и целевого файлов.
   Например, чтобы скопировать файл FIRST.TXT в файл
   SECOND.TXT, используйте следующую командную строку:
   java CopyFile FIRST.TXT SECOND.TXT
*/
import java.io.*;
class CopyFile {
public static void main(String args[]) throws IOException
{
  int i;
  FileInputStream fin;
  FileOutputStream fout;
  try {
    // открыть входной файл
    try {
      fin = new FileInputStream(args[0]);
    } catch(FileNotFoundException e) {
      System.out.println("Входной файл не найден");
      return;
    }
    // открыть выходной файл
    try {
       fout = new FileOutputStream(args[1]);
    } catch(FileNotFoundException e) {
       System.out.println("Ошибка открытия выходного файла");
       return;
    }
  } catch(ArrayIndexOutOfBoundsException e) {
    System.out.println("Использование: CopyFile Исходный Целевой");
    return;
  }
  // Копировать файл
  try {
    do {
      i = fin.read();
```

```
        if(i != -1) fout.write(i);
      } while(i != -1);
    } catch(IOException e) {
      System.out.println("Ошибка файла");
    }
    fin.close();
    fout.close();
  }
  }
```

Обратите внимание на способ обработки потенциальных ошибок ввода-вывода в этой программе. В отличие от некоторых других языков программирования, включая C и C++, которые используют коды ошибок для обнаружения файловых ошибок, Java применяет свой механизм исключений. Это не только делает управление файлами понятнее, но также позволяет Java просто отличать условие достижения конца файла от файловых ошибок во время ввода. В C/C++ многие функции ввода возвращают одно и то же значение, когда происходит ошибка и когда достигается конец файла. (То есть в C/C++ условие EOF часто отображается на то же значение, что и ошибка ввода.) Обычно это означает, что программист обязан включать дополнительные операторы для определения того, какое событие на самом деле произошло. В Java ошибки передаются вашей программе в виде исключений, а не через значение, возвращаемое `read()`. То есть, когда `read()` возвращает `-1`, это значит только одно: достигнут конец файла.

Основы организации аплетов

Все предшествующие примеры программ в этой книге были консольными приложениями Java. Однако приложения этого типа — только один класс Java-программ. Другой тип программ Java — аплеты. Как упоминалось в главе 1, *аплет* — это маленькое приложение, которое находится на Internet-сервере, транспортируется по Internet, автоматически инсталлируется и запускается как часть Web-документа. После того, как аплет появляется у клиента, он получает ограниченный доступ к ресурсам так, что он обеспечивает сложный графический пользовательский интерфейс и выполняет сложные вычисления, не подвергая клиента риску вирусной атаки или повреждения целостности его данных.

Многие случаи, связанные с созданием и применением аплетов вы можете найти во второй части книги, где рассматривается пакет `applet`. Однако основы, имеющие отношение к созданию аплетов, мы рассмотрим прямо сейчас, поскольку аплеты имеют структуру, отличную от программ, с которыми мы имели дело до сих пор в этой книге. Как вы увидите, аплеты отличаются от приложений по нескольким ключевым признакам.

Начнем с простейшего аплета:

```
import java.awt.*;
import java.applet.*;
public class SimpleApplet extends Applet {
   public void paint(Graphics g) {
      g.drawString("Простейший аплет", 20, 20);
   }
}
```

Этот аплет начинается с двух операторов `import`. Первое импортирует классы Abstract Window Toolkit (AWT). Аплеты взаимодействуют с пользователем (непосредственно или опосредованно) через AWT, а не через классы консольного ввода-вывода. Как вы можете предположить, AWT значительно больше и сложнее, и полное обсуждение его возможно-

стей занимает несколько глав во второй части настоящей книги. К счастью, этот простой аплет очень ограниченно использует AWT. (Аплеты также могут использовать Swing для предоставления графического пользовательского интерфейса, но этот подход рассматривается далее в книге.) Второй оператор `import` импортирует пакет `applet`, в котором находится класс `Applet`. Каждый аплет, который вы создадите, должен быть подклассом класса `Applet`.

Следующая строка в программе объявляет класс `SimpleApplet`. Этот класс должен быть объявлен как `public`, чтобы быть доступным коду вне нашей программы.

Внутри `SimpleApplet` объявлен метод `paint()`. Этот метод определен AWT и должен быть переопределен нашим аплетом. `paint()` вызывается всякий раз, когда аплет должен перерисовать свой вывод. Эта ситуация может быть порождена несколькими причинами. Например, окно, в котором аплет запущен, может быть перекрыто другим окном, а затем вновь открыто. Или же окно аплета может быть минимизировано, а затем восстановлено. `paint()` также вызывается, когда аплет начинает выполнение. Независимо от причины, всякий раз, когда аплет должен перерисовать свое содержимое, вызывается `paint()`. Метод `paint()` принимает один параметр типа `Graphics`. Этот параметр содержит графический контекст, который описывает графическую среду, в которой аплет работает. Этот контекст используется всякий раз, когда запрашивается его вывод.

Внутри `paint()` вызывается `drawString()`, являющийся методом класса `Graphics`. Этот метод выводит строку в позиции, заданной координатами X,Y. Он имеет следующую общую форму:

```
void drawString(String message, int x, int y)
```

Здесь *`message`* — это строка, которая должна быть выведена в позиции, начиная с *`x`*, *`y`*. В окне Java верхний левый угол имеет координаты 0,0. Вызов `drawString()` в аплете отображает отображение строки “Простейший аплет”, начиная с позиции 20,20.

Обратите внимание, что аплет не имеет метода `main()`. В отличие от Java-программ, аплет не начинает выполнение с метода `main()`. Фактически большинство аплетов даже не имеют этого метода. Вместо этого аплет начинает выполнение, когда имя его класса передается средству просмотра аплетов или сетевому браузеру.

После ввода исходного текста `SimpleApplet` его компиляция выполняется так же, как компиляция обычных программ. Однако запуск `SimpleApplet` осуществляется иначе. Фактически есть два способа, которыми можно запустить аплет:

- Выполнение аплета внутри Java-совместимого браузера.
- Использование средства просмотра аплетов, такого как стандартный инструмент `appletviewer`. Он выполняет ваш аплет в окне. Обычно это самый быстрый и простой способ тестирования аплета.

Ниже подробно описывается каждый из этих способов.

Чтобы выполнить аплет в Web-браузере, вам необходимо написать короткий текстовый HTML-файл, который должен содержать соответствующий дескриптор. В настоящее время Sun рекомендует использовать дескриптор `APPLET`. Вот пример такого файла, выполняющего аплет `SimpleApplet`:

```
<applet code="SimpleApplet" width=200 height=60>
</applet>
```

Опции `width` и `height` указывают размеры области отображения, используемой аплетом. (Дескриптор `APPLET` содержит несколько других опций, которые рассматри-

ваются более подробно во второй части.) После того, как вы создадите этот файл, вы должны запустить браузер и затем загрузить в него этот файл, что вызовет выполнение SimpleApplet.

Чтобы выполнить SimpleApplet в средстве просмотра аплетов, вы также должны выполнить HTML-файл, показанный выше. Например, если предыдущий HTML-файл называется RunApp.html, то следующая командная строка запустит его на выполнение:

```
C:\>appletviewer RunApp.html
```

Однако существует более удобный метод, который ускорит тестирование. Просто включите комментарий в начало исходного файла Java, который указан в дескрипторе APPLET. В результате ваш код будет документирован прототипом необходимых HTML-конструкций, и вы сможете тестировать скомпилированный аплет, запуская средство просмотра аплетов с указанием исходного Java-файла. Если вы прмиеняете этот метод, то исходный файл 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("A Simple Applet", 20, 20);
  }
}
```

При таком подходе вы сможете быстро проходить итерации разработки аплета, выполняя перечисленные ниже три шага.

1. Редактирование файлы исходного текста Java.
2. Компиляция программы.
3. Выполнение средства просмотра аплетов с указанием имени вашего исходного файла. Средство просмотра аплетов обнаружит дескриптор APPLET внутри комментария и запустит ваш аплет.

Окно, порожденное SimpleApplet, как оно будет отображено средством просмотра аплетов, показано на рис. 13.1.

Рис. 13.1. Аплет SimpleApplet во время выполнения

Хотя сущность аплетов будет обсуждаться далее в этой книге, здесь мы укажем ключевые моменты, о которых нужно знать сейчас.

- Аплеты не нуждаются в методе `main()`.
- Аплеты должны запускаться под управлением средства просмотра аплетов или Java-совместимого Web-браузера.
- Пользовательский ввод-вывод в аплетах не выполняется посредством классов ввода-вывода. Вместо этого аплеты применяют интерфейс, предоставляемый средствами AWT или Swing.

Модификаторы `transient` и `volatile`

Java определяет два интересных модификатора типов: `transient` и `volatile`. Эти модификаторы служат для управления некоторыми специфическими ситуациями.

Когда экземпляр переменной объявлен как `transient`, его значение не должно удерживаться, когда объект сохраняется. Например:

```
class T {
   transient int a;  // не будет удерживаться
   int b;            // будет удерживаться
}
```

Здесь если объект типа `T` записывается в область постоянного хранения, то содержимое `a` не должно сохраняться, а содержимое `b` — должно.

Модификатор `volatile` сообщает компилятору, что переменная, модифицированная с помощью `volatile`, может быть неожиданно изменена другими частями вашей программы. Одна из таких ситуаций возникает в многопоточных программах. (Вы видели пример в главе 11.) В многопоточных программах иногда два или более потоков управления разделяют доступ к одной и той же переменной. Из соображений эффективности каждый поток может хранить свою собственную приватную копию этой переменной. Реальная (или *мастер*-) копия переменной обновляется в различные моменты, например при входе в метод `synchronized`. Хотя такой подход работает нормально, все же иногда он недостаточно эффективен. В некоторых случаях все, что действительно происходит — это то, что мастер-копия переменной всегда отражает ее текущее состояние. Чтобы обеспечить это, просто объявите переменную как `volatile`, что сообщит компилятору, что он должен всегда использовать мастер-копию переменной `volatile` (или же, как минимум, всегда держать приватные ее копии синхронизированными с мастер-копией и наоборот). Кроме того, доступ к мастер-копии переменной должен осуществляться в том же порядке, как он выполнялся к приватной копии.

Использование `instanceof`

Иногда может оказаться желательным знать тип объекта во время выполнения программы. Например, вы можете иметь один поток выполнения, который генерирует объекты различных типов, и другой поток, который их использует. В этой ситуации для обрабатывающего потока может быть удобно знать тип каждого объекта, который он получает. Другая ситуация, когда знание типа объекта во время выполнения важно — это когда используется приведение типа. В Java неправильное приведение типа вызывает ошибку времени выполнения. Множество неверных приведений типа могут быть пере-

хвачены на этапе компиляции. Однако приведение типов в пределах иерархии классов может стать причиной ошибок приведения, которые обнаруживаются только во время выполнения. Например, суперкласс по имени A может порождать два подкласса — B и C. Таким образом, приведение объекта типа B к типу A, или приведение C к A допустимо, но приведение объекта типа B к типу C (и наоборот) — некорректно. Поскольку объект типа A может ссылаться на объекты и типа B, и типа C, как вы можете узнать во время выполнения, к какому именно типу обращается ссылка перед тем, как осуществить приведение к типу C? Это может быть объект типа A, B или C. Если это объект типа B, то будет возбуждено исключение времени выполнения. Для получения ответа на этот вопрос Java предлагает операцию времени выполнения `instanceof`.

Общая форма операции `instanceof` такова:

```
object instanceof type
```

Здесь *object* — ссылка на экземпляр класса, а *type* — тип класса. Если *object* относится к указанному типу или может быть приведен к нему, то операция `instanceof` дает в результате `true`. В противном случае результатом будет `false`. То есть `instanceof` — это средство, с помощью которого программа может получить информацию об объекте во время выполнения.

В следующей программе демонстрируется применение `instanceof`.

```
// Демонстрация использования операции 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 есть экземпляр A");
  if(b instanceof B)
     System.out.println("b есть экземпляр B");
  if(c instanceof C)
     System.out.println("c есть экземпляр C");
  if(c instanceof A)
     System.out.println("c может быть приведен к A");
  if(a instanceof C)
     System.out.println("a может быть приведен к C");
  System.out.println();
  // сравнение типов с порожденными типами
  A ob;
  ob = d; // Ссылка на d
```

```
    System.out.println("ob теперь ссылается на d");
    if(ob instanceof D)
      System.out.println("ob есть экземпляр 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 может быть приведен к D");
    else
      System.out.println("ob не может быть приведен к D");
    if(ob instanceof A)
      System.out.println("ob может быть приведен к A");
    System.out.println();
    // все объекты могут быть приведены к Object
    if(a instanceof Object)
      System.out.println("a может быть приведен к Object");
    if(b instanceof Object)
      System.out.println("b может быть приведен к Object");
    if(c instanceof Object)
      System.out.println("c может быть приведен к Object");
    if(d instanceof Object)
      System.out.println("d может быть приведен к Object");
  }
  }
```

Результат работы этой программы:

```
a есть экземпляр A
b есть экземпляр B
c есть экземпляр C
c может быть приведен к A
ob теперь ссылается на d
ob есть экземпляр D
ob теперь ссылается на c
ob не может быть приведен к D
ob может быть приведено к A
a может быть приведен к Object
b может быть приведен к Object
c может быть приведен к Object
d может быть приведен к Object
```

Большинство программ не нуждаются в операции `instanceof`, поскольку обычно вам известны типы объектов, с которыми работаете. Однако она может оказаться очень полезной, когда вы разрабатываете обобщенные процедуры, имеющие дело с объектами из сложной иерархии классов.

strictfp

`strictfp` является относительно новым ключевым словом. Когда была выпущена Java 2, модель вычислений с плавающей точкой была слегка упрощена. В частности, новая модель не требовала округления некоторых промежуточных результатов вычислений. В ряде случаев это предотвращает переполнение. Модифицируя класс или метод словом `strictfp`, вы гарантируете, что вычисления с плавающей точкой будут выполняться точ-

но так, как они выполнялись в ранних версиях Java. Когда класс модифицирован словом `strictfp`, все его методы автоматически модифицируются `strictfp`.

Например, следующий фрагмент сообщает Java, что нужно использовать исходную модель вычислений с плавающей точкой при вычислении всех методов, определенных в классе `MyClass`:

```
strictfp class MyClass { //...
```

Откровенно говоря, большинству программистов никогда не понадобится использовать `strictfp`, поскольку оно касается лишь небольшого класса проблем.

Родные методы

Хотя это случается редко, но все же иногда может понадобиться вызвать подпрограмму, написанную на языке, отличном от Java. Обычно такая подпрограмма существует в виде исполняемого кода для центрального процессора и среды, в которой вы работаете, то есть в виде родного (native) кода. Например, вы можете решить вызвать такую подпрограмму для достижения более высокой скорости выполнения. Или же вам может понадобиться работать со специализированной библиотекой от независимых поставщиков, например, с пакетом статистических расчетов. Однако поскольку Java-программы компилируются в байт-код, который затем интерпретируется (или компилируется "на лету") исполняющей системой Java, вызов подпрограмм родного кода из программ на Java может показаться невозможным. К счастью, это заключение ложно. В Java предусмотрено ключевое слово `native`, которое используется для объявления родных методов. Однажды объявленные, эти методы могут быть вызваны из вашей Java-программы точно так же, как вызываются любой другой метод Java.

Чтобы объявить родной метод, предварите его имя модификатором `native`, но не определяйте тело метода. Например:

```
public native int meth() ;
```

После объявления метода нужно собственно написать его и выполнить серию относительно сложных шагов, чтобы соединить его с кодом Java.

Большинство родных методов пишется на языке C. Механизм интеграции кода C с программой Java называется интерфейсом *JNI* (Java Native Interface). Подробное описание JNI выходит за рамки настоящей книги, но предложенное ниже краткое описание дает достаточную информацию для большинства приложений.

На заметку! *Точные шаги, которым вы должны следовать, варьируются между различными средами Java. Они также зависят от языка, который используется для реализации родных методов. Следующий пример ориентирован на среду Windows. Язык реализации метода — C.*

Простейший способ понять процесс — исследовать его на примере. Для начала введите следующую короткую программу, которая использует `native`-метод по имени `test()`:

```
// Простой пример использования native-метода.
public class NativeDemo {
   int i;
   public static void main(String args[]) {
      NativeDemo ob = new NativeDemo();
      ob.i = 10;
      System.out.println("Это ob.i перед вызовом native-метода:" + ob.i);
```

```
        ob.test(); // вызов native метода
        System.out.println("Это ob.i после вызова native-метода:" + ob.i);
    }

    // Объявление native-метода
    public native void test() ;

    // загрузить DLL-библиотеку, содержащую статический метод
    static {
        System.loadLibrary("NativeDemo");
    }
}
```

Обратите внимание, что метод `test()` объявлен как `native` и не имеет тела. Это метод, который будет вскоре реализован на C. Также посмотрите на блок `static`. Как уже объяснялось ранее в нашей книге, блок `static` выполняется только однажды — при запуске программы (или, точнее говоря, при первой загрузке ее класса). В этом случае он используется для загрузки динамической библиотеки, которая содержит реализацию метода `test()`. (Вскоре вы увидите, как создавать такую библиотеку.)

Библиотека загружается методом `loadLibrary()`, который является частью класса `System`. Его общая форма такова:

```
static void loadLibrary(String filename)
```

Здесь *`filename`* — строка, которая специфицирует имя файла, содержащего библиотеку. Для среды Windows предполагается, что файл имеет расширение `.DLL`.

После ввода текста программы скомпилируйте ее, чтобы получить `NativeDemo.class`. Далее вы должны использовать `javah.exe`, чтобы сгенерировать один заголовочный файл: `NativeDemo.h` (`javah.exe` включена в JDK). Вы включите `NativeDemo.h` в свою реализацию `test()`. Чтобы получить `NativeDemo.h`, выполните следующую команду:

```
javah -jni NativeDemo
```

Эта команда сгенерирует файл по имени `NativeDemo.h`. Этот файл должен быть включен в C-файл, реализующий `test()`. Вывод, сгенерированный этой командой, показан ниже.

```
/* 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, которая определяет прототип для создаваемой вами функции `test()`:

```
JNIEXPORT void JNICALL Java_NativeDemo_test(JNIEnv *, jobject);
```

Отметим, что именем функции будет `Java_NativeDemo_test()`. Вы должны использовать его в качестве имени `native`-функции, которую вы реализуете. То есть вместо написания на языке C функции `test()` вы создаете `Java_NativeDemo_test()`. Компонент `NativeDemo` в префиксе добавляется, поскольку он идентифицирует, что метод `test()` является членом класса `NativeDemo`. Помните, что другой класс может объявить свой собственный метод `test()`, абсолютно отличный от того, что объявлен в `NativeDemo`. Включение имени класса в префикс обеспечивает возможность различения разных версий. Основное правило — родным функциям присваивается имя, чей префикс включает имя класса, в котором он объявлен.

После генерации необходимого заголовочного файла вы можете написать свою реализацию метода `test()` и сохранить ее в файле `NativeDemo.c`:

```
/* Этот файл содержит C-версию метода test().*/
#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 native-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 native-метода.\n");
}
```

Отметим, что этот файл включает `jni.h`, содержащий интерфейсную информацию. Этот файл поставляется вместе с компилятором Java. Напомним, что заголовок `NativeDemo.h` ранее сгенерирован `javah`.

В этой функции метод `GetObjectClass()` используется для получения C-структуры, имеющей информацию о классе `NativeDemo`. Метод `GetFieldID()` возвращает C-структуру с информацией о поле класса по имени `i`. `GetIntField()` извлекает исходное значение этого поля. `SetIntField()` сохраняет обновленное значение этого поля (см. в файле `jni.h` дополнительные методы, которые управляют другими типами данных).

После создания `NativeDemo.c` вы должны скомпилировать его и создать DLL-библиотеку. Чтобы сделать это с помощью компилятора Microsoft C/C++, используйте следующую командную строку (возможно, понадобится указать путь к `jni.h` и его подчиненному файлу `jni_md.h`):

```
Cl /LD NativeDemo.c
```

Эта команда создаст файл NativeDemo.dll. Посте того как только все будет сделано, вы сможете запустить Java-программу, которая выдаст такой результат:

```
Это ob.i перед вызовом native-метода: 10
Запуск native-метода.
i = 10
Завершение native-метода.
Это ob.i после вызова native-метода: 20
```

Проблемы, связанные с родными методами

Родные методы выглядят многообещающе, поскольку они дают возможность получить доступ к существующей базе библиотечных подпрограмм, а также позволяют надеяться на обеспечения высокой скорости работы программ. Однако с этими методами связаны две существенных проблемы.

- **Потенциальный риск нарушения безопасности**. Поскольку родной метод выполняет реальный машинный код, он может получить доступ к любой части системы. То есть native-код не относится к исполняющей среде Java. Это, например, угрожает вирусной инфекцией. По этой причине аплеты не могут использовать native-методы. Кроме того, загрузка DLL-библиотеки может быть ограничена и она может быть субъектом утверждения для менеджера по безопасности.
- **Потеря переносимости**. Поскольку native-код содержится в DLL-библиотеке, он должен быть представлен на машине, которая выполняет Java-программу. Более того, поскольку каждый native-метод зависит от процессора и операционной системы, каждая DLL-библиотека, как следствие, не является переносимой. То есть Java-приложение, которое использует native-методы, сможет выполняться только на машине, на которой установлена совместимая DLL-библиотека.

Применение native-методов должно быть ограничено, поскольку они делают вашу Java-программу непереносимой и представляют существенный риск нарушения безопасности.

Использование assert

Другим относительно новым дополнением к Java является ключевое слово assert. Оно используется во время разработки программ для создания так называемых *утверждений* (assertion), представляющих собой условия, которые должны быть истинными во время выполнения программы. Например, вы можете иметь метод, который всегда возвращает положительное целое значение. Вы можете тестировать его утверждением, что возвращаемое значение больше нуля, используя оператор assert. Во время выполнения, если условие действительно истинно, то никаких других действий не выполняется. Однако если условие окажется ложным, будет возбуждено исключение AssertionError. Утверждения часто применяются при тестировании для верификации того, что некоторое ожидаемое условие действительно выполняется. В коде окончательной версии они, как правило, не присутствуют.

Ключевое слово assert имеет две формы. Первая выглядит так:

```
assert condition;
```

Здесь *condition* — выражение, которое должно при вычислении дать булевский результат. Если результат равен `true`, то утверждение истинно, и никаких действий не выполняется. Если же условие дает `false`, значит, произошел сбой и возбуждается объект исключения по умолчанию `AssertionError`.

Вторая форма `assert` выглядит следующим образом:

```
assert condition:expr;
```

В этой версии *expr* — значение, которое передается конструктору `AssertionError`. Это значение преобразуется в строковую форму и отображается, если утверждение ложно. Обычно вы специфицируете строку для *expr*, но разрешено любое выражение, отличное от `void`, до тех пор, пока оно допускает осмысленное строковое преобразование.

Ниже показан пример использования `assert`. В нем осуществляется проверка того, что возвращаемое значение `getnum()` положительно.

```
// Демонстрация assert.
class AssertDemo {
static int val = 3;
// Возвращает целое.
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; // произойдет сбой, если n == 0
     System.out.println("n равно " + n);
  }
}
}
```

Чтобы включить проверку утверждений во время выполнения, вы должны указать опцию `-ea`. Например, чтобы сделать это для `AssertDemo`, выполните следующую команду:

```
java -ea AssertDemo
```

После компиляции и запуска, как показано выше, программа выдает 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
  в AssertDemo.main(AssertDemo.java:17)
```

В `main()` выполняются повторяющиеся вызовы метода `getnum()`, который возвращает целое значение. Возвращаемое значение `getnum()` присваивается `n` и затем тестируется оператором `assert`:

```
assert n > 0; // произойдет сбой, если n == 0
```

Этот оператор завершится сбоем, когда n будет равно нулю, что произойдет после четвертого вызова. Когда подобное случится, будет возбуждено исключение.

Как объяснялось, вы можете специфицировать сообщение, отображаемое при сбое утверждения. Например, если вы подставите

```
assert n > 0 : "n отрицательное!";
```

в утверждение из предыдущей программы, 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)
```

Один момент, важный для понимания утверждений — это то, что вы не должны полагаться на них для выполнения каких-либо действий программы. Причина в том, что нормальный код окончательной версии будет выполняться с отключенным механизмом проверки утверждений. Например, рассмотрим следующий вариант предыдущей программы:

```
// Плохой способ применения assert!!!
class AssertDemo {
// получить генератор случайных чисел
static int val = 3;
// Возвращает целое.
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; // Плохая идея!
     System.out.println("n is " + n);
  }
}
}
```

В этой версии программы вызов `getnum()` перемещен в оператор `assert`. Хотя это хорошо работает, когда механизм проверки утверждений включен, его отключение приведет к неправильной работе программы, потому что вызов `getnum()` никогда не произойдет! Фактически n теперь должно быть инициализировано, поскольку компилятор распознает ситуацию, что значение может не быть присвоено в операторе `assert`.

Утверждения — хорошее нововведение в Java, потому что оно упрощает тип проверки ошибок, который часто используется во время разработки. Так, например, до появления `assert`, если вы хотели проверить, что n имеет положительное значение в приведенной выше программе, то должны были написать примерно следующую последовательность кода:

```
if(n < 0) {
   System.out.println("n отрицательное!");
   return; // или возбудить исключение
}
```

Для применения `assert` нужна только одна строка кода. Более того, вам не придется удалять строки с `assert` из окончательного варианта кода.

Опции включения и отключения утверждений

При выполнении кода вы можете отключить механизм утверждений опцией `-da`. Вы можете включить или отключить его для специфического пакета, указав его имя и опцию `-ea` или `-da`. Например, чтобы включить механизм проверки утверждений для пакета `MyPack`, используйте

```
-ea:MyPack
```

а чтобы отключить:

```
-da:MyPack
```

Чтобы включить или отключить механизм для всех вложенных пакетов, после имени пакета нужно поставить многоточие:

```
-ea:MyPack...
```

Вы можете также специфицировать класс с опцией `-ea` или `-da`. Например, это включает индивидуально `AssertDemo`:

```
-ea:AssertDemo...
```

Статический импорт

В J2SE 5 появилось новое средство Java — *статический импорт*, что расширяет возможности ключевого слова `import`. `import` с предшествующим ключевым словом `static` может применяться для импорта статических членов класса или интерфейса. При использовании статического импорта появляется возможность ссылаться на статические члены непосредственно по именам, без необходимости квалифицировать их именем класса. Это упрощает и сокращает синтаксис, необходимый для работы со статическими членами.

Чтобы понять удобство статического импорта, давайте начнем с примера, который *не* использует его. Следующая программа вычисляет гипотенузу прямоугольного треугольника. Она использует два статических метода из встроенного Java-класса `Math`, входящего в пакет `java.lang`. Первый из них — `Math.pow()` — возвращает значение, возведенное в указанную степень. Второй — `Math.sqrt()` — возвращает квадратный корень аргумента.

```
// Вычисляет 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, что sqrt() и pow() должны быть
  // квалифицированы именем их класса - Math.
  hypot = Math.sqrt(Math.pow(side1, 2) + Math.pow(side2, 2));

  System.out.println("Даны длины сторон " +
     side1 + " и " + side2 + " гипотенуза равна " + hypot);
}
}
```

Поскольку pow() и sqrt() — статические методы, они должны быть вызваны с указанием имени их класса — Math. Это приводит к следующему громоздкому вычислению гипотенузы:

```
hypot = Math.sqrt(Math.pow(side1, 2) + Math.pow(side2, 2));
```

Как иллюстрирует этот простой пример, необходимость каждый раз специфицировать имя класса при вызовах pow() и sqrt() (или любых других математических методов Java вроде sin(), cos() и tan()) довольно утомительно.

Вы можете избежать утомительного повторения имени пакета благодаря применению статического импорта, как показано в следующей версии предыдущей программы:

```
// Применение статического импорта, делающего sqrt() и pow() видимыми.
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;
// Вычисление гипотенузы прямоугольного треугольника.
class Hypot {
public static void main(String args[]) {
  double side1, side2;
  double hypot;
  side1 = 3.0;
  side2 = 4.0;

  // Здесь sqrt() ирow() можно вызывать
  // непосредственно, без их имени класса.
  hypot = sqrt(pow(side1, 2) + pow(side2, 2));

  System.out.println("Даны длины сторон " +
     side1 + " и " + side2 + " гипотенуза равна " + hypot);
}
}
```

В этой версии имена sqrt и pow становятся видимыми благодаря оператору статического импорта:

```
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;
```

После этих операторов больше нет необходимости квалифицировать pow() и sqrt() именем их класса. Таким образом, вычисление гипотенузы может быть выражено более удобно:

```
hypot = sqrt(pow(side1, 2) + pow(side2, 2));
```

Как видите, эта форма и более читабельна.

Существуют две основных формы оператора import static. Первая, которая использовалась в предыдущем примере, делает видимым единственное имя. Его общая форма такова:

```
import static pkg.type-name.static-member-name;
```

Здесь *type-name* — имя класса или интерфейса, который содержит требуемый статический член. Полное имя его пакета указано в *pkg*, а имя члена — в *static-member-name*.

Вторая форма статического импорта позволяет импортировать все статические члены данного класса или интерфейса. Его общая форма показана ниже.

```
import static pkg.type-name.*;
```

Если вы будете использовать много статических методов или полей, определенных в классе, то эта форма позволит вам сделать их видимыми без необходимости специфицировать каждый отдельно. Таким образом, в предыдущей программе с помощью единственного оператора `import` можно ввести в область видимости `pow()` и `sqrt()` (а также *все другие* статические члены `Math`):

```
import static java.lang.Math.*;
```

Конечно же, статический импорт не ограничивается только классом `Math` или его методами. Например, следующая строка вводит в область видимости статическое поле `System.out`:

```
import static java.lang.System.out;
```

После этого оператора вы можете выводить информацию на консоль потоком `out`, не квалифицируя его `System`, как показано здесь:

```
out.println("Импортировав System.out, вы можете использовать его непосредственно.");
```

Однако импортировать `System.out`, как показано выше — это не только хорошая идея, но и предмет обсуждения. Несмотря на то что это сокращает текст программы, все же теперь не будет очевидно тем, кто читает программу, что `out` ссылается на `System.out`.

Еще один момент: в дополнение к импорту статических членов классов и интерфейсов, определенных в Java API, вы можете также использовать статический импорт для импортирования статических членов ваших собственных классов и интерфейсов.

Каким бы удобным ни показался статический импорт, важно не злоупотреблять им. Помните, что причина того, что библиотечные классы Java объединены в пакеты, состоит в том, чтобы избежать конфликтов пространств имен и для непреднамеренного сокрытия прочих имен. Если вы используете в своей программе статический член однажды или дважды, лучше его не импортировать. К тому же некоторые статические имена, как, например `System.out`, настолько привычны и узнаваемы, что, вероятно, вы вообще не захотите импортировать их. Статический импорт предназначен для тех ситуаций, когда вы применяете статические члены многократно, как, например, при выполнении серии математических вычислений. То есть, в сущности, вам стоит использовать это средство, но не злоупотреблять им.

Вызов перегруженных конструкторов через `this()`

Имея дело с перегруженными конструкторами, иногда удобно один конструктор вызывать из другого. В Java это обеспечивается использованием другой формы ключевого слова `this`. Вот его общая форма:

```
this(arg-list)
```

Когда выполняется `this()`, сначала выполняется перегруженный конструктор, который соответствует списку параметров *`arg-list`*. Затем выполняются операторы, находящиеся внутри исходного конструктора, если таковые присутствуют. Вызов `this()` должен быть первым оператором в конструкторе.

Чтобы понять, как следует использовать `this()`, рассмотрим короткий пример. Для начала приведем класс, который *не* использует `this()`:

```
class MyClass {
   int a;
   int b;
   // Инициализировать a и b индивидуально
   MyClass(int i, int j) {
      a = i;
      b = j;
   }
   // Инициализировать a и b одним и тем же значением
   MyClass(int i) {
      a = i;
      b = i;
   }
   // Присвоить a и b значение по умолчанию 0
   MyClass( ) {
      a = 0;
      b = 0;
   }
}
```

Этот класс включает в себя три конструктора, каждый из которых инициализирует значения `a` и `b`. Первому передаются индивидуальные значения для `a` и `b`. Второй принимает только одно значение и присваивает его и `a`, и `b`. Третий присваивает `a` и `b` значение по умолчанию — 0.

Используя `this()`, можно переписать приведенный `MyClass` следующим образом:

```
class MyClass {
   int a;
   int b;
   // Инициализировать a и b индивидуально
   MyClass(int i, int j) {
      a = i;
      b = j;
   }
   // Инициализировать a и b одним и тем же значением
   MyClass(int i) {
      this(i, i); // вызывается MyClass(i, i);
   }
   // Присвоить a и b значение по умолчанию 0
   MyClass( ) {
      this(0); // вызывается MyClass(0)
   }
}
```

В этой версии `MyClass` единственным конструктором, который в действительности присваивает значения полям `a` и `b`, является `MyClass(int, int)`. Например, посмотрим, что случится при выполнении следующего оператора:

```
MyClass mc = new MyClass(8);
```

Вызов `MyClass(8)` заставляет выполниться `this(8,8)`, что транслируется в вызов `MyClass(8,8)`, поскольку именно эта версия конструктора `MyClass` соответствует данному вызову `this()` по списку параметров. Теперь рассмотрим следующий оператор, использующий конструктор по умолчанию:

```
MyClass mc2 = new MyClass();
```

В этом случае вызывается `this(0)`. Это заставляет выполниться `MyClass(0)`, поскольку именно эта версия конструктора подходит по списку параметров. Конечно же, `MyClass(0)` затем обращается к `MyClass(0,0)`, как только что было описано.

Одной из причин, по которым стоит вызывать перегруженные конструкторы через `this()`, является исключение дублирования кода. Во многих случаях сокращение дублированного кода сокращает время загрузки ваших классов, поскольку код объекта уменьшается. Это особенно важно для программ, доставляемых по Internet, когда время загрузки критично. Применение `this()` может также помочь структурировать ваш код, когда конструкторы содержат большой объем дублированного кода.

14 ГЛАВА

Обобщения

После выхода в 1995 г. первоначальной версии 1.0 в язык Java было добавлено множество новых средств. Одним из наиболее значительных и влиятельных новшеств стали *обобщения* (generics). Во-первых, их появление означало добавление новых синтаксических элементов в язык. Во-вторых, они повлекли за собой изменения во многих классах и методах самого API ядра. Поскольку обобщения оказали столь значительное влияние на язык, многие программисты с большим трудом привыкают к их использованию. Однако с появлением версии JDK 6 игнорировать обобщения становится практически невозможно. Примите как неизбежность то, что собираясь программировать на Java SE, вам обязательно придется иметь дело с обобщениями. К счастью, это средство не так трудно использовать и оно сулит значительные выгоды для программистов Java.

За счет применения обобщений стало возможным создавать классы, интерфейсы и методы, работающие в безопасной к типам манере 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 типов `Integer`, `String`, `Object` или `Thread`. Благодаря обобщениям, вы можете определить алгоритм однажды, независимо от конкретного типа данных, и затем применять его к широкому разнообразию типов данных без каких-либо дополнительных усилий. Впечатляющая мощь добавленных к языку обобщений, фундаментально изменила способы написания кода Java.

Вероятно, одно из средств Java, которое в наибольшей степени испытало влияние обобщений — это *каркас коллекций* (Collections Framework). Упомянутый каркас является частью Java API и подробно описана в главе 17, но стоит вкратце пояснить ее здесь. *Коллекция* — это группа объектов. Каркас коллекций определяет несколько классов, таких как списки и карты, которые управляют коллекциями. Классы коллекций всегда готовы работать с объектами любых типов. Выгода от добавления в язык обобщений состоит в том, что классы коллекций теперь могут использоваться с полным обеспечением безопасности типов. То есть, помимо предоставления мощного нового элемента языка, обобщения также значительно усовершенствовали существующие средства. Вот почему обобщения представляют собой столь значимое дополнение к Java.

В настоящей главе описан синтаксис, теория и применение обобщений. Мы покажем, как обобщения обеспечивают безопасность типов в некоторых ранее трудных случаях.

Когда вы изучите эту главу, то захотите ознакомиться с главой 17, описывающей систему коллекций. Там вы найдете множество примеров работы обобщений.

Помните! *Обобщения были добавлены в J2SE 5.0. Исходные тексты, использующие их, не могут быть скомпилированы старыми версиями* `javac`.

Что такое обобщения?

По сути дела, *обобщения* — это *параметризованные типы*. Параметризованные типы важны, поскольку позволяют объявлять классы, интерфейсы и методы, где тип данных, которыми они оперируют, указан в виде параметра. Используя обобщения, можно создать единственный класс, например, который будет автоматически работать с разными типами данных. Классы, интерфейсы или методы, имеющие дело с параметризованными типами, называются *обобщениями*, *обобщенными классами* или *обобщенными методами*.

Важно понимать, что Java всегда предлагала возможность создавать в определенной мере обобщенные классы, интерфейсы и методы, оперирующие ссылками на тип `Object`. Поскольку `Object` — это суперкласс для всех остальных классов, ссылка на `Object` может обращаться к объекту любого типа. То есть в старом коде обобщенные классы, интерфейсы и методы использовали ссылки на `Object` для того, чтобы оперировать объектами различного типа. Проблема была в том, что они не могли обеспечить безопасность типов.

Обобщения добавили в язык безопасность типов, которой так не хватало. Они также упростили процесс, поскольку теперь нет необходимости применять явные приведения для транслирования объектов `Object` в реальные типы данных, с которыми выполняются действия. Благодаря обобщениям, все приведения выполняются автоматически и неявно. То есть обобщения расширили ваши возможности повторного использования кода и позволили вам делать это легко и безопасно.

На заметку! *Предупреждение для программистов C++: хотя обобщения похожи на шаблоны в C++, это не одно и то же. Существует ряд фундаментальных отличий между двумя подходами к обобщенным типам. Если у вас имеется опыт применения C++, важно не делать поспешных выводов о том, как обобщения работают в Java.*

Простой пример обобщения

Давайте начнем с простого примера обобщенного класса. В следующей программе определены два класса. Первый — это обобщенный класс `Gen`, а второй — `GenDemo`, использующий `Gen`.

```
// Простой обобщенный класс.
// Здесь T — это параметр типа,
// который будет заменен реальным типом
// при создании объекта типа Gen.
class Gen<T> {
   T ob; // объявление объекта типа T
   // Передать конструктору ссылку
   // на объект типа T.
   Gen(T o) {
      ob = o;
   }
```

```
   // Вернуть ob.
   T getob() {
      return ob;
   }
   // Показать тип T.
   void showType() {
      System.out.println("Типом T является " + ob.getClass().getName());
   }
}

// Демонстрация обобщенного класса.
class GenDemo {
   public static void main(String args[]) {
      // Создать Gen-ссылку для Integers.
      Gen<Integer> iOb;
      // Создать объект Gen<Integer> и присвоить
      // ссылку на iOb. Отметьте применение автоупаковки
      // для инкапсуляции значения 88 в объект Integer.
      iOb = new Gen<Integer>(88);
      // Показать тип данных, используемый iOb.
      iOb.showType();
      // Получить значение iOb. Обратите внимание,
      // что никакого приведения не нужно.
      int v = iOb.getob();
      System.out.println("значение: " + v);
      System.out.println();
      // Создать объект Gen для String.
      Gen<String> strOb = new Gen<String>("Обобщенный тест");
      // Показать тип данных, используемый strOb.
      strOb.showType();
      // Получить значение strOb. Опять же
      // приведение не требуется.
      String str = strOb.getob();
      System.out.println("Значение: " + str);
   }
}
```

Результат работы этой программы:

```
Типом T является java.lang.Integer
Значение: 88

Типом T является java.lang.String
Значение: Обобщенный тест
```

Давайте внимательно исследуем эту программу.
Во-первых, обратите внимание на объявление `Gen` в следующей строке:

```
class Gen<T> {
```

Здесь `T` — имя *типа-параметра*. Это имя используется в качестве заполнителя, куда будет подставлено имя реального типа, переданного `Gen` при создании реальных типов. То есть `T` применяется в `Gen` всякий раз, когда требуется *тип-параметр*. Обратите внимание, что `T` заключен в <>. Этот синтаксис может быть обобщен. Всякий раз, когда объявляется тип-параметр, он указывается в угловых скобках. Поскольку `Gen` применяет тип-параметр, `Gen` является обобщенным классом, который называется также *параметризованным типом*.

Далее T используется для объявления объекта по имени ob, как показано ниже:

```
T ob; // объявляет объект типа T
```

Как объяснялось, T — это место подстановки реального типа, который будет указан при создании объекта Gen. То есть ob будет объектом типа, переданного в T. Например, если T передан тип String, то экземпляр ob будет типа String.

Теперь рассмотрим конструктор Gen:

```
Gen(T o) {
   ob = o;
}
```

Как видите, параметр o имеет тип T. Это значит, что реальный тип o определяется типом, переданным T при создании объекта Gen. К тому же, поскольку и переменная-параметр o, и переменная-член ob имеют тип T, они обе получают одинаковый реальный тип при создании Gen.

Параметр типа T также может быть использован для спецификации типа возврата для метода, как в случае метода getob(), показанного здесь:

```
T getob() {
   return ob;
}
```

Так как ob тоже имеет тип T, его тип совместим с типом, возвращаемым getob().

Метод showType() отображает тип T вызовом getName() на объекте Class, возвращенным вызовом getClass() на ob. Метод getClass() определен в Object, и потому является членом всех классов. Он возвращает объект Class, соответствующий типу класса объекта, на котором он вызван. Class определяет метод getName(), который возвращает строковое представление имени класса.

Класс GenDemo демонстрирует обобщенный класс Gen. Сначала он создает версию Gen для целых, как показано ниже:

```
Gen<Integer> iOb;
```

Посмотрим на это объявление внимательней. Во-первых, отметим, что тип Integer специфицирован в угловых скобках после Gen. В этом случае Integer — это тип-аргумент, который передается в параметре типа Gen, T. Это эффективно создает версию Gen, в которой все ссылки на T транслируются в ссылки на Integer. То есть в данном объявлении ob имеет тип Integer и тип возврата getob() также имеет тип Integer.

Прежде чем двигаться дальше, необходимо заявить, что компилятор Java на самом деле не создает различные версии Gen или любого другого обобщенного класса. Хотя было бы удобно думать в таких терминах, на самом деле подобное не происходит. Вместо этого компилятор удаляет всю обобщенную информацию о типах, выполняя необходимые приведения, чтобы сделать поведение вашего кода таким, будто создана специфическая версия Gen. То есть имеется только одна версия Gen, которая существует в вашей программе. Процесс удаления обобщенной информации о типе называется *очисткой* (erasure), и мы вернемся к этой теме чуть позднее в настоящей главе.

Следующая строка присваивает iOb ссылку на экземпляр Integer-версию класса Gen:

```
iOb = new Gen<Integer>(88);
```

Отметим, что когда вызывается конструктор Gen, аргумент типа Integer также указывается. Это необходимо, потому что типом объекта (в данном случае iOb), которому

присваивается ссылка, является Gen<Integer>. То есть ссылка, возвращаемая new, также должна иметь тип Gen<Integer>. Если это не так, получается ошибка времени компиляции. Например, следующее присваивание вызовет ошибку компиляции:

```
iOb = new Gen<Double>(88.0); // Ошибка!
```

Поскольку iOb имеет тип Gen<Integer>, он не может быть использован для присваивания ссылки типа Gen<Double>. Эта проверка типа является одним из основных преимуществ обобщений, потому что обеспечивает безопасность типов.

Как указано в комментарии к программе, присваивание

```
iOb = new Gen<Integer>(88);
```

использует автоупаковку для инкапсуляции значения 88, имеющего тип int, в Integer. Это работает, потому что Gen<Integer> создает конструктор, принимающий аргумент Integer. Поскольку ожидается Integer, Java автоматически упаковывает 88 внутрь него. Конечно, присваивание также может быть написано явно, как здесь:

```
iOb = new Gen<Integer>(new Integer(88));
```

Однако с этой версией не связано никаких преимуществ.

Программа затем отображает тип ob внутри iOb, который есть Integer. Далее программа получает значение ob в следующей строке:

```
int v = iOb.getob();
```

Поскольку возвращаемым типом getob() будет T, который заменяется на Integer при объявлении iOb, то возвращаемым типом getob() также будет Integer, который автоматически распаковывается в int и присваивается переменной v, имеющей тип int. То есть нет никакой необходимости приводить тип возвращаемого значения getob() к Integer. Конечно, использовать автоупаковку не обязательно. Предыдущая строка может быть написана так:

```
int v = iOb.getob().intValue();
```

Однако автоупаковка позволяет сделать код более компактным.

Далее в GenDemo объявляется объект типа Gen<String>:

```
Gen<String> strOb = new Gen<String>(" Обобщенный тест");
```

Поскольку типом-аргументом является String, String подставляется вместо T внутри Gen. Это создает (концептуально) String-версию Gen, что и демонстрируют остальные строки программы.

Обобщения работают только с объектами

Когда объявляется экземпляр обобщенного типа, аргумент, переданный в качестве параметра типа, должен быть типом класса. Вы не можете использовать примитивный тип вроде int или char. Например, Gen можно передать любой тип класса в T, но нельзя передать примитивный тип в качестве параметра типа. Таким образом, следующее объявление недопустимо:

```
Gen<int> strOb = new Gen<int>(53); // Ошибка, нельзя использовать
                                   // примитивные типы
```

Конечно, невозможность использовать примитивный тип не является серьезным ограничением, так как вы можете применять оболочки типов (как это и делается в предыду-

щем примере) для инкапсуляции примитивных типов. Более того, механизм автоупаковки и автораспаковки Java делает использование оболочек типов прозрачным.

Обобщенные типы отличаются в зависимости от типов-аргументов

Ключевой момент в понимании обобщенных типов в том, что ссылка на одну специфическую версию обобщенного типа не совместима с другой версией того же обобщенного типа. Например, следующая строка, если ее добавить к предыдущей программе, вызовет ошибку и программа не скомпилируется:

```
iOb = strOb; // Не верно!
```

Даже несмотря на то, что `iOb` и `strOb` имеют тип `Gen<T>`, они являются ссылками на разные типы, потому что типы их параметров отличаются. Это часть того способа, благодаря которому обобщения добавляют безопасность типов и предотвращают ошибки.

Обобщения повышают безопасность типов

Теперь вы можете задать себе следующий вопрос: если та же функциональность, которую мы обнаружили в обобщенном классе `Gen`, может быть достигнута без обобщений, т.е. простым указанием `Object` в качестве типа данных и применением правильных приведений, в чем же выгода от того, что класс `Gen` параметризован? Ответ: в том, что обобщения автоматически гарантируют безопасность типов во всех операциях, где задействован `Gen`. В процессе работы с ним исключается необходимость явного приведения и ручной проверки типов в коде.

Чтобы понять выгоды от обобщений, для начала рассмотрим следующую программу, которая создает необобщенный эквивалент `Gen`:

```
// NonGen — функциональный эквивалент Gen
// не использующий обобщений.
class NonGen {
Object ob; // 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 ob.
void showType() {
  System.out.println("Типом ob является " +
  ob.getClass().getName());
}
}
// Демонстрация необобщенного класса.
class NonGenDemo {
public static void main(String args[]) {
  NonGen iOb;
  // Создать объект NonGen и сохранить
  // Integer в нем. Автоупаковка используется.
  iOb = new NonGen(88);
```

```
    // Показать тип данных, используемый iOb.
    iOb.showType();
    // Получить значение iOb.
    // На этот раз приведение необходимо.
    int v = (Integer) iOb.getob();
    System.out.println("значение: " + v);
    System.out.println();
    // Создать другой объект NonGen и
    // сохранить в нем String.
    NonGen strOb = new NonGen("Тест без обобщений");
    // Показать тип данных, используемый strOb.
    strOb.showType();
    // Получить значение strOb.
    // Опять же — приведение необходимо.
    String str = (String) strOb.getob();
    System.out.println("Значение: " + str);
    // Это компилируется, но концептуально неверно!
    iOb = strOb;
    v = (Integer) iOb.getob(); // ошибка времени выполнения!
  }
  }
```

В этой версии программы присутствует несколько интересных моментов. Для начала `NonGen` заменяет все обращения к типу `T` на `Object`. Это дает `NonGen` возможность хранить объекты любого типа, как это делает и обобщенная версия. Однако это не дает возможности Java-компилятору иметь какую-то реальную информацию о типе данных, в действительности сохраняемых в `NonGen`, что плохо по двум причинам. Во-первых, для извлечения сохраненных данных требуется явное приведение. Во-вторых, многие ошибки несоответствия типов не могут быть обнаружены до времени выполнения. Рассмотрим каждую из этих проблем поближе.

Обратите внимание на строку:

```
int v = (Integer) iOb.getob();
```

Поскольку типом возврата `getob()` является `Object`, необходимо привести его к `Integer`, чтобы позволить выполнить автораспаковку и сохранить значение в `v`. Если убрать приведение, программа не скомпилируется. В версии с обобщением приведение происходит неявно. В версии без обобщения приведение должно быть явным. Это не только неудобство, но также потенциальный источник ошибок.

Теперь рассмотрим следующую кодовую последовательность в конце программы:

```
// Это компилируется, но концептуально неверно!
iOb = strOb;
v = (Integer) iOb.getob(); // ошибка времени выполнения!
```

Здесь `strOb` присваивается `iOb`. Однако `strOb` ссылается на объект, содержащий строку, а не целое число. Это присваивание синтаксически корректно, потому что все ссылки `NonGen` одинаковы, и любая ссылка `NonGen` может указывать на любой другой объект типа `NonGen`. Однако этот оператор семантически неверен, что и отражено в следующей строке. Здесь тип возврата `getob()` приводится к `Integer`, и затем делается попытка присвоить это значение `v`. Проблема в том, что `iOb` теперь ссылается на объект, который хранит `String`, а не `Integer`. К несчастью, без использования обобщений компилятор Java не имеет возможности обнаружить это. Вместо этого возбуждается исключение времени выполнения. Возможность создавать безопасный в отношении типов код, в котором

ошибки несоответствия типов перехватываются компилятором — это главная выгода от обобщений. Хотя использование ссылок на `Object` для создания "псевдо-обобщенного" кода всегда возможно, нужно помнить, что такой код не является безопасным в отношении типов, и злоупотребление им приводит к исключениям времени выполнения.

Обобщения предотвращают такие вещи. По сути, благодаря обобщениям, то, что было ошибками времени выполнения, становится ошибками времени компиляции. Это и есть главное преимущество.

Обобщенный класс с двумя параметрами типа

Для обобщенного типа можно объявлять более одного типа-параметра. Чтобы указать два или более типа-параметра, просто используйте разделенный запятыми список. Например, следующий класс `TwoGen` — это вариант класса `Gen`, который принимает два параметра:

```
// Простой обобщенный класс с двумя
// типами-параметрами: T и V.
class TwoGen<T, V> {
T ob1;
V ob2;
// Передать конструктору ссылки на объект типа T и объект типа 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;
}
}

// Демонстрация 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);
}
}
```

Результат работы этой программы:

```
Тип T: java.lang.Integer
Тип V: java.lang.String
Значение: 88
Значение: Обобщения
```

Обратите внимание на объявление `TwoGen`:

```
class TwoGen<T, V> {
```

Она специфицирует два параметра типа: `T` и `V`, разделенные запятой. Поскольку он имеет два параметра типа, при создании объекта `TwoGen` должны быть переданы два типа аргумента, как показано ниже:

```
TwoGen<Integer, String> tgObj =
new TwoGen<Integer, String>(88, "Обобщения");
```

В этом случае `Integer` подставляется вместо `T`, а `String` — вместо `V`.

Хотя два аргумента в примере отличаются, допустимо передать в параметрах два одинаковых типа. Например, следующая строка кода вполне корректна:

```
TwoGen<String, String> x = new TwoGen<String, String>("A", "B");
```

В этом случае оба аргумента `V` и `T` будут иметь тип `String`. Конечно, если оба аргумента всегда будут одинаковы, то два параметра не обязательны.

Общая форма обобщенного класса

Синтаксис, показанный в предыдущих примерах, может быть обобщен. Так выглядит синтаксис объявления обобщенного класса:

```
class имя_класса<список_параметров_типов> { // ...
```

Ниже показан синтаксис объявления ссылки на обобщенный класс:

```
имя_класса<список_аргументов_типов> имя_переменной =
  new имя_класса<список_аргументов_типов>(список_аргументов_констант);
```

Ограниченные типы

В предыдущих примерах параметры типов могли быть заменены любыми типами классов. Это подходит ко многим случаям, но иногда удобно ограничить перечень типов, передаваемых в параметрах. Например, предположим, что вы хотите создать обобщенный класс, который содержит метод, возвращающий среднее значение массива чисел. Более того, вы хотите использовать этот класс для получения среднего значения чисел, включая целые и числа с плавающей точкой одинарной и двойной точности. То есть вы хотите специфицировать тип числовых данных обобщенно, используя параметр типа. Чтобы создать такой класс, можно попробовать что-то вроде этого:

```
// Stats пытается (безуспешно) создать
// обобщенный класс, который вычисляет
// среднее значение массива чисел
// заданного типа.
//
// 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;
}
// Возвращает double во всех случаях.
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()` в `Stats` пытается получить `double`-версию каждого числа в массиве `nums`, вызывая `doubleValue()`. Поскольку все числовые классы, такие как `Integer` и `Double`, являются подклассами `Number`, и `Number` определяет метод `doubleValue()`, этот метод доступен всем числовым классам-оболочкам. Проблема в том, что компилятор не имеет возможности узнать, что вы намерены создавать объекты `Stats`, используя только числовые типы. То есть, когда вы компилируете `Stats`, выдается сообщение об ошибке, говорящее о том, что метод `doubleValue()` не известен. Чтобы решить эту проблему, вам нужен какой-то способ сообщить компилятору, что вы собираетесь передавать в параметре `T` только числовые типы. Более того, необходим еще некоторый способ *гарантии* того, что будут передаваться *только* числовые типы.

Чтобы справиться с этой ситуацией, Java предлагает *ограниченные типы*. Когда указывается параметр типа, вы можете создать ограничение сверху, которое объявляет суперкласс, от которого все типы-аргументы должны быть унаследованы. Это достигается применением слова `extends` при указании типа параметра, как показано ниже:

```
<T extends superclass>
```

Это означает, что `T` может быть заменено только классом *`superclass`*, либо его подклассами. То есть *`superclass`* объявляет включающую верхнюю границу.

Вы можете использовать ограничение сверху, чтобы исправить класс `Stats`, показанный выше, задав верхнюю границу используемого параметра типа `Number`:

```
// В этой версии Stats тип-аргумент
// T должен быть либо Number, либо классом,
// унаследованным от Number.
class Stats<T extends Number> {
T[] nums; // массив Number или подклассов
// Передать конструктору ссылку на массив
// элементов Number или его подклассов.
Stats(T[] o) {
  nums = o;
}
// Возвратить double во всех случаях.
double average() {
  double sum = 0.0;
  for(int i=0; i < nums.length; i++)
  sum += nums[i].doubleValue();
  return sum / nums.length;
}
}
```

```
// Демонстрация 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);
  // Это не скомпилируется, потому что String не является
  // подклассом Number.
  // String strs[] = { "1", "2", "3", "4", "5" };
  // Stats<String> strob = new Stats<String>(strs);
  // double x = strob.average();
  // System.out.println("Среднее значение strob равно " + v);
}
}
```

Результат работы этой программы выглядит следующим образом:

```
Среднее значение iob равно 3.0
Среднее значение dob равно 3.3
```

Обратите внимание, что `Stats` теперь объявлен так:

```
class Stats<T extends Number> {
```

Поскольку тип `T` теперь ограничен классом `Number`, компилятор Java знает, что все объекты типа `T` могут вызывать `doubleValue()`, так как это метод класса `Number`. Это уже серьезное преимущество. Однако в качестве дополнительного бонуса ограничение `T` также предотвращает создание нечисловых объектов `Stats`. Например, если вы попытаетесь убрать комментарии в строках, находящихся в конце программы, и перекомпилировать ее, то получите ошибку времени компиляции, потому что `String` не является подклассом `Number`.

В дополнение к использованию типа класса как ограничения вы также можете также применять тип интерфейса. Фактически вы можете специфицировать в качестве ограничений множество интерфейсов. Более того, такое ограничение может включать как тип класса, так и один или более интерфейсов. В этом случае тип класса должен быть задан первым. Когда ограничение включает тип интерфейса, допустимы только типы-аргументы, реализующие этот интерфейс. Указывая ограничение, имеющее класс и интерфейс либо множество интерфейсов, применяйте операцию `&` для их объединения, например:

```
class Gen<T extends MyClass & MyInterface> { // ...
```

Здесь `T` ограничено классом по имени `MyClass` и интерфейсом `MyInteface`. То есть любой тип, переданный в `T`, должен быть подклассом `MyClass` и иметь реализацию `MyInteface`.

Использование шаблонных аргументов

Кроме того, что контроль типов удобен сам по себе, иногда он позволяет получить чрезвычайно полезные конструкции. Например, класс `Stats`, рассмотренный в предыдущем разделе, предполагает, что вы хотите добавить метод по имени `sameAvg()`, который опре-

деляет, содержат ли два объекта Stats массивы, порождающие одинаковое среднее значение, независимо от того, какого типа числовые значения в них содержатся. Например, если один объект содержит double-значения 1.0, 2.0 и 3.0, а другой объект — целочисленные значения 2, 1 и 3, то среднее значение у них будет одинаково. Один из способов реализации sameAvg() — передать ему аргумент Stats, а затем сравнивать его среднее значение со средним вызывающего объекта, возвращая true, если они равны. Например, пусть необходимо иметь возможность вызывать sameAvg(), как показано ниже:

```
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.");
```

Поначалу написание sameAvg() кажется простой задачей. Поскольку Stats — обобщенный класс и его метод average() может работать с объектами Stats любого типа, кажется, что написание sameAvg() не представляет сложности. К сожалению, проблема появляется сразу, как только вы попытаетесь объявить параметр типа Stats. Так как Stats — параметризованный тип, какой тип параметра вы укажете для Stats, когда создадите параметр типа Stats?

Вначале вы можете подумать о решении вроде такого, в котором T будет использоваться как параметр типа:

```
// Это не сработает!
// Определение эквивалентности двух средних значений.
boolean sameAvg(Stats<T> ob) {
   if(average() == ob.average())
      return true;
   return false;
}
```

С приведенным выше кодом связана проблема, которая заключается в том, что такой код будет работать только с другим объектом Stats, чей тип совпадает с вызывающим объектом. Например, если вызывающий объект имеет тип Stats<Integer>, то параметр ob также должен быть типа Stats<Integer>. Он, например, не может применяться для сравнения среднего значения Stats<Double> со средним значением Stats<Short>. Таким образом, этот подход будет работать только в очень ограниченном контексте, и не представляет собой общего (а, стало быть, и обобщенного) решения.

Чтобы создать обобщенную версию метода sameAvg(), вы должны использовать другое средство обобщений Java: аргументы-шаблоны. Аргумент-шаблон специфицируется символом ? и представляет собой неизвестный тип. Применение шаблона — единственный способ написать работающий метод sameAvg():

```
// Определение эквивалентности двух средних значений.
// Отметим использования шаблонного символа.
boolean sameAvg(Stats<?> ob) {
   if(average() == ob.average())
      return true;
   return false;
}
```

Здесь Stats<?> соответствует любому объекту Stats, что позволяет сравнивать между собой средние значения любых двух объектов Stats. Это демонстрируется в следующей программе.

```
// Использование шаблона.
class Stats<T extends Number> {
T[] nums; // массив Number или подклассов

// Передать конструктору ссылку на массив
// типа Number или его подклассов.
Stats(T[] o) {
   nums = o;
}

// Во всех случаях возвращает double.
double average() {
  double sum = 0.0;
  for(int i=0; i < nums.length; i++)
     sum += nums[i].doubleValue();
  return sum / nums.length;
}

// Определение эквивалентности двух средних.
// Обратите внимание на использование шаблонов.
boolean sameAvg(Stats<?> ob) {
  if(average() == ob.average())
      return true;
   return false;
}
}

// Демонстрация шаблона.
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 для 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 для 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 для fob равно " + x);
  // Посмотреть, какие массивы имеют одинаковые средние.
  System.out.print("Средние 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("равны.");
  else
     System.out.println("отличаются.");
}
}
```

Результат работы этой программы:

```
Среднее для iob равно 3.0
Среднее для dob равно 3.3
Среднее для fob равно is 3.0
Средние iob и dob отличаются.
Средние iob and fob равны.
```

И еще один, последний, момент: важно понимать, что шаблон не влияет на то, объект `Stats` какого конкретного типа создается. Этим управляет слово `extends` в объявлении `Stats`. Шаблон просто соответствует *корректному* объекту `Stats`.

Ограниченные шаблоны

Аргументы-шаблоны могут быть ограничены почти таким же способом, как параметры типов. Ограниченный шаблон особенно важен, когда вы создаете обобщенный тип, оперирующий иерархией классов. Чтобы понять почему, обратимся к примеру. Рассмотрим следующую иерархию классов, которые инкапсулируют 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 иерархии находится класс `TwoD`, который инкапсулирует двумерные координаты XY. Его наследник — `ThreeD` добавляет третье измерение, описывая координаты XYZ. От `ThreeD` наследуется класс `FourD`, который добавляет четвертое измерение (время), порождая четырехмерные координаты.

Ниже показан обобщенный класс, называемый `Coords`, который хранит массив координат:

```
// Этот класс хранит массив координатных объектов.
class Coords<T extends TwoD> {
   T[] coords;
   Coords(T[] o) { coords = o; }
}
```

Обратите внимание, что Coords специфицирует тип параметра, ограниченный TwoD. Это значит, что любой массив, сохраненный в объекте Coords, будет содержать объект типа TwoD или любой из его подклассов.

Теперь предположим, что вы хотите написать метод, который отображает координаты X и Y для каждого элемента в массиве coords объекта Coords. Поскольку все типы объектов Coords имеют, как минимум, пару координат (X и Y), это легко сделать с помощью шаблона:

```
static void showXY(Coords<?> c) {
   System.out.println("X Y Coordinates:");
   for(int i=0; i < c.coords.length; i++)
      System.out.println(c.coords[i].x + " " + c.coords[i].y);
   System.out.println();
}
```

Поскольку Coords — ограниченный обобщенный тип, который специфицирует TwoD как верхнюю границу, все объекты, которые можно использовать для создания объекта Coords, будут массивами типа TwoD или классов, наследуемых от TwoD. Таким образом, showXY() может отображать содержимое любого объекта Coords.

Однако, что если вы хотите создать метод, отображающий координаты X,Y и Z объекта ThreeD или FourD? Беда в том, что не все объекты Coords будут иметь три координаты, так как Coords<TwoD> будет иметь только X и Y. А поэтому — как написать метод, который будет отображать координаты X,Y и Z для Coords<ThreeD> и Coords<FourD>, в то же время предотвращая использование этого метода с объектами Coords<TwoD>? Ответ состоит в использовании *ограниченных шаблонных аргументов*.

Ограниченный шаблон специфицирует верхнюю или нижнюю границу типа аргумента. Это позволяет ограничить типы объектов, которыми будет оперировать метод. Более часто употребляемый ограниченный шаблон — это ограничивающий сверху, который создается применением оператора extends, почти таким же способом, как это делается при описании ограниченного типа.

Применяя ограниченные шаблоны, легко создать метод, отображающий координаты X, Y и Z для объекта Coords, если этот объект действительно имеет эти три координаты. Например, следующий метод showXYZ() показывает координаты элементов, сохраненных в объекте Coords, если эти элементы имеют тип ThreeD (или унаследованы от ThreeD):

```
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, что слово extends может быть добавлено к шаблону в параметре объявления c. Это устанавливает, что ? должно соответствовать любому типу до тех пор, пока он является ThreeD или унаследованным от него. То есть extends накладывает верхнее ограничение соответствия ?. Из-за этого ограничения showXYZ() может быть вызвано со ссылкой на объекты типа Coords<ThreeD> или Coords<FourD>, но не со ссылкой на Coords<TwoD>. Попытка вызвать showXYZ() со ссылкой на Coords<TwoD> вызывает ошибку времени компиляции, что обеспечивает безопасность типов.

Ниже приведена полная программа, которая демонстрирует действие ограниченного шаблонного аргумента.

```
// Ограниченные шаблонные аргументы.
// Двумерные коорбинаты.
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; }
}

// Демонстрация ограниченных шаблонов.
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 tdlocs.");
  showXY(tdlocs);       // OK, это TwoD
  // showXYZ(tdlocs);   // Ошибка, не ThreeD
  // showAll(tdlocs);   // Ошибка, не FourD
  // Теперь, создаем несколько объектов 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 fdlocs.");
  // Здесь все OK.
  showXY(fdlocs);
  showXYZ(fdlocs);
  showAll(fdlocs);
}
}
```

Результат работы этой программы выглядит следующим образом:

```
Содержимое tdlocs.
Координаты X Y:
0 0
7 9
18 4
-1 -23

Содержимое 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, не ThreeD
// showAll(tdlocs); // Ошибка, не FourD
```

Поскольку `tdlocs` — это объект `Coords<TwoD>`, он не может быть использован для вызова `showXYZ()` или `showAll()`, потому что ограничивающий шаблонный аргумент в их объявлении предотвращает это. Чтобы убедиться в этом, попробуйте убрать комментарии с упомянутых строк и попытаться скомпилировать программу. Вы получите ошибку компиляции по причине несоответствия типов.

В общем случае, для того, чтобы установить верхнюю границу шаблона, используйте следующий тип шаблонного выражения:

```
<? extends superclass>
```

здесь *`superclass`* — это имя класса, который служит верхней границей. Помните, что это включающее выражение, потому что класс, заданный в качестве верхней границы (т.е. *`superclass`*), также находится в пределах допустимых типов.

Вы также можете указать нижнюю границу шаблона, добавив выражение `super` к объявлению шаблона. Вот его общая форма:

```
<? super subclass>
```

В этом случае допустимыми аргументами могут быть только классы, которые являются суперклассами для *`subclass`*. Это исключающая конструкция, поскольку она не включает класса *`subclass`*.

Создание обобщенного метода

Как было показано в предыдущих примерах, методы внутри обобщенного класса могут использовать параметр-класс, и, следовательно, обобщения касаются также параметров методов. Однако можно объявить обобщенный метод, который сам по себе использует один или более параметров типов. Более того, можно объявить обобщенный метод, который включен в не параметризованный (необобщенный) класс.

Начнем с примера. В следующей программе объявлен необобщенный класс по имени `GenMethDemo` и статический обобщенный метод внутри класса по имени `isIn()`. Метод `isIn()` определяет, является ли объект членом массива. Он может быть использован с любым типом объектов и массивов до тех пор, пока массив содержит объекты, совместимые с типом искомого объекта.

```
// Демонстрация простого обобщенного метода.
class GenMethDemo {
// Определение, содержится ли объект в массиве.
static <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[]) {
    // Применение isIn() для Integer.
    Integer nums[] = { 1, 2, 3, 4, 5 };
    if(isIn(2, nums))
      System.out.println("2 содержится в nums");
    if(!isIn(7, nums))
      System.out.println("7 не содержится в nums");
    System.out.println();
    // Применение isIn() для String.
    String strs[] = { "один", "два", "три",
                      "четыре", "пять" };
    if(isIn("два", strs))
      System.out.println("два содержится в strs");
    if(!isIn("семь", strs))
      System.out.println("семь содержится в strs");
    // Не скомпилируется! Типы должны быть совместимыми.
    // if(isIn("два", nums))
    //  System.out.println("два содержится в strs ");
  }
  }
```

Результат работы этой программы показан ниже:

```
2 содержится в nums
7 не содержится в nums
два содержится в strs
семь не содержится в strs
```

Рассмотрим `isIn()` поближе. Для начала посмотрите, как объявлен метод в следующей строке:

```
static <T, V extends T> boolean isIn(T x, V[] y) {
```

Параметр типа объявлен *перед* типом возврата метода. Второе — тип `V` ограничен сверху типом `T`. То есть `V` либо должен тем же типом, что и `T`, либо типом его подклассов. Это отношение указывает, что `isIn()` может быть вызван только с аргументами, совместимыми между собой. Также обратите внимание, что метод `isIn()` статический, что позволяет вызывать его независимо от какого-либо объекта. Однако ясно, что обобщенные методы могут быть как статическими, так и нестатическими. Нет никаких ограничений на этот счет.

Теперь обратите внимание на то, как `isIn()` вызывается внутри `main()` — с нормальным синтаксисом вызовов, без необходимости специфицировать аргументы типа. Это потому, что типы аргументов подразумеваются автоматически, и типы `T` и `V` определяются соответственно. Например, в следующем вызове:

```
if(isIn(2, nums))
```

тип первого аргумента — `Integer` (благодаря автоупаковке), что подставляет `Integer` вместо `T`. Базовый тип второго аргумента — также `Integer`, что подставляет его и вместо `V`.

Во втором вызове используются типы `String`, и вместо типов `T` и `V` подставляются `String`.

Теперь обратите внимание на закомментированный код:

```
// if(isIn("два", nums))
//    System.out.println("два содержится в strs ");
```

Если вы уберете комментарии с этих строк и затем попытаетесь скомпилировать программу, то получите ошибку. Причина в том, что тип параметра `V` ограничен `T` конструкцией `extends` в объявлении `V`. Это значит, что `V` должно иметь либо тип `T`, либо тип его подкласса. А в этом случае первый аргумент имеет тип `String`, но второй — `Integer`, который не является подклассом `String`. Это вызовет ошибку несоответствия типов во время компиляции. Такая способность обеспечивать безопасность типов — одно из наиболее существенных преимуществ обобщенных методов.

Синтаксис, использованный для создания `isIn()`, можно обобщить. Вот синтаксис обобщенного метода:

```
<type-param-list> ret-type meth-name(param-list) { // ...
```

Во всех случаях *type-param-list* — это разделенный запятыми список параметров типа. Обратите внимание, что для обобщенного метода список параметров типа предшествует типу возврата.

Обобщенные конструкторы

Конструкторы также могут быть обобщенными, даже если их классы таковыми не являются. Например, рассмотрим следующую короткую программу:

```
// Использование обобщенного конструктора.
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();
}
}
```

Вывод этой программы:

```
val: 100.0
val: 123.5
```

Поскольку `GenCons()` специфицирует параметр обобщенного типа, который может быть подклассом `Number`, `GenCons()` можно вызывать с любым числовым типом, включая `Integer`, `Float` или `Double`. Таким образом, даже несмотря на то, что `GenCons` — необобщенный класс, его конструктор обобщен.

Обобщенные интерфейсы

В дополнение к обобщенным классам и методам вы можете объявлять обобщенные интерфейсы. Обобщенные интерфейсы специфицируются так же, как и обобщенные классы. Ниже показан пример. В нем создается интерфейс `MinMax`, объявляющий методы `min()` и `max()`, которые, как ожидается, должны возвращать минимальное и максимальное значения из некоторого множества объектов.

```
// Пример обобщенного интерфейса.
// Интерфейс Min/Max.
interface MinMax<T extends Comparable<T>> {
   T min();
   T max();
}

// Реализация MinMax
class MyClass<T extends Comparable<T>> implements MinMax<T> {
   T[] vals;
   MyClass(T[] o) { vals = o; }

// Возвращает минимальное значение из 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;
}

// Возвращает максимальное значение из 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 inums: " + iob.max());
  System.out.println("Минимальное значение в inums: " + iob.min());
  System.out.println("Максимальное значение в chs: " + cob.max());
  System.out.println("Минимальное значение в chs: " + cob.min());
}
}
```

Результат работы этой программы:

```
Максимальное значение в inums: 8
Минимальное значение в inums: 2
Максимальное значение в chs: w
Минимальное значение в chs: b
```

Хотя большинство аспектов этой программы просты для понимания, некоторые ключевые моменты следует пояснить. Во-первых, обратите внимание на то, как объявлен MinMax:

```
interface MinMax<T extends Comparable<T>> {
```

В общем случае, обобщенный интерфейс объявляется так же, как и обобщенный класс. В этом случае тип параметра T имеет верхнюю границу Comparable — интерфейс, определенный в java.lang. Класс, который реализует Comparable, определяет объекты, которые могут быть упорядочены. То есть требование, чтобы T был ограничен сверху Comparable, гарантирует, что MinMax может быть использован только с объектами, которые могут сравниваться между собой. (Более подробную информацию о Comparable можно найти в главе 16.) Отметим, что Comparable сам по себе также является обобщенным интерфейсом (он был преобразован в обобщенную форму в JDK 5). Он принимает параметр типа объектов, которые должны сравниваться.

Далее MinMax реализует класс MyClass. Вот как выглядит объявление класса MyClass:

```
class MyClass<T extends Comparable<T>> implements MinMax<T> {
```

Обратите особое внимание на способ, которым параметр типа T объявлен в MyClass и затем передан MinMax. Поскольку MinMax требует типа, который реализует Comparable, реализующий класс (в данном случае — MinMax) должен специфицировать то же ограничение. Более того, однажды установленное, это ограничение уже не нужно повторять в операторе implements. Фактически так поступать некорректно. Например, следующая строка неверна и не может быть скомпилирована:

```
// Это не правильно!
class MyClass<T extends Comparable<T>>
  implements MinMax<T extends Comparable<T>> {
```

Однажды установленный, параметр типа просто передается интерфейсу без последующих модификаций.

Вообще если класс реализует обобщенный интерфейс, то такой класс также должен быть обобщенным, по крайней мере, в тех пределах, где он принимает параметр типа, переданный интерфейсу. Например, следующая попытка объявления MyClass вызовет ошибку:

```
class MyClass implements MinMax<T> { // Неверно!
```

Поскольку MyClass не объявляет параметра, нет способа передать его MinMax. В этом случае идентификатор T просто неизвестен, и компилятор выдаст ошибку. Конечно, если класс реализует *специфический тип* обобщенного интерфейса, как показано ниже:

```
class MyClass implements MinMax<Integer> { // OK
```

то реализующий класс не обязан быть обобщенным.

Обобщенный интерфейс представляет два преимущества. Во-первых, он может быть реализован для разных типов данных. Во-вторых, он позволяет включить ограничения на типы данных, для которых он может быть реализован. В примере MinMax вместо T могут быть подставлены только типы, совместимые с Comparable.

Вот общий синтаксис обобщенного интерфейса:

```
interface имя_интерфейса<список_параметров_типов> { // ...
```

Здесь *список_параметров_типов* — разделенный запятыми список параметров типов. Когда реализуется обобщенный интерфейс, вы должны специфицировать аргументы типов, как показано ниже:

```
class имя_класса<список_параметров_типов>
  implements имя_интерфейса<список_аргументов_типов> {
```

Raw-типы и унаследованный код

Поскольку поддержка обобщений — новое средство, существует необходимость в том, чтобы Java предоставляла некоторый способ трансляции старого кода, разработанного до появления обобщений. На момент написания этой книги доступны многие миллионы строк унаследованного старого кода, который должен и оставаться функциональным, и быть совместимым с обобщениями. Код, предшествующий обобщениям, должен иметь возможность работать с обобщенным кодом, и обобщенный код должен работать со старым кодом.

Чтобы облегчить переход к обобщениям, Java позволяет обобщенным классам быть использованными без аргументов типа. Это создает *raw-тип* ("сырой" тип) для класса. Этот raw-тип совместим с унаследованным кодом, который не имеет представления об обобщенном синтаксисе. Главный недостаток использования raw-типа в том, что безопасность типов утрачивается.

Вот пример, демонстрирующий raw-тип в действии:

```
// Демонстрация raw-типа.
class Gen<T> {
T ob; // Объявление объекта типа T

// Передача конструктору ссылки
// на объект типа T.
Gen(T o) {
  ob = o;
}

// Возврат ob.
T getob() {
  return ob;
}
}

// Демонстрация raw-типа.
class RawDemo {
public static void main(String args[]) {
  // Создать объект Gen для Integer.
  Gen<Integer> iOb = new Gen<Integer>(88);

  // Создать объект Gen для String.
  Gen<String> strOb = new Gen<String>("Обобщенный тест");

  // Создать объект raw-типа Gen и дать ему значение Double.
  Gen raw = new Gen(new Double(98.6));

  // Приведение необходимо, поскольку тип неизвестен.
  double d = (Double) raw.getob();
  System.out.println("значение: " + d);

  // Использование raw-типов может вызвать исключения
  // времени выполнения. Ниже представлены некоторые примеры.
```

```
    // Следующее приведение вызовет ошибку времени выполнения!
    // int i = (Integer) raw.getob(); // ошибка времени выполнения
    // Это присваивание нарушает безопасность типов.
    strOb = raw; // OK, но потенциально неверно
    // String str = strOb.getob();    // ошибка времени выполнения

    // Это присваивание также нарушает безопасность типов.
    raw = iOb; // OK, но потенциально неверно
    // d = (Double) raw.getob();      // ошибка времени выполнения
  }
}
```

С этой программой связано несколько интересных моментов. Во-первых, следующее объявление создает класс `Gen` raw-типа:

```
Gen raw = new Gen(new Double(98.6));
```

Обратите внимание, что никаких аргументов типа не указывается. По сути, оператор создает объект `Gen`, чей тип `T` заменяется `Object`.

Raw-типы не обеспечивают безопасности. То есть переменный raw-типа можно присвоить ссылку на любой тип объектов `Gen`. Обратное также возможно: переменной специфического типа `Gen` можно присвоить ссылку на объект raw-типа `Gen`. Однако обе операции потенциально небезопасны, так как механизм проверки типов при этом обходится.

Недостаток безопасности иллюстрируется закомментированными строками в конце программы. Рассмотрим каждую из них. Вот первая ситуация:

```
// int i = (Integer) raw.getob();  // ошибка времени выполнения
```

В этом операторе получается значение `ob` внутри `raw`, и это значение приводится к типу `Integer`. Проблема в том, что `raw` содержит значение `Double` вместо целого. Однако это не может быть обнаружено на этапе компиляции, поскольку тип `raw` неизвестен. То есть этот оператор вызовет сбой во время выполнения.

Следующая последовательность присваивает `strOb` (ссылке на `Gen<String>`) ссылку на объект `Gen`:

```
strOb = raw;                        // OK, но потенциально неверно
// String str = strOb.getob();      // ошибка времени выполнения
```

Это присваивание само по себе синтаксически корректно, но сомнительно. Поскольку `strOb` имеет тип `Gen<String>`, предполагается, что оно содержит `String`. Однако после присваивания объект, на который ссылается `strOb`, содержит `Double`. То есть во время выполнения, когда предпринимается попытка присвоить `strOb` переменной `str`, происходит ошибка времени выполнения, так как `strOb` теперь содержит `Double`. То есть присваивание raw-ссылки обобщенной ссылке минует механизм проверки типов.

Следующая последовательность представляет собой противоположный случай:

```
raw = iOb;                          // OK, но потенциально неверно
// d = (Double) raw.getob();        // ошибка времени выполнения
```

Здесь обобщенная ссылка присваивается переменной raw-ссылки. Хотя это синтаксически корректно, но также может привести к проблемам, как показывает вторая строка. В этом случае `raw` теперь ссылается на объект, содержащий `Integer`, но приведение предполагает, что в нем содержится `Double`. Эта ошибка не может быть предотвращена во время компиляции. Вместо этого она проявляется во время выполнения.

Из-за того, что raw-типы представляют опасность, `javac` отображает *непроверенные предупреждения*, когда обнаруживает, что их использование может нарушить безопас-

ность типов. В предыдущей программе следующие строки вызовут появление таких предупреждений:

```
Gen raw = new Gen(new Double(98.6));
strOb = raw;    // OK, но потенциально неверно
```

В первой строке происходит вызов конструктора Gen без аргумента типа, что вызывает предупреждение. Во второй строке происходит присваивание raw-ссылки обобщенной переменной, что также вызывает появление предупреждения.

На первый взгляд, может показаться, что эта строка также должна порождать предупреждение, но этого не происходит:

```
raw = iOb;      // OK, но потенциально неверно
```

Здесь не выдается никаких предупреждений компилятора, потому что присваивание не вызывает никакой *дополнительной* потери безопасности типов сверх той, что уже происходит при создании raw.

Один заключительный момент: вы должны ограничивать использование raw-типов теми случаями, когда приходится смешивать унаследованный код с новым, обобщенным. Raw-типы — это просто средство переноса, а не что-то такое, что должно применяться в новом коде.

Иерархии обобщенных классов

Обобщенные классы могут быть частью иерархии классов — так же, как и любые другие необобщенные классы. То есть обобщенный класс может выступать в качестве суперкласса или подкласса. Ключевое отличие между обобщенными и необобщенными иерархиями состоит в том, что в обобщенной иерархии любые аргументы типов, необходимые обобщенному суперклассу, всеми подклассами должны передаваться по иерархии вверх. Это похоже на способ, которым аргументы конструкторов передаются по иерархии.

Использование обобщенного суперкласса

Ниже приведен пример иерархии, которая использует обобщенный суперкласс:

```
// Простая иерархия обобщенных классов.
class Gen<T> {
   T ob;
   Gen(T o) {
      ob = o;
   }

   // Возвращает ob.
   T getob() {
      return ob;
   }
}

// Подкласс Gen.
class Gen2<T> extends Gen<T> {
   Gen2(T o) {
      super(o);
   }
}
```

В этой иерархии Gen2 расширяет обобщенный класс Gen. Обратите внимание на то, как в следующей строке объявлен Gen2:

```
class Gen2<T> extends Gen<T> {
```

Параметр типа T специфицирован в Gen2 и также передается Gen в операторе extends. Это значит, что тип, переданный Gen2, будет также передан Gen. Например, следующее объявление

```
Gen2<Integer> num = new Gen2<Integer>(100);
```

передает Integer как параметр типа классу Gen. То есть ob внутри части Gen класса Gen2 будет иметь тип Integer. Отметим также, что Gen2 никак не использует параметр типа T, кроме того, что передает его суперклассу Gen. Даже если подкласс обобщенного суперкласса никак не нуждается в том, чтобы быть обобщенным, несмотря на это он все же должен специфицировать параметры типа, необходимые его обобщенному суперклассу.

Конечно, при необходимости подкласс может добавлять свои собственные параметры типа. Например, ниже показан вариант предыдущей иерархии, в котором Gen2 добавляет собственный параметр типа:

```
// Подкласс может добавлять свои собственные параметры типа.
class Gen<T> {
T ob; // Объявление объекта типа T
// Передача конструктору
// ссылки на объект типа T.
Gen(T o) {
  ob = o;
}
// Возвращает ob.
T getob() {
  return ob;
}
}

// Подкласс Gen, который определяет
// второй параметр типа по имени V.
class Gen2<T, V> extends Gen<T> {
V ob2;
Gen2(T o, V o2) {
  super(o);
  ob2 = o2;
}
V getob2() {
  return ob2;
}
}

// Создание объекта типа Gen2.
class HierDemo {
public static void main(String args[]) {
  // Создание объектов Gen2 для String и Integer.
  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 Gen2, показанное в следующей строке:

```
class Gen2<T, V> extends Gen<T> {
```

Здесь T — тип, переданный Gen, а V — тип, специфичный для Gen2. V используется для объявления объекта, названного ob2, а также в качестве типа возврата метода getob2(). В main() создается объект Gen2 с типом String для параметра T и типом Integer для параметра V.

Программа выдает следующий вполне ожидаемый результат:

```
Значение равно: 99
```

Обобщенный подкласс

Абсолютно приемлемо, когда суперклассом для обобщенного класса выступает класс необобщенный. Например, рассмотрим 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; // Объявление объекта типа T
// Передать конструктору объект
// типа T.
Gen(T o, int i) {
  super(i);
  ob = o;
}
// Возвращает ob.
T getob() {
  return ob;
}
}
// Создать объект Gen.
class HierDemo2 {
public static void main(String args[]) {
  // Создать объект Gen для String.
  Gen<String> w = new Gen<String>("Добро пожаловать", 47);
  System.out.print(w.getob() + " ");
  System.out.println(w.getnum());
}
}
```

Результат работы этой программы показан ниже:

```
Добро пожаловать 47
```

Обратите внимание на то, как в этой программе Gen наследуется от NonGen:

```
class Gen<T> extends NonGen {
```

Поскольку NonGen — необобщенный класс, никакие аргументы типа не указываются. То есть даже если Gen объявляет тип-параметр T, он не требуется (и не может быть использован) NonGen. То есть Gen наследуется от NonGen обычным способом. Никаких специальных условий не требуется.

Сравнение типов обобщенной иерархии во время выполнения

Вспомните операцию получения информации о типе времени выполнения — instanceof, которая была описана в главе 13. Как уже объяснялось, instanceof определяет, является ли объект экземпляром класса. Она возвращает true, если объект относится к указанному типу либо может быть приведен к этому типу. Операция instanceof может быть применима к объектам обобщенных классов. Следующий класс демонстрирует следствия совместимости типов в обобщенных иерархиях.

```
// Использование операции instanceof с иерархией обобщенных классов.
class Gen<T> {
T ob;
Gen(T o) {
  ob = o;
}
// Возвращает ob.
T getob() {
  return ob;
}
}
// Подкласс Gen.
class Gen2<T> extends Gen<T> {
Gen2(T o) {
  super(o);
}
}
// Демонстрация определения идентификатора
// типа в иерархии обобщенных классов.
class HierDemo3 {
public static void main(String args[]) {
  // Создать объект Gen для Integer.
  Gen<Integer> iOb = new Gen<Integer>(88);
  // Создать объект Gen2 для Integer.
  Gen2<Integer> iOb2 = new Gen2<Integer>(99);
  // Создать объект Gen2 для String.
  Gen2<String> strOb2 = new Gen2<String>("Обобщенный тест");
  // Проверить, является ли iOb2 какой-то из форм Gen2.
  if(iOb2 instanceof Gen2<?>)
     System.out.println("iOb2 является экземпляром Gen2");
  // Проверить, является ли iOb2 какой-то из форм Gen.
  if(iOb2 instanceof Gen<?>)
     System.out.println("iOb2 является экземпляром Gen");
  System.out.println();
  // Проверить, является ли strOb2 объектом Gen2.
  if(strOb2 instanceof Gen2<?>)
     System.out.println("strOb2 является экземпляром Gen2");
```

```
    // Проверить, является ли strOb2 объектом Gen.
    if(strOb2 instanceof Gen<?>)
       System.out.println("strOb2 является экземпляром Gen");
    System.out.println();
    // Проверить, является ли iOb экземпляром Gen2, что не так.
    if(iOb instanceof Gen2<?>)
       System.out.println("iOb является экземпляром Gen2");
    // Проверить, является ли iOb экземпляром Gen, что так и есть.
    if(iOb instanceof Gen<?>)
       System.out.println("iOb является экземпляром Gen");
    // Следующее не скомпилируется, потому что информация
    // об обобщенном типе во время выполнения отсутствует.
    // if(iOb2 instanceof Gen2<Integer>)
    // System.out.println("iOb2 является экземпляром Gen2<Integer>");
  }
  }
```

Вывод программы показан здесь:

```
iOb2 является экземпляром Gen2
iOb2 является экземпляром Gen
strOb2 является экземпляром Gen2
strOb2 является экземпляром Gen
iOb является экземпляром Gen
```

В этой программе `Gen2` — подкласс `Gen`, который обобщен по типу параметра `T`. В `main()` создается три объекта. Первый, `iOb` — объект типа `Gen<Integer>`. Второй, `iOb2` — экземпляр `Gen2<Integer>`. И, наконец, третий, `strOb2` — объект типа `Gen2<String>`.

Затем программа выполняет проверку `instanceof` типа `iOb2`:

```
// Проверить, является ли iOb2 какой-то из форм Gen2.
if(iOb2 instanceof Gen2<?>)
   System.out.println("iOb2 является экземпляром Gen2");

// Проверить, является ли iOb2 какой-то из форм Gen.
if(iOb2 instanceof Gen<?>)
  System.out.println("iOb2 является экземпляром Gen");
```

Как показывает вывод, обе проверки успешны. В первом тесте `iOb2` проверяется на соответствие типу `Gen2<?>`. Этот тест успешен, потому что он просто подтверждает, что `iOb2` является объектом какого-то из типов `Gen2`. Применение шаблонного символа позволяет `instanceof` определить, относится ли `iOb2` к какому-то из типов `Gen2`. Далее `iOb` проверяется на принадлежность к типу суперкласса `Gen<?>`. Это также верно, поскольку `iOb2` представляет собой некоторую форму суперкласса `Gen`. Следующие несколько строк в `main()` показывают ту же последовательность (и некоторый результат) для `strOb2`.

Далее `iOb`, являющийся экземпляром суперкласса `Gen<Integer>`, тестируется в следующих строках:

```
// Проверить, является ли iOb экземпляром Gen2, что не так.
if(iOb instanceof Gen2<?>)
   System.out.println("iOb является экземпляром Gen2");

// Проверить, является ли iOb экземпляром Gen, что так и есть.
if(iOb instanceof Gen<?>)
  System.out.println("iOb является экземпляром Gen");
```

Первый оператор if возвращает false, поскольку iOb не является никакой из форм типа Gen2. Следующая проверка успешна, потому что iOb является некоторым типом объекта Gen.

Теперь посмотрим внимательно на закомментированные строки:

```
// Следующее не скомпилируется, потому что информация
// об обобщенном типе отсутствует во время выполнения.
// if(iOb2 instanceof Gen2<Integer>)
//     System.out.println("iOb2 является экземпляром Gen2<Integer>");
```

Как следует из текста комментария, эти строки не скомпилируются, потому что они пытаются сравнить iOb2 со специфическим типом Gen2 — в данном случае с Gen2<Integer>. Помните, что во время выполнения не существует никакой доступной информации о типе. Таким образом, у instanceof нет способа узнать, является ли iOb2 экземпляром Gen2<Integer> или нет.

Приведение

Вы можете приводить один экземпляр обобщенного класса к другому, только если они между собой совместимы и их аргументы типов одинаковы. Например, для предыдущей программы следующее приведение корректно:

```
(Gen<Integer>) iOb2  // допустимо
```

потому что iOb2 является экземпляром Gen<Integer>. Однако следующее приведение:

```
(Gen<Long>) iOb2     // недопустимо
```

недопустимо, поскольку iOb2 не является экземпляром Gen<Long>.

Переопределение методов в обобщенном классе

Метод обобщенного класса, как и любой другой метод, может быть переопределен. Например, рассмотрим следующую программу, в которой переопределяется метод getob():

```
// Переопределение обобщенного метода в обобщенном классе.
class Gen<T> {
T ob; // Объявление объекта типа T

// Передать конструктору ссылку
// на объект типа T.
Gen(T o) {
  ob = o;
}

// Возвращение ob.
T getob() {
  System.out.print("getob() класса Gen: " );
  return ob;
}
}

// Подкласс Gen, переопределяющий getob().
class Gen2<T> extends Gen<T> {
Gen2(T o) {
  super(o);
}
```

```
// Переопределение getob().
T getob() {
  System.out.print("getob() класса Gen2: ");
  return ob;
}
}

// Демонстрация переопределения обобщенных методов.
class OverrideDemo {
public static void main(String args[]) {
  // Создание объекта Gen для Integer.
  Gen<Integer> iOb = new Gen<Integer>(88);
  // Создание объекта Gen2 для Integer.
  Gen2<Integer> iOb2 = new Gen2<Integer>(99);
  // Создание объекта Gen2 для String.
  Gen2<String> strOb2 = new Gen2<String>("Обобщенный тест");
  System.out.println(iOb.getob());
  System.out.println(iOb2.getob());
  System.out.println(strOb2.getob());
}
}
```

Результат работы этой программы показан ниже:

```
getob() класса Gen: 88
getob() класса Gen2: 99
getob() класса Gen2: Обобщенный тест
```

Как подтверждает вывод, переопределенная версия `getob()` вызывается для объекта `Gen2`, но для объектов типа `Gen` вызывается версия суперкласса.

Очистка

Обычно детали того, как компилятор Java трансформирует исходный текст в объектный код, знать не нужно. Однако в случае с обобщениями некоторое общее представление о процессе важно, поскольку объясняет, как работает механизм обобщения — и почему иногда его поведение несколько необычно. По этой причине мы приведем краткое описание того, как обобщения реализованы в Java.

Важное ограничение, которое было наложено на способ реализации обобщений в Java, заключалось в том, что необходимо было обеспечить совместимость с предыдущими версиями Java. Только представьте, что обобщенный код должен быть совместимым со старым, не обобщенным, кодом. То есть любые изменения в синтаксисе языка Java либо в JVM должны были избегать разрушения старого кода. Способ, которым Java реализует обобщения для удовлетворения этому требованию — это так называемая *очистка* (erasure).

В общих чертах вот как она работает. При компиляции вашего Java-кода вся информация об обобщенных типах удаляется (чистится). Это означает замену параметров типа их ограничивающим типом, который является `Object`, если только никакого явного ограничения не указано, с последующим использованием необходимых приведений (как того требуют аргументы типа) для поддержки совместимости типов с типами, указанными в аргументах. Компилятор также обеспечивает эту совместимость типов. Такой подход к обобщениям означает, что никакой информации о типах во время выполнения не существует. Это просто механизм автоматической обработки исходного кода.

Чтобы лучше понять, как работает очистка, рассмотрим два следующих класса:

```
// Здесь T ограничен типом Object по умолчанию.
class Gen<T> {
   T ob; // Здесь T будет заменен Object
   Gen(T o) {
      ob = o;
   }
   // Возвращает ob.
   T getob() {
      return ob;
   }
}
// Здесь T ограничен String.
class GenStr<T extends String> {
   T str; // Здесь T будет заменен String
   GenStr(T o) {
      str = o;
   }
   T getstr() { return str; }
}
```

После того, как эти классы компилируются, `T` в `Gen` будет заменен `Object`. `T` в `GenStr` будет заменен `String`. Вы можете убедиться в этом, запустив `javap` на скомпилированных классах. Результат будет выглядеть так:

```
class Gen extends java.lang.Object{
   java.lang.Object ob;
   Gen(java.lang.Object);
   java.lang.Object getob();
}
class GenStr extends java.lang.Object{
   java.lang.String str;
   GenStr(java.lang.String);
   java.lang.String getstr();
}
```

Внутри кода `Gen` и `GenStr` для обеспечения корректной типизации применяются приведение. Например, следующая последовательность:

```
Gen<Integer> iOb = new Gen<Integer>(99);
int x = iOb.getob();
```

будет скомпилирована, как если бы она была написана так:

```
Gen iOb = new Gen(99);
int x = (Integer) iOb.getob();
```

Благодаря очистке, некоторые вещи работают несколько иначе, чем может показаться. Например, рассмотрим короткую программу, создающую два объекта обобщенного класса `Gen`:

```
class GenTypeDemo {
public static void main(String args[]) {
  Gen<Integer> iOb = new Gen<Integer>(99);
  Gen<Float> fOb = new Gen<Float>(102.2F);
  System.out.println(iOb.getClass().getName());
  System.out.println(fOb.getClass().getName());
}
}
```

Вывод этой программы показан ниже:

```
Gen
Gen
```

Как видите, типом и iOb, и fOb является Gen, а не Gen<Integer> и Gen<Float>, как вы могли ожидать. Помните, что все параметры типов во время компиляции удаляются. Во время выполнения существуют только raw-типы.

Методы-мосты

Иногда компилятору приходится добавлять классу так называемый *метод-мост* (bridge method), чтобы справиться с ситуациями, когда результат очистки типов в перегруженном методе подкласса не совпадает с тем, что получается при очистке в суперклассе. В этом случае генерируется метод, который использует очистку типов суперкласса, и этот метод вызывает соответствующий метод подкласса, выполняющий очистку. Конечно, методы-мосты появляются только на уровне байт-кода, невидимы для вас и недоступны для непосредственного вызова.

Несмотря на то что методы-мосты — это не то, в чем вы обычно нуждаетесь и с чем имеете дело, все же полезно рассмотреть ситуацию, в которой они генерируются. Взгляните на следующую программу:

```
// Ситуация, в которой генерируется метод-мост.
class Gen<T> {
T ob; // объявить объект типа T
// Передать конструктору ссылку на
// объект типа T.
Gen(T o) {
  ob = o;
}
// Возвращает ob.
T getob() {
  return ob;
}
}
// Подкласс Gen.
class Gen2 extends Gen<String> {
Gen2(String o) {
  super(o);
}
// String-ориентированная перегрузка getob().
String getob() {
  System.out.print("Вызван String getob(): ");
  return ob;
}
}
// Демонстрация ситуации, в которой необходим метод-мост.
class BridgeDemo {
public static void main(String args[]) {
  // Создать объект Gen2 для Strings.
  Gen2 strOb2 = new Gen2("Обобщенный тест");
  System.out.println(strOb2.getob());
}
}
```

В этой программе Gen2 расширяет Gen, но делает это с использованием специфичной String-версии Gen, как показывает следующее объявление:

```
class Gen2 extends Gen<String> {
```

Более того, внутри Gen2 метол getob() переопределен с типом возврата String:

```
// String-ориентированная перегрузка getob().
String getob() {
   System.out.print("Вызван String getob(): ");
   return ob;
}
```

Все это совершенно допустимо. Единственная проблема в том, что из-за очистки типов ожидаемая форма getob() будет выглядеть так:

```
Object getob() { // ...
```

Чтобы справиться с этой проблемой, компилятор генерирует метод-мост с показанной выше сигнатурой, который вызывает String-версию. То есть, если вы посмотрите на интерфейс класса Gen2 с помощью javap, то увидите следующие методы:

```
class Gen2 extends Gen{
   Gen2(java.lang.String);
   java.lang.String getob();
   java.lang.Object getob(); // метод-мост
}
```

Как видите, сюда включен метод-мост. (Комментарий добавлен автором, а не javap.)

Последнее замечание о методах-мостах. Обратите внимание, что единственная разница между двумя методами getob() заключается в типе возврата. Обычно это вызывает ошибку, но поскольку это происходит не в исходном коде, проблема не возникает и JVM успешно справляется с этой ситуацией.

Ошибки неоднозначности

Включение в язык обобщений породило новый тип ошибок, от которых вам нужно защищаться: *неоднозначность* (ambiquity). Ошибки неоднозначности случаются, когда очистка порождает два внешне разных обобщенных объявления, разрешаемых в виде одного очищенного типа, что вызывает конфликт. Вот пример, который включает перегрузку методов:

```
// Неоднозначность порождается очисткой перегруженных методов.
class MyGenClass<T, V> {
   T ob1;
   V ob2;
   // ...
   // Эти два перегруженных метода неоднозначны
   // и не скомпилируются.
   void set(T o) {
      ob1 = o;
   }
   void set(V o) {
      ob2 = o;
  }
}
```

Обратите внимание, что MyGenClass объявляет два обобщенных типа: T и V. Внутри MyGenClass предпринимается попытка перегрузить set() на основе параметров T и V. Это выглядит резонным, потому что кажется, что T и V — разные типы. Однако здесь возникают две проблемы неоднозначности.

Первая — судя по тому, как написан MyGenClass, нет требования, чтобы T и V были разными типами. Например, в принципе совершенно корректно сконструировать объект MyGenClass следующим образом:

```
MyGenClass<String, String> obj = new MyGenClass<String, String>()
```

В этом случае T и V будут заменены String. Это делает обе версии set() идентичными, что, конечно же, представляет собой ошибку.

Вторая, более фундаментальная проблема состоит в том, что очистка типов приводит обе версии метода к следующему виду:

```
void set(Object o) { // ...
```

То есть перегрузка set(), как пытается сделать MyGenClass, в действительности неоднозначна.

Ошибки неоднозначности бывает трудно исправить. Например, если вы знаете, что V всегда будет неким подтипом String, то можете попытаться исправить MyGenClass, переписав его объявление следующим образом:

```
class MyGenClass<T, V extends String> { // почти OK!
```

Это позволит скомпилировать MyGenClass, и вы даже сможете создавать экземпляры объектов этого класса примерно так:

```
MyGenClass<Integer, String> x = new MyGenClass<Integer, String>();
```

Это работает, потому что Java может аккуратно определить, когда какой метод должен быть вызван. Однако неоднозначность возникнет, когда вы попытаетесь выполнить строку:

```
MyGenClass<String, String> x = new MyGenClass<String, String>();
```

В этом случае, поскольку и T, и V являются String, то какую версию set() нужно вызвать?

Честно говоря, в предыдущем примере было бы лучше использовать два метода с разными именами, вместо того чтобы перегружать set(). Часто разрешение неоднозначности требует реструктуризации кода, потому что неоднозначность свидетельствует о концептуальной ошибке в дизайне.

Некоторые ограничения обобщений

Существует несколько ограничений, о которых вы должны помнить, применяя обобщения. Они включают создание объектов типа параметров, статических членов, исключений и массивов. Каждое из них мы рассмотрим далее.

Нельзя создавать экземпляр типа параметра

Создавать экземпляр типа параметра невозможно. Например, рассмотрим такой класс:

```
// Нельзя создавать экземпляр типа T.
class Gen<T> {
   T ob;
   Gen() {
      ob = new T(); // Недопустимо!!!
   }
}
```

Здесь выполняется недопустимая попытка создать экземпляр `T`. Причину просто понять: поскольку `T` не существует во время выполнения, как компилятор может узнать, какого типа объект следует создать? Вспомните, что очистка удаляет все параметры типа в процессе компиляции.

Ограничения на статические члены

Никакой `static`-член не может использовать тип параметра, объявленный в его классе. Например, все `static`-члены этого класса являются недопустимыми:

```
class Wrong<T> {
// Неверно, нельзя создать статические переменные типа T.
static T ob;
// Неверно, ни один статический метод не может использовать T.
static T getob() {
  return ob;
}
// Неверно, ни один статический метод не может иметь доступ
// к объекту типа T.
static void showob() {
  System.out.println(ob);
}
}
```

Несмотря на то что вы не можете объявить `static`-члены, которые используют тип параметра, объявленный в окружающем классе, вы *можете* объявлять обобщенные `static` методы, определяющие их собственные параметры типа, как это делалось ранее в настоящей главе.

Ограничения обобщенных массивов

Существуют два важных ограничения обобщений, касающиеся массивов. Во-первых, вы не можете создать экземпляр массива, чей базовый тип — параметр типа. Во-вторых, вы не можете создать массив специфичных для типа обобщенных ссылок. В следующей короткой программе демонстрируются обе ситуации.

```
// Обобщения и массивы.
class Gen<T extends Number> {
T ob;
T vals[]; // OK
Gen(T o, T[] nums) {
  ob = o;
  // Этот оператор неверен.
  // vals = new T[10]; // нельзя создавать массив объектов T
  // Однако 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 специфичных для типа обобщенных ссылок.
  // Gen<Integer> gens[] = new Gen<Integer>[10]; // Неверно!
  // Это верно.
  Gen<?> gens[] = new Gen<?>[10]; // OK
}
}
```

Как показывает эта программа, объявлять ссылку на массив объектов `T` допустимо, как это сделано в строке:

```
T vals[]; // OK
```

Тем не менее, нельзя создать массив объектов `T`, как показано в следующей закомментированной строке:

```
// vals = new T[10]; // нельзя создавать массив объектов T
```

Причина, по которой нельзя создать массив объектов типа `T`, связана с тем, что `T` не существует во время выполнения, а потому у компилятора нет способа знать, массив элементов какого типа в действительности надо создавать.

Однако вы можете передать ссылку на совместимый по типу массив конструктору `Gen()`, когда объект создается, и присвоить эту ссылку `vals`, как это делается в программе в строке:

```
vals = nums; // можно присвоить ссылку существующему массиву
```

Это работает, поскольку массив, переданный `Gen`, имеет известный тип, который будет тем же типом, что и `T` на момент создания объекта.

Обратите внимание, что внутри `main()` вы не можете объявить массив ссылок на объекты специфического обобщенного типа. То есть строка

```
// Gen<Integer> gens[] = new Gen<Integer>[10]; // Неверно!
```

не скомпилируется. Массивы специфических обобщенных типов попросту недопустимы, так как они могут нарушить безопасность типов.

Вы *можете* создать массив ссылок на обобщенный тип, если используете шаблоны:

```
Gen<?> gens[] = new Gen<?>[10]; // OK
```

Этот подход лучше применения массивов raw-типов, поскольку, по крайней мере, некоторые проверки типа по-прежнему могут быть выполнены компилятором.

Ограничения обобщенных исключений

Обобщенный класс не может расширять `Throwable`. Это значит, что вы не сможете создать обобщенных классов исключений.

Заключительные мысли по поводу обобщений

Обобщения — это мощное расширение языка Java, поскольку они упрощают создание безопасного в отношении типов и повторно используемого кода. Хотя обобщенный синтаксис поначалу может показаться несколько громоздким, после длительного применения он станет вашей второй натурой. Обобщенный код станет частью будущего для всех программистов на Java.

II ЧАСТЬ

Библиотека Java

15
ГЛАВА

Обработка строк

Краткий обзор обработки строк в Java был представлен в главе 7. В настоящей главе мы рассмотрим эту тему подробнее. Как и в других языках программирования, в Java *строка* — это последовательность символов. Но в отличие от многих других языков, в которых строки реализованы как массивы символов, в Java строки реализованы в виде объектов типа `String`.

Реализация строк в виде встроенных объектов позволяет Java обеспечить полный комплект средств, делающих управления строками удобным. Например, Java предоставляет методы для сравнения двух строк, поиска подстрок, объединения двух строк и изменения регистра символов в строке. Кроме того, объекты `String` могут быть сконструированы множеством разных способов, что позволяет легко получать строки, когда они требуются.

Что несколько неожиданно, так это тот факт, что когда вы создаете объект типа `String`, то вы создаете строку, которая не может быть изменена. То есть, как только объект `String` создан, вы не можете изменить символы, образующие строку. На первый взгляд это может показаться серьезным ограничением. Однако на самом деле это не так уж важно. Вы можете осуществлять любые операции над строками. Особенность в том, что всякий раз, когда вам нужна измененная версия существующей строки, создается новый объект `String`, включающий все модификации. Оригинальная строка остается неизменной. Этот подход используется потому, что фиксированная, неизменная строка может быть реализована более эффективно, нежели изменяемая. Для тех случаев, когда нужны модифицируемые строки, Java предлагает два выбора: `StringBuffer` и `StringBuilder`. Оба содержат строки, которые могут быть изменены после того, как созданы.

Классы `String`, `StringBuffer` и `StringBuilder` определены в пакете `java.lang`. Поэтому они доступны всем программистам автоматически. Все они объявлены с модификатором `final`, что означает, что ни от одного из них нельзя порождать подклассы. Это позволяет осуществить некоторую оптимизацию, которая повышает производительность общих операций со строками. Все три класса реализуют интерфейс `CharSequence`.

И последнее: когда говорится о том, что строки в объектах типа `String` неизменяемы, это означает, что содержимое экземпляра `String` не может быть изменено после его создания. Однако переменная, объявленная как ссылка на `String` в любой момент может быть переназначена так, чтоб указывать на другой объект `String`.

Конструкторы строк

Класс `String` поддерживает несколько конструкторов. Чтобы создать пустой объект типа `String`, вызывается конструктор по умолчанию. Например, следующий оператор создает экземпляр `String`, не содержащий в себе символов:

```
String s = new String();
```

Часто будет требоваться создать строку, которая содержит начальное значение. Класс `String` предлагает множество конструкторов для этого. Чтобы создать `String`, инициализированный массивом символов, используйте следующий конструктор:

```
String(char chars[])
```

Вот примеры:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
```

Этот конструктор инициализирует `s` строкой “abc”.

Вы можете задать поддиапазон символьного массива в качестве инициализирующей строки с помощью следующего конструктора:

```
String(char chars[ ], int startIndex, int numChars)
```

Здесь *`startIndex`* указывает начало диапазона, а *`numChars`* — количество символов, которые нужно использовать. Вот пример:

```
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
String s = new String(chars, 2, 3);
```

Это инициализирует строку `s` символами “cde”.

Вы можете сконструировать объект `String`, который содержит ту же последовательность символов, что и другой объект `String`, используя конструктор:

```
String(String strObj)
```

Здесь *`strObj`* — объект `String`. Рассмотрим следующий пример:

```
// Конструировать один объект 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);
}
}
```

Вывод этой программы будет выглядеть, как показано ниже:

```
Java
Java
```

Как видите, `s1` и `s2` содержат одинаковые строки.

Даже несмотря на то, что Java-тип `char` использует 16 бит для представления базового набора символов Unicode, типичный формат строк в Internet использует массивы 8-битных байт, сконструированных из набора символов ASCII. Поскольку 8-битные ASCII-

строки употребляются наиболее часто, класс `String` предлагает конструкторы, которые инициализируют строки массивом `byte`. Их форма приведена ниже:

```
String(byte asciiChars[ ])
String(byte asciiChars[ ], int startIndex, int numChars)
```

Здесь *asciiChars* представляет массив байт. Вторая форма позволяет вам указать поддиапазон. В каждом из этих конструкторов преобразование байт в символы выполняется в соответствии с кодировкой по умолчанию для конкретной платформы. Применение этих конструкторов иллюстрируется в следующей программе.

```
// Конструирование строки из подмножества символьного масива.
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 показан вывод, генерируемый этой программой.

```
ABCDEF
CDE
```

Существуют также расширенные версии конструкторов “байт-строка”, в которых вы можете указать кодировку символов, определяющую способ преобразования байтов в символы. Однако в большинстве случаев вам подойдет кодировка для данной платформы по умолчанию.

На заметку! *Содержимое массива копируется всякий раз, когда вы создаете объект `String` из массива. Если вы модифицируете содержимое массива после создания строки, ваш объект `String` останется неизменным.*

Вы можете сконструировать `String` из `StringBuffer`, используя следующий конструктор:

```
String(StringBuffer strBufObj)
```

Конструкторы строк, добавленные в J2SE 5

В J2SE 5 были добавлены два новых конструктора `String`. Первый из них поддерживает расширенный набор символов Unicode и выглядит следующим образом:

```
String(int codePoints[], int startIndex, int numChars)
```

Здесь *codePoints* — массив, содержащий символы Unicode. Результирующая строка конструируется из диапазона, начинающегося со *startIndex* и длиной *numChars*.

На заметку! *Обсуждение элементов кода Unicode и способов работы с ними в Java можно найти в главе 16.*

Второй новый конструктор поддерживает новый класс `StringBuilder`. Он выглядит так:

```
String(StringBuilder strBuildObj)
```

Это конструирует `String` из `StringBuilder`, переданного в параметре *`strBuildObj`*.

Длина строки

Длина строки — это количество символов, из которых она состоит. Чтобы получить это значение, вызывайте метод `length()`:

```
int length()
```

Следующий фрагмент печатает 3, поскольку именно три символа содержит строка `s`:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());
```

Специальные строковые операции

Поскольку строки — общая и важная часть программирования, Java добавляет специальную поддержку некоторых строковых операций в рамках синтаксиса языка. Эти операции включают автоматическое создание новых экземпляров `String` из строковых литералов, конкатенацию множества объектов `String` с помощью операции `+`, а также преобразование других типов данных в строковое представление. Существуют явные методы для реализации всех этих функций, но Java также выполняет их автоматически для удобства программистов и большей ясности.

Строковые литералы

В предшествующих примерах было показано, как явным образом создавать объекты `String` из массива символов с помощью операции `new`. Однако есть более простой способ сделать это с помощью строковых литералов. Для каждого строкового литерала в вашей Java-программе автоматически конструируется объект `String`. Таким образом, вы можете использовать строковый литерал для инициализации объекта `String`. Например, следующий фрагмент кода создает две эквивалентные строки:

```
char chars[] = { 'a', 'b', 'c' };
String s1 = new String(chars);

String s2 = "abc"; // используется строковый литерал
```

Поскольку объект `String` конструируется для каждого строкового литерала, вы можете использовать литерал в любом месте, где допускается применение объекта `String`. Например, вы можете вызывать методы непосредственно со строками в кавычках, как если бы они были ссылками на объекты, что показано в следующем примере. Здесь вызывается метод `length()` для строки “abc”. Как и ожидалось, он возвращает 3.

```
System.out.println("abc".length());
```

Конкатенация строк

В общем случае, Java не позволяет применять операции к объектам `String`. Одно исключение из правил — применение операции +, которая соединяет две строки, порождая в результате объект `String`. Это позволяет соединять вместе серии операций +. Например, в следующем фрагменте кода осуществляется конкатенация трех строк:

```
String age = "9";
String s = "Ему " + age + " лет.";
System.out.println(s);
```

В результате отображается строка “Ему 9 лет”.

Одно практическое применение конкатенации строк — когда вы создаете очень длинные строки. Вместо того чтобы включать в код длинные строки одним куском, вы можете разбить их на маленькие фрагменты, используя + для конкатенации. Вот пример:

```
// Использование конкатенации во избежание длинных строк.
class ConCat {
public static void main(String args[]) {
  String longStr = "Это может быть " +
       "очень длинная строка, которую следует " +
       "перенести. Но конкатенация позволяет " +
       "предотвратить это.";
  System.out.println(longStr);
}
}
```

Конкатенация строк с другими типами данных

Вы можете соединять строки с данными других типов. Например, рассмотрим слегка измененную версию предыдущего примера:

```
int age = 9;
String s = "Ему " + age + " лет.";
System.out.println(s);
```

В этом случае `age` имеет тип `int`, а не `String`, но выходной результат получается тот же самый, что и раньше. Так происходит потому, что значение типа `int` автоматически преобразуется в строковое представление в объекте `String`. После этого строки конкатенируются, как и прежде. Компилятор преобразует операнды в их строковые эквиваленты, в то время как другие операнды + являются экземплярами `String`.

Будьте внимательны, когда смешиваете операнды других типов со строками в выражениях конкатенации. В противном случае возможно получить весьма неожиданные результаты. Рассмотрим следующий код:

```
String s = "четыре: " + 2 + 2;
System.out.println(s);
```

Этот фрагмент отображает:

```
четыре: 22
```

вместо:

```
четыре: 4
```

чего вы, вероятно, ожидали. И вот почему. Приоритеты операций обеспечивают в начале конкатенацию “четыре” со строковым эквивалентом числа 2. Результат затем конкатени-

руется со строковым эквивалентом второго числа 2. Чтобы вначале выполнить целочисленное сложение, вы должны применить скобки:

```
String s = " четыре: " + (2 + 2);
```

Теперь s содержит строку " четыре: 4".

Преобразование строк и toString()

Когда Java преобразует данные в строковое представление при конкатенации, она делает это посредством вызова одной из перегруженных версий преобразующих методов valueOf(), определенных в String. Метод valueOf() перегружен для всех простых типов и для типа Object. Для простых типов valueOf() возвращает строку, которая содержит читабельный для человека эквивалент значения, с которым он был вызван. Для объектов valueOf() вызывает метод этого объекта toString(). Мы рассмотрим более подробно valueOf() позднее в этой главе. А сейчас давайте изучим метод toString(), потому что это средство строкового представления объекта классов, которые вы создадите.

Каждый класс реализует toString(), поскольку этот метод определен в Object. Однако реализация toString() по умолчанию редко может быть полезной. Для всех наиболее важных классов, которые вы создадите, вы пожелаете переопределить toString() и предоставить свое собственное строковое представление. К счастью, это легко сделать. Общий метод toString() имеет следующую форму:

```
String toString()
```

Чтобы реализовать его, просто возвратите объект String, который содержит читабельную для человека строку, адекватно описывающую объект вашего класса.

Переопределяя toString() для создаваемых вами классов, вы позволяете им полностью интегрироваться в программное окружение Java. Например, они могут применяться в операторах print() и println(), а также в строковых выражениях с конкатенацией. Следующая программа демонстрирует это, переопределяя toString() для класса Box.

```
// Переопределение toString() для класса 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 = "Box b: " + b;    // конкатенация объекта Box
  System.out.println(b);       // преобразование Box в строку
  System.out.println(s);
}
}
```

Вывод этой программы выглядит так:

```
Размеры 10.0 на 14.0 на 12.0
Box b: Размеры 10.0 на 14.0 на 12.0
```

Извлечение символов

Класс `String` предлагает множество способов извлечения символов из объекта `String`. Каждый из них мы рассмотрим. Хотя символы, которые составляют строку, не могут быть индексированы подобно тому, как это делается в символьных массивах, однако многие методы класса `String` используют индексы (или смещения) в строке для выполнения своих операций. Подобно массивам, строки индексируются, начиная с нуля.

charAt()

Чтобы выделить единственный символ из `String`, вы можете сослаться непосредственно к индивидуальному символу с помощью метода `charAt()`. Он имеет следующую общую форму:

```
char charAt(int where)
```

Здесь *where* — индекс символа, который нужно получить. Значение *where* должно быть не отрицательным и указывать положение в строке. `charAt()` возвращает символ, находящийся в указанном положении. Например,

```
char ch;
ch = "abc".charAt(1);
```

присваивает значение 'b' переменной `ch`.

getChars()

Если вам нужно извлечь более одного символа сразу, вы можете применить метод `getChars()`. Он имеет следующую общую форму:

```
void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)
```

Здесь *sourceStart* указывает индекс начала подстроки, а *sourceEnd* — индекс символа, следующего за концом требуемой подстроки. Таким образом, извлекается подстрока, содержащая символы от *sourceStart* до *sourceEnd* - 1. Массив, который принимает выделенные символы, указан в параметре *target*. Индекс в массиве *target*, начиная с которого будет записываться подстрока, передается в *targetStart*. Следует позаботиться о том, чтобы массив *target* был достаточного размера, чтобы в нем поместились все символы указанной подстроки.

В следующей программе демонстрируется использование `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);
}
}
```

Вывод программы показан ниже:

```
демо
```

getBytes()

Существует альтернатива `getChars()`, которая сохраняет символы в массив байт. Этот метод называется `getBytes()` и использует преобразование символов в байты по умолчанию, предоставляемое платформой. Вот его простейшая форма:

```
byte[] getBytes()
```

Доступны также другие формы этого метода. `getBytes()` в основном применим, когда вы экспортируете значения `String` в среды, которые не поддерживают 16-битные символы Unicode. Например, большинство протоколов Internet и форматов текстовых файлов используют 8-битный код ASCII для всех текстовых взаимодействий.

toCharArray()

Если вы хотите преобразовать все символы в объекте `String` в символьный массив, то простейший способ сделать это — вызвать метод `toCharArray()`. Он возвращает массив символов для всей строки. Его общая форма такова:

```
char toCharArray()
```

Эта функция представлена в качестве дополнения, поскольку тот же результат можно получить, применив `getChars()`.

Сравнение строк

Класс `String` включает некоторые методы, предназначенные для сравнения строк или подстрок в строках. Рассмотрим их все.

equals() и equalsIgnoreCase()

Чтобы сравнить две строки на эквивалентность, используйте `equals()`. Он имеет следующую общую форму:

```
boolean equals(Object str)
```

Здесь *str* — это объект `String`, который сравнивается с вызывающим объектом `String`. Метод возвращает `true`, если строка содержит те же символы и в том же порядке, и `false` — в противном случае. Сравнение зависит от регистра.

Чтобы выполнить сравнение, игнорирующее регистр символов, вызывайте `equalsIgnoreCase()`. Когда этот метод сравнивает две строки, он рассматривает диапазон `A-Z` как то же самое, что и `a-z`. Он имеет следующую общую форму:

```
boolean equalsIgnoreCase(Object str)
```

Здесь *str* — это объект `String`, который сравнивается с вызывающим объектом `String`. Метод также возвращает `true`, если строки содержат одинаковые символы в том же порядке, и `false` в противном случае.

Вот пример, демонстрирующий применение `equals()` и `equalsIgnoreCase()`:

```
// Демонстрация применения equals() и equalsIgnoreCase().
class equalsDemo {
public static void main(String args[]) {
  String s1 = "Привет";
  String s2 = "Привет";
  String s3 = "Пока";
  String s4 = "ПРИВЕТ";
  System.out.println(s1 + " эквивалентно " + s2 + " -> " +
  s1.equals(s2));
  System.out.println(s1 + " эквивалентно " + s3 + " -> " +
  s1.equals(s3));
  System.out.println(s1 + " эквивалентно " + s4 + " -> " +
  s1.equals(s4));
  System.out.println(s1 + " эквивалентно, игнорируя регистр " + s4 + " -> " +
  s1.equalsIgnoreCase(s4));
}
}
```

Вывод программы показан ниже:

```
Привет эквивалентно Привет -> true
Привет эквивалентно Good-bye -> false
Привет эквивалентно ПРИВЕТ -> false
Привет эквивалентно, игнорируя регистр ПРИВЕТ -> true
```

regionMatches()

Метод `regionMatches()` сравнивает указанную часть строки с другой частью строки. Существует также перегруженная форма, которая игнорирует регистр символов при сравнении. Вот общая форма этих двух методов:

```
boolean regionMatches(int startIndex, String str2,
  int str2StartIndex, int numChars)
boolean regionMatches(boolean ignoreCase, int startIndex, String str2,
  int str2StartIndex, int numChars)
```

В обеих версиях *startIndex* задает индекс начала диапазона строки вызывающего объекта `String`. Строка, подлежащая сравнению, передается в *str2*. Индекс символа, начиная с которого нужно выполнять сравнение в *str2*, передается в *str2StartIndex*, а длина сравниваемой подстроки — в *numChars*. Во второй версии, если *ignoreCase* равно `true`, регистр символов игнорируется. В противном случае регистр учитывается.

startsWith() и endsWith()

В `String` определены два метода, представляющие собой более или менее специализированные формы `regionMatches()`. Метод `startWith()` определяет, начинается ли заданный объект `String` с указанной строки. В дополнение `endsWith()` определяет, завершается ли объект `String` заданным фрагментом. Эти методы имеют следующую общую форму:

```
boolean startsWith(String str)
boolean endsWith(String str)
```

Здесь *str* — фрагмент строки, наличие которого соответственно в начале или конце данного объекта `String` нужно проверить. Если он присутствует, возвращается `true`, иначе `false`.

Например:

```
"Foobar".endsWith("bar")
```

и

```
"Foobar".startsWith("Foo")
```

возвращают `true`.

Сравнение `equals()` и операции ==

Важно понимать разницу между методом `equals()` и операцией ==. Это два разных действия. Как было объяснено, метод `equals()` сравнивает символы внутри объекта `String`. Операция == сравнивает две ссылки на объекты и определяет, ссылаются ли они на один и тот же экземпляр. В следующей программе показано, как два разных объекта `String` могут содержать одинаковые символы, но ссылки на эти объекты при сравнении будут не эквивалентными.

```
// equals() против ==
class EqualsNotEqualTo {
public static void main(String args[]) {
  String s1 = "Привет";
  String s2 = new String(s1);

  System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));

  System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
}
}
```

Переменная `s1` ссылается на экземпляр `String`, созданный присвоением литерала “Привет”. Объект, на который ссылается `s2`, создается с использованием `s1` в качестве инициализатора. Таким образом, содержимое обоих объектов `String` идентично, но это отличные друг от друга объекты. Это означает, что `s1` и `s2` не ссылаются на один и тот же объект, а потому не равны (при сравнении операцией ==), как доказывает вывод предыдущей программы:

```
Привет equals Привет -> true
Привет == Привет -> false
```

`compareTo()`

Часто недостаточно знать, что строки просто идентичны. Для приложений, выполняющих сортировку, нужно знать, какая из строк *меньше*, *равна* или *больше* следующей. Строка меньше другой, если она расположена перед ней в лексикографическом порядке. Строка больше другой, если расположена после нее. Метод `String` по имени `compareTo()` служит этой цели. Он имеет следующую общую форму:

```
int compareTo(String str)
```

Здесь *`str`* — объект `String`, сравниваемый с вызывающим объектом `String`. Возвращаемый результат интерпретируется так, как показано в твбл. 15.1.

Таблица 15.1. Возвращаемый результат метода `compareTo()`

Значение	Описание
Меньше нуля	Вызывающая строка меньше `str`.
Больше нуля	Вызывающая строка больше `str`.
Ноль	Две строки эквивалентны.

Ниже представлен пример программы, которая сортирует массив строк. Программа использует `compareTo()` для определения порядка сортировки в алгоритме пузырьковой сортировки.

```
// Пузырьковая сортировка объектов 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]);
  }
}
}
```

Выводом этой программы является список слов:

```
Now
aid
all
come
country
for
good
is
men
of
the
the
their
time
to
to
```

Как вы можете видеть из вывода этого примера, `compareTo()` принимает во внимание заглавные и прописные буквы. Слово “Now” идет прежде всех остальных, поскольку начинается с заглавной буквы, что означает, что она имеет меньшее значение в наборе символов ASCII.

Если вы хотите игнорировать регистр символов при сравнении строк, используйте `compareToIgnoreCase()`:

```
int compareToIgnoreCase(String str)
```

Этот метод возвращает тот же результат, что и `compareTo()`, за исключением того, что регистр символов игнорируется. Вы можете попытаться подставить этот метод в предыдущую программу. После этого "New" уже не будет первым в списке.

Поиск строк

Класс `String` предлагает два метода, которые позволяют вам выполнять поиск в строке определенного символа или подстроки.

- `indexOf()` — ищет первое вхождение символа или подстроки.
- `lastIndexOf()` — ищет последнее вхождение символа или подстроки.

Эти два метода перегружены несколькими разными способами. Во всех случаях эти методы возвращают позицию в строке (индекс), где символ или подстрока была найдена, либо `-1` в случае неудачи.

Чтобы найти первое вхождение символа, применяйте:

```
int indexOf(char ch)
```

Чтобы найти последнее вхождение символа:

```
int lastIndexOf(char ch)
```

Здесь *ch* — символ, который нужно искать.

Чтобы найти первое или последнее вхождение подстроки, применяйте:

```
int indexOf(String str)
int lastIndexOf(String str)
```

Здесь *str* задает искомую подстроку.

Вы можете указать начальную позицию для поиска, воспользовавшись следующими формами:

```
int indexOf(int ch, int startIndex)
int lastIndexOf(int ch, int startIndex)
int indexOf(String str, int startIndex)
int lastIndexOf(String str, int startIndex)
```

Здесь *startIndex* задает начальную позицию поиска. Для `indexOf()` поиск начинается от *startIndex* до конца строки, а для `lastIndexOf()` — от *startIndex* до нуля.

Следующий пример показывает, как использовать различные индексные методы для поиска внутри `String`:

```
// Демонстрация использования indexOf() и lastIndexOf().
class indexOfDemo {
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 показан вывод этой программы:

```
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
```

Модификация строк

Поскольку объекты `String` неизменяемы, всякий раз, когда вы хотите их модифицировать, вы либо должны скопировать их содержимое в `StringBuffer` или `StringBuilder`, либо воспользоваться одним из следующих методов `String`, которые конструируют новые копии строк с проведенными в них модификациями.

substring()

Вы можете извлечь подстроку, используя `substring()`. Этот метод имеет две формы. Первая:

```
String substring(int startIndex)
```

Здесь *startIndex* указывает индекс, с которого начнется подстрока. Эта форма возвращает копию подстроки, которая начинается с позиции *startIndex* и продолжается до завершения вызывающей строки.

Вторая форма `substring()` позволяет указать как начальный, так и конечный индексы подстроки:

```
String substring(int startIndex, int endIndex)
```

Здесь *startIndex* указывает индекс, с которого начнется подстрока, а *endIndex* — точку конца подстроки. Возвращаемая строка содержит все символы, начиная от первой позиции и до последней, исключая ее.

В следующей программе substring() используется для замены в строке всех экземпляров одной подстроки другой.

```
// 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 { // замена всех совпадающих подстрок
     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);
}
}
```

Вывод этой программы показан ниже:

```
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.
```

concat()

Вы можете соединить две подстроки, используя concat(), как показано ниже:

```
String concat(String str)
```

Этот метод создает новый объект, который содержит вызываемую строку с содержимым *str*, присоединенным к ее концу. concat() выполняет ту же функцию, что и операция +. Например, следующий код помещает строку “onetwo” в s2.

```
String s1 = "one";
String s2 = s1.concat("two");
```

Код генерирует тот же результат, что и такая последовательность:

```
String s1 = "one";
String s2 = s1 + "two";
```

replace()

Метод replace() имеет две формы. Первая заменяет в исходной строке все вхождения одного символа другим. Вот эта форма:

```
String replace(char original, char replacement)
```

Здесь *original* задает символ, который должен быть заменен *replacement*. Возвращается результирующая строка. Например:

```
String s = "Hello".replace('l', 'w');
```

помещает в s строку “Hewwo”.

Вторая форма replace() заменяет одну последовательность символов на другую. Она выглядит так:

```
String replace(CharSequence original, CharSequence replacement)
```

Эта форма появилась в J2SE 5.

trim()

Метод trim() возвращает копию вызывающей строки, из которой удалены все ведущие и завершающие пробелы. Он имеет следующую общую форму:

```
String trim()
```

Вот пример:

```
String s = "   Hello World    ".trim();
```

В результате в s будет помещена строка “Hello World”.

Метод trim() достаточно удобен при обработке других команд. Например, следующая программа приглашает пользователя ввести название штата, а затем отображает название города — столицы штата. Она использует trim() для удаления всех ведущих и хвостовых пробелов, которые могут быть непреднамеренно введены пользователем.

```
// Использование trim() для обработки команд.
import java.io.*;
class UseTrim {
public static void main(String args[])
  throws IOException
{
  // Создается BufferedReader с использованием System.in
  BufferedReader br = new
  BufferedReader(new InputStreamReader(System.in));
  String str;
  System.out.println("Введите 'стоп' для завершения.");
  System.out.println("Введите штат: ");
  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 с помощью valueOf()

Метод valueOf() преобразует данные из внутреннего представления в читабельную для пользователя форму. Это статический метод, который перегружен в String для всех встроенных в Java типов таким образом, что каждый тип может быть правильно преобразован в строку. valueOf() также перегружен для типа Object, поэтому объект типа любого класса, который вы создадите, также может использоваться в качестве аргумента. (Вспомните, что Object — суперкласс для всех классов.) Ниже показаны некоторые из его форм:

```
static String valueOf(double num)
static String valueOf(long num)
static String valueOf(Object ob)
static String valueOf(char chars[])
```

Как упоминалось ранее, valueOf() вызывается, когда требуется строковое представление некоторого другого типа данных — например, при операции конкатенации. Вы можете вызывать этот метод непосредственно с любым типом данных и получать адекватное представление типа String. Все простые типы преобразуются в их общее String-представление. Любой объект, который вы можете передать valueOf(), возвратит результат методу объекта toString(). Фактически вы можете просто вызвать toString() и получить тот же результат.

Для большинства массивов valueOf() возвращает зашифрованную строку, означающую, что это массив определенного типа. Для массивов char, однако, создается объект String, содержащий все символы массива char. Он имеет следующую форму:

```
static String valueOf(char chars[], int startIndex, int numChars)
```

Здесь *chars* — это массив, который содержит символы, *startIndex* — начальная позиция в массиве, с которой начинается подстрока, а *numChars* указывает длину подстроки.

Изменение регистра символов в строке

Метод toLowerCase() преобразует все символы строки из верхнего регистра в нижний. Метод toUpperCase() преобразует все символы строки из нижнего регистра в верхний. Небуквенные символы, такие как десятичные цифры, остаются неизменными. Вот общая форма этих методов:

```
String toLowerCase( )
String toUpperCase( )
```

Оба метода возвращают объект String, содержащий эквивалент вызывающей строки соответственно в нижнем или верхнем регистре.

Ниже показан пример, в котором используются toLowerCase() и toUpperCase().

```
// Демонстрация 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);
  }
  }
```

Вывод приведенной выше программы:

```
Исходная строка: Это тест
Верхний регистр: ЭТО ТЕСТ
Нижний регистр: это тест
```

Дополнительные методы `String`

В дополнение к методам, перечисленным выше, `String` включает также и ряд других. Они перечислены в табл. 15.2. Следует иметь в виду, что многие методы появились только в J2SE 5.

Таблица 15.2. Дополнительные методы класса `String`

Метод	Описание
`int codePointAt(int i)`	Возвращает точку кода Unicode символа, находящегося в позиции `i`. Добавлен в J2SE 5.
`int codePointBefore(int i)`	Возвращает точку кода Unicode символа, находящегося в позиции, предшествующей `i`. Добавлен в J2SE 5.
`int codePointCount(int start, int end)`	Возвращает количество точек кода в части вызывающей строки между *start* и *end*-1. Добавлен в J2SE 5.
`boolean contains(CharSequence str)`	Возвращает `true`, если вызывающий объект содержит строку, указанную в *str*. В противном случае возвращает `false`. Добавлен в J2SE 5.
`boolean contentEquals(CharSequence str)`	Возвращает `true`, если вызывающий объект содержит ту же строку, что и *str*. В противном случае возвращает `false`. Добавлен в J2SE 5.
`boolean contentEquals(StringBuffer str)`	Возвращает `true`, если вызывающий объект содержит ту же строку, что и *str*. В противном случае возвращает `false`. Добавлен в J2SE 5.
`static String format(String fmtstr, Object ... args)`	Возвращает строку, форматированную в соответствии с *fmtstr*. (Подробности о форматировании описаны в главе 18.) Добавлен в J2SE 5.
`static String format(Locale loc, String fmtstr, Object ... args)`	Возвращает строку, форматированную в соответствии с *fmtstr*. (Подробности о форматировании описаны в главе 18.) Добавлен в J2SE 5.
`boolean matches(string regExp)`	Возвращает `true`, если вызывающая строка соответствует регулярному выражению, переданному в *regExp*. В противном случае возвращает `false`.
`int offsetByCodePoints(int start, int num)`	Возвращает индекс в вызывающей строке, который находится на *num* точек кода за начальным индексом, указанным в *start*. Добавлен в J2SE 5.

Окончание табл. 15.2

Метод	Описание
`String replaceFirst(String regExp, String newStr)`	Возвращает строку, в которой первая подстрока, соответствующая регулярному выражению *regExp*, заменяется *newStr*.
`String replaceAll(String regExp, String newStr)`	Возвращает строку, в которой все подстроки, соответствующие регулярному выражению *regExp*, заменяется *newStr*.
`String[] split(String regExp)`	Разбирает вызывающую строку на части и возвращает массив, содержащий результат. Каждая часть ограничена регулярным выражением *regExp*.
`String[] split(String regExp, int max)`	Разбирает вызывающую строку на части и возвращает массив, содержащий результат. Каждая часть ограничена регулярным выражением *regExp*. Количество частей указано в *max*. Если *max* отрицательное, значит, вызывающая строка разбирается полностью. В противном случае, если *max* содержит неотрицательное значение, то последний элемент возвращаемого массива содержит остаток вызывающей строки. Если *max* равно нулю, строка также разбирается полностью.
`CharSequence subSequence(int startIndex, int stopIndex)`	Возвращает подстроку вызывающей строки, начиная с *startIndex* и заканчивая *stopIndex*. Этот метод требует интерфейса `CharSequence`, который теперь реализует класс `String`.

Обратите внимание, что некоторые из этих методов работают с регулярными выражениями. Регулярные выражения описаны в главе 27.

StringBuffer

`StringBuffer` — это класс, подобный `String`, который представляет большую часть функциональности строк. Как вы знаете, `String` представляет неизменяемые последовательности символов постоянной длины. В отличие от этого, `StringBuffer` представляет расширяемые и доступные для изменений последовательности символов. `StringBuffer` позволяет вставлять символы и подстроки в середину либо добавлять их в конец. `StringBuffer` автоматически растет, чтобы обеспечить место для подобных расширений, и часто, чтобы обеспечить возможность возрастания, имеет больше предварительно выделенных символов, чем фактически нужно в данный момент. Java интенсивно использует оба класса, но многие программисты имеют дело только с классом `String`, позволяя Java манипулировать объектами `StringBuffer` "за кулисами" за счет использования перегруженной операции +.

Конструкторы StringBuffer

В `StringBuffer` определены следующие четыре конструктора:

```
StringBuffer()
StringBuffer(int size)
StringBuffer(String str)
StringBuffer(CharSequence chars)
```

Конструктор по умолчанию (не имеющий параметров) резервирует место под 16 символов без перераспределения памяти. Вторая версия принимает целый аргумент, который явно устанавливает размер буфера. Третья версия принимает аргумент типа `String`, который устанавливает начальное содержимое объекта `StringBuffer` и резервирует 16 символов без повторного распределения. `StringBuffer` выделяет место под 16 дополнительных символов, когда не указывается конкретный размер буфера, поскольку распределение памяти — дорогостоящая операция в смысле временных затрат. Кроме того, повторное распределение может фрагментировать память. Выделяя место под несколько дополнительных символов, `StringBuffer` снижает количество необходимых повторных распределений. Четвертый конструктор создает объект, содержащий последовательность символов из параметра *chars*.

length() и capacity()

Текущую длину `StringBuffer` можно получить методом `length()`, а текущий объем выделенной памяти — методом `capacity()`. Они имеют следующую общую форму:

```
int length()
int capacity()
```

Ниже показан пример:

```
// Сравнение методов length() и capacity() класса StringBuffer.
class StringBufferDemo {
public static void main(String args[]) {
  StringBuffer sb = new StringBuffer("Hello");

  System.out.println("buffer = " + sb);
  System.out.println("length = " + sb.length());
  System.out.println("capacity = " + sb.capacity());
}
}
```

Ниже представлен вывод этой программы, который показывает, как `StringBuffer` резервирует запасное пространство для дополнительных манипуляций:

```
buffer = Hello
length = 5
capacity = 21
```

Поскольку `sb` инициировано строкой “Hello” при создании, его длина равна 5. Объем выделенной памяти (capacity) равен 21, так как 16 дополнительных символов добавлены автоматически.

ensureCapacity()

Если вы хотите предварительно выделить место для определенного количества символов после того, как `StringBuffer` сконструирован, вы можете воспользоваться `ensureCapacity()`, чтобы установить размер буфера. Это удобно, если вы знаете наперед, что собираетесь добавлять большое количество маленьких строк к `StringBuffer`. `ensureCapacity()` имеет следующую общую форму:

```
void ensureCapacity(int capacity)
```

Здесь *capacity* указывает размер буфера.

setLength()

Чтобы установить длину буфера внутри объекта StringBuffer, используйте setLength(). Общая форма этого метода выглядит следующим образом:

```
void setLength(int len)
```

Здесь *len* указывает длину буфера. Значение должно быть неотрицательным.

Когда вы увеличиваете размер буфера, в конец существующего буфера добавляются нулевые символы. Если вы вызываете setLength() со значением, меньшим чем текущее значение, возвращаемое length(), то символы, находящиеся за пределами вновь установленной длины, будут утеряны. Пример программы setCharAtDemo() в следующем разделе использует setLength() для сокращения StringBuffer.

charAt() и setCharAt()

Значение отдельного символа может быть извлечено из StringBuffer методом charAt(). Вы можете установить значение символа внутри StringBuffer с помощью setCharAt(). Общая форма этих методов показана ниже:

```
char charAt(int where)
void setCharAt(int where, char ch)
```

Для метода charAt() параметр *where* указывает индекс символа, который нужно извлечь. Для setCharAt() параметр *where* указывает индекс символа, который нужно установить, а *ch* — его значение. Для обоих методов *where* должен быть неотрицательным и не должен находиться за пределами конца буфера.

В следующем примере демонстрируется применение charAt() и setCharAt().

```
// Демонстрация 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 charAt(1) = " + sb.charAt(1));
  sb.setCharAt(1, 'i');
  sb.setLength(2);
  System.out.println("буфер после = " + sb);
  System.out.println("после charAt(1) = " + sb.charAt(1));
}
}
```

Вывод, сгенерированный этой программой выглядит так:

```
буфер до = Hello
до charAt(1) = e
буфер после = Hi
после charAt(1) = i
```

getChars()

Чтобы скопировать подстроку из StringBuffer в массив, используйте метод getChars(). Он имеет следующую форму:

```
void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)
```

Здесь *sourceStart* указывает индекс начала подстроки, а *sourceEnd* — индекс символа, следующего за концом требуемой подстроки. Это означает, что подстрока содержит символы от *sourceStart* до *sourceEnd*-1. Массив, который принимает символы, передается через *target*. Индекс внутри *target*, куда копируется подстрока, передается в параметре *targetStart*. Необходимо позаботиться о том, чтобы массив *target* был достаточного размера, чтобы вместить количество символов указанной подстроки.

append()

Метод append() соединяет строковое представление любого другого типа данных с концом вызывающего объекта StringBuffer. Он имеет несколько перегруженных версий. Вот несколько из его форм:

```
StringBuffer append(String str)
StringBuffer append(int num)
StringBuffer append(Object obj)
```

String.valueOf() вызывается для каждого параметра, чтобы получить его строковое представление. Результат добавляется к текущему объекту StringBuffer. Сам буфер возвращается каждой из версий append(). Это позволяет соединять несколько последовательных вызовов вместе, как показано в следующем примере:

```
// Демонстрация применения 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("!").toString();
  System.out.println(s);
}
}
```

Вывод этого примера показан ниже:

```
a = 42;
```

Метод append() чаще всего вызывается, когда используется операция + над объектами String. Java автоматически заменяет модификации экземпляров String на соответствующие операции экземпляра StringBuffer. Таким образом, конкатенация приводит к вызову метода append() объекта StringBuffer. После того, как выполнится конкатенация, компилятор вставляет вызов toString(), чтобы обратить изменяемый StringBuffer обратно в String. Все это может показаться неоправданно сложным. Почему бы не ограничиться каким-то одним строковым классом, ведущим себя подобно StringBuffer? Ответ — в целях производительности. Существует множество оптимизаций, которые исполняющая система Java может выполнить, предполагая, что объект String неизменен. К счастью, Java скрывает большую часть сложности при преобразовании между String и StringBuffer. Фактически большинство программистов вообще никогда не испытывают потребности в непосредственной работе с StringBuffer и могут выразить большую часть операций в терминах операции + над объектами String.

insert()

Метод `insert()` вставляет одну строку в другую. Он перегружен, чтобы принимать в параметре значения всех простых типов плюс объекты `String`, `Object` и `CharSequence`. Подобно `append()`, он обращается к `String.valueOf()` для получения строкового представления значения, с которым вызван. Эта строка затем вставляется в вызывающий объект `StringBuffer`. Существуют несколько форм этого метода:

```
StringBuffer insert(int index, String str)
StringBuffer insert(int index, char ch)
StringBuffer insert(int index, Object obj)
```

Здесь *index* указывает индекс позиции вызывающего объекта `StringBuffer`, в которой будет вставлена строка.

Следующий пример программы демонстрирует вставку “like” между “I” и “Java”:

```
// Демонстрация применения insert().
class insertDemo {
public static void main(String args[]) {
  StringBuffer sb = new StringBuffer("I Java!");

  sb.insert(2, "like ");
  System.out.println(sb);
}
}
```

Вывод этой программы выглядит следующим образом:

```
I like Java!
```

reverse()

Изменить порядок символов в объекте `StringBuffer` на обратный можно с помощью `reverse()`:

```
StringBuffer reverse()
```

Этот метод возвращает объект с обратной последовательностью символов по отношению к тому, который его вызвал. В следующей программе демонстрируется использование `reverse()`.

```
// Применения reverse() для обращения порядка StringBuffer.
class ReverseDemo {
public static void main(String args[]) {
  StringBuffer s = new StringBuffer("abcdef");

  System.out.println(s);
  s.reverse();
  System.out.println(s);
}
}
```

Вывод этой программы показан ниже:

```
abcdef
fedcba
```

delete() и deleteCharAt()

Вы можете удалять символы из StringBuffer с помощью методов delete() и deleteCharAt(). Эти методы показаны ниже:

```
StringBuffer delete(int startIndex, int endIndex)
StringBuffer deleteCharAt(int loc)
```

Метод delete() удаляет последовательность символов из вызывающего объекта. Здесь *startIndex* задает индекс первого символа, который надо удалить, а *endIndex* — индекс символа, следующего за последним из удаляемых. Таким образом, удаляемая подстрока начинается с *startIndex* и заканчивается *endIndex*-1. Возвращается результирующий объект StringBuffer.

Метод deleteCharAt() удаляет символ, находящийся в позиции *loc*. Возвращает результирующий объект StringBuffer.

Вот программа, которая демонстрирует методы delete() и deleteCharAt():

```
// Демонстрация применения delete() и deleteCharAt()
class deleteDemo {
public static void main(String args[]) {
  StringBuffer sb = new StringBuffer("This is a test.");
  sb.delete(4, 7);
  System.out.println("После delete: " + sb);
  sb.deleteCharAt(0);
  System.out.println("После deleteCharAt: " + sb);
}
}
```

Программа генерирует вывод:

```
После delete: This a test.
После deleteCharAt: his a test.
```

replace()

Вы можете заменить один набор символов другим внутри StringBuffer вызовом replace(). Сигнатура этого метода показана ниже:

```
StringBuffer replace(int startIndex, int endIndex, String str)
```

Подстрока, которую нужно заменить, задается индексами *startIndex* и *endIndex*. Таким образом, заменяется подстрока от символа *startIndex* до *endIndex*-1. Строка замены передается в *str*. Возвращается результирующий объект StringBuffer.

В следующей программе демонстрируется использование replace().

```
// Демонстрация применения replace()
class replaceDemo {
public static void main(String args[]) {
  StringBuffer sb = new StringBuffer("This is a test.");
  sb.replace(5, 7, "was");
  System.out.println("После замены: " + sb);
}
}
```

Ниже показан вывод программы:

```
После замены: This was a test.
```

substring()

Вы можете получить часть StringBuffer вызовом substring(). Этот метод имеет две следующие формы:

```
String substring(int startIndex)
String substring(int startIndex, int endIndex)
```

Первая форма возвращает подстроку, которая начинается от *startIndex* и продолжается до конца вызывающего объекта StringBuffer. Вторая форма возвращает подстроку от позиции *startIndex* до *endIndex*-1. Эти методы работают точно так же, как их описанные выше аналоги в классе String.

Дополнительные методы StringBuffer

В дополнение к описанным методам StringBuffer включает и ряд других. Они перечислены в табл. 15.3. Следует иметь в виду, что некоторые методы появились только в J2SE 5.

Таблица 15.3. Дополнительные методы класса StringBuffer

Метод	Описание
StringBuffer appendCodePoint(int *ch*)	Добавляет точку кода Unicode в конец вызывающего объекта. Возвращается ссылка на объект. Добавлен в J2SE 5.
int codePointAt(int *i*)	Возвращает точку кода Unicode в позиции, указанной в параметре *i*. Добавлен в J2SE 5.
int codePointBefore(int *i*)	Возвращает точку кода Unicode в позиции, предшествующей *i*. Добавлен в J2SE 5.
int codePointCount(int *start*, int *end*)	Возвращает число точек кода в части вызывающей строки, заключенной между *start* и *end*-1. Добавлен в J2SE 5.
int indexOf(String *str*)	Выполняет поиск в вызывающем StringBuffer первого вхождения *str*. Возвращает индекс позиции совпадения или -1 в случае неудачи.
int indexOf(String *str*, int *startIndex*)	Выполняет поиск в вызывающем StringBuffer первого вхождения *str*, начиная с *startIndex*. Возвращает индекс позиции совпадения или -1 в случае неудачи.
int lastIndexOf(String *str*)	Выполняет поиск в вызывающем StringBuffer последнего вхождения *str*. Возвращает индекс позиции совпадения или -1 в случае неудачи.
int lastIndexOf(String *str*, int *startIndex*)	Выполняет поиск в вызывающем StringBuffer последнего вхождения *str*, начиная с *startIndex*. Возвращает индекс позиции совпадения или -1 в случае неудачи.
int offsetByCodePoints(int *start*, int *num*)	Возвращает индекс символа в вызывающей строке, который находится на *num* точек кода позади начального индекса, указанного в *start*. Добавлен в J2SE 5.
CharSequence subSequence(int *startIndex*, int *stopIndex*)	Возвращает подстроку вызывающей строки, начиная со *startIndex* и заканчивая *stopIndex*. Этот метод требует интерфейса CharSequence, который реализует StringBuffer.
void trimToSize()	Уменьшает размер символьного буфера вызывающего объекта с тем, чтобы он соответствовал текущему содержимому. Добавлен в J2SE 5.

Кроме метода sunSequence(), который реализует метод интерфейса CharSequence, другие методы из табл. 15.3 позволяют StringBuffer выполнять поиск вхождения String. В следующей программе демонстрируется применение 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);
}
}
```

Вывод будет таким:

```
Индекс первого вхождения: 0
Индекс последнего вхождения: 8
```

StringBuilder

В J2SE 5 в дополнение к существующим богатым возможностям обработки строк Java появился новый строковый класс. Этот новый класс называется StringBuilder. Он идентичен StringBuilder за исключением одного важного отличия: он не синхронизирован, что означает, что он не является безопасным в отношении потоков. Выгода от применения StringBuilder связана с более высокой производительностью. Однако в случае разработки многопоточных программ вы должны использовать StringBuffer, а не StringBuilder.

16 ГЛАВА

Пакет java.lang

Настоящая глава посвящена классам и интерфейсам, определенным в пакете java.lang. Как вы знаете, java.lang автоматически импортируется во все программы. Он содержит классы и интерфейсы, которые являются фундаментальными для всех программ на Java. Это наиболее широко используемый пакет Java.

java.lang включает следующие классы:

Boolean	InheritableThreadLocal	Runtime	System
Byte	Integer	RuntimePermission	Thread
Character	Long	SecurityManager	ThreadGroup
Class	Math	Short	ThreadLocal
ClassLoader	Number	StackTraceElement	Throwable
Compiler	Object	StrictMath	Void
Double	Package	String	
Enum	Process	StringBuffer	
Float	ProcessBuilder	StringBuilder	

Также определены два вложенных класса Character: Character.SubSet и Character.UnicodeBlock.

В java.lang определены следующие интерфейсы:

Appendable	Comparable	Runnable
CharSequence	Iterable	
Cloneable	Readable	

Некоторые из классов, включенных в пакет java.lang, содержат устаревшие (deprecated) методы, большинство из которых относятся еще к Java 1.0. Эти устаревшие методы все еще предоставляются Java для поддержки постепенно отмирающего унаследованного кода и не рекомендуются для применения в новом коде. Большинство из устаревших элементов существовали до версии Java SE 6, и эти устаревшие методы здесь не обсуждаются.

Оболочки примитивных типов

Как упоминалось в первой части настоящей книги, Java использует примитивные типы, такие как `int` и `char`, из соображений производительности. Эти типы данных не являются частью объектной иерархии. Они передаются по значению в методы и не могут быть переданы по ссылке. Также нет способа для двух методов ссылаться на *один и тот же экземпляр* `int`. Рано или поздно у вас возникнет необходимость в объектном представлении одного из примитивных типов. Например, существуют классы коллекций, описанные в главе 17, которые имеют дело только с объектами. Чтобы сохранить примитивный тип в одном из этих классов, необходимо поместить примитивный тип в оболочку класса. Чтобы удовлетворить эту потребность, Java предлагает классы, которые соответствуют каждому из примитивных типов. По сути, эти классы инкапсулируют примитивные типы в классы (или, что то же самое, помещают примитивные типы в *оболочки* классов). Таким образом, их обычно называют оболочками типов. Оболочки типов были впервые представлены в главе 12. Здесь мы рассмотрим их более подробно.

Number

Абстрактный класс `Number` определяет суперкласс, который реализован классами, являющимися оболочками для числовых типов `byte`, `short`, `int`, `long`, `float` и `double`. `Number` имеет абстрактные методы, которые возвращают значение объекта в каждом из разных числовых форматов. Например, `doubleValue()` возвращает значение как `double`, `floatValue()` — как `float`, и так далее. Эти методы перечислены ниже.

```
byte byteValue()
double doubleValue()
float floatValue()
int intValue()
long longValue()
short shortValue()
```

Значения, возвращаемые этими методами, могут быть округлены.

`Number` имеет шесть конкретных подклассов, которые содержат явные значения каждого из числовых типов: `Double`, `Float`, `Byte`, `Short`, `Integer` и `Long`.

Double и Float

`Double` и `Float` — это оболочки для значений с плавающей точкой типов `double` и `float` соответственно. Конструкторы для `Float` показаны ниже:

```
Float(double num)
Float(float num)
Float(String str) throws NumberFormatException
```

Как вы можете видеть, объекты `Float` должны быть сконструированы со значениями типа `float` или `double`. Они могут также быть сконструированы из строкового представления числа с плавающей точкой.

Вот как выглядят конструкторы для `Double`:

```
Double(double num)
Double(String str) throws NumberFormatException
```

Объекты `Double` могут быть сконструированы из значения `double` или строки, содержащей значение с плавающей точкой.

Методы, определенные в классе Float, описаны в табл. 16.1. Методы, определенные в Double, перечислены в табл. 16.2. И Float, и Double определяют следующие константы:

MAX_EXPONENT	Максимальная экспонента (добавлена в Java SE 6).
MAX_VALUE	Максимальное положительное значение.
MIN_EXPONENT	Минимальная экспонента (добавлена в Java SE 6).
MIN_NORMAL	Минимальное положительное нормальное значение (добавлена в Java SE 6).
MIN_VALUE	Минимальное положительно значение.
NaN	Не число.
POSITIVE_INFINITY	Положительная бесконечность.
NEGATIVE_INFINITY	Отрицательная бесконечность.
SIZE	Размер помещенного в оболочку значения в битах.
TYPE	Объект Class для float и double.

Таблица 16.1. Методы класса Float

Метод	Описание
byte byteValue()	Возвращает значение вызывающего объекта как byte.
static int compare(float *num1*, float *num2*)	Сравнивает значения *num1* и *num2*. Возвращает 0, если значения эквивалентны. Возвращает отрицательное значение, если *num1* меньше *num2*, и положительное — если *num1* больше *num2*.
int compareTo(Float *f*)	Сравнивает числовое значение вызывающего объекта со значением *f*. Возвращает 0, если значения эквивалентны. Возвращает отрицательное значение, если вызывающий объект имеет меньшее значение. Возвращает положительное значение, если вызывающий объект имеет большее значение.
double doubleValue()	Возвращает значение вызывающего объекта как double.
boolean equals(Object *FloatObj*)	Возвращает true, если вызывающий объект Float эквивалентен *FloatObj*. В противном случае возвращает false.
static int floatToIntBits(float *num*)	Возвращает совместимый с IEEE битовый шаблон одинарной точности, который соответствует *num*.
static int floatToRawIntBits(float *num*)	Возвращает совместимый с IEEE битовый шаблон одинарной точности, который соответствует *num*. Значение NaN предохраняется.
float floatValue()	Возвращает значение вызывающего объекта как float.
int hashCode()	Возвращает хеш-код вызывающего объекта.
static float intBitsToFloat(int *num*)	Возвращает float-эквивалент совместимого с IEEE битового шаблона одинарной точности, который соответствует *num*.
int intValue()	Возвращает значение вызывающего объекта как int.
boolean isInfinite()	Возвращает true, если вызывающий объект содержит бесконечное значение. В противном случае возвращает false.

Окончание табл. 16.1

Метод	Описание
`static boolean isInfinite(float num)`	Возвращает `true`, если *num* определяет бесконечное значение. В противном случае возвращает `false`.
`boolean isNaN()`	Возвращает `true`, если вызывающий объект содержит значение, которое не является числом. В противном случае возвращает `false`.
`static boolean isNaN(float num)`	Возвращает `true`, если *num* определяет значение, не являющееся числом. В противном случае возвращает `false`.
`long longValue()`	Возвращает значение вызывающего объекта как `long`.
`static float parseFloat(String str) throws NumberFormatException`	Возвращает `float`-эквивалент числа с основанием 10, содержащегося в строке *str*.
`short shortValue()`	Возвращает значение вызывающего объекта как `short`.
`static String toHexString(float num)`	Возвращает строку, содержащую значение *num* в шестнадцатеричном формате (добавлен в J2SE 5).
`String toString()`	Возвращает строковый эквивалент вызывающего объекта.
`static String toString(float num)`	Возвращает строковый эквивалент значения, представленного в *num*.
`static Float valueOf(float num)`	Возвращает объект `Float`, содержащий значение, переданное в *num* (добавлен в J2SE 5).
`static Float valueOf(String str) throws NumberFormatException`	Возвращает объект `Float`, содержащий значение, представленное в строке *str*.

Таблица 16.2. Методы класса `Double`

Метод	Описание
`byte byteValue()`	Возвращает значение вызывающего объекта как `byte`.
`static int compare(double num1, double num2)`	Сравнивает значения *num1* и *num2*. Возвращает 0, если значения эквивалентны. Возвращает отрицательное значение, если *num1* меньше *num2*. Возвращает положительное значение, если *num1* больше *num2*.
`int compareTo(Double d)`	Сравнивает числовое значение вызывающего объекта с *d*. Возвращает 0, если значения эквивалентны. Возвращает отрицательное значение, если вызывающий объект имеет меньшее значение. Возвращает положительное значение, если вызывающий объект имеет большее значение
`static long doubleToLongBits(double num)`	Возвращает совместимый с IEEE битовый шаблон двойной точности, который соответствует *num*.
`static long doubleToRawLongBits (double num)`	Возвращает совместимый с IEEE битовый шаблон двойной точности, который соответствует *num*. Значение `NaN` предохраняется.
`double doubleValue()`	Возвращает значение вызывающего объекта как `double`.

Окончание табл. 16.2

Метод	Описание
`boolean equals(Object` *`DoubleObj`*`)`	Возвращает `true`, если вызывающий объект `Double` эквивалентен `DoubleObj`.
`float floatValue()`	Возвращает значение вызывающего объекта как `float`.
`int hashcode()`	Возвращает хеш-код вызывающего объекта.
`int intValue()`	Возвращает значение вызывающего объекта как `int`.
`boolean isInfinite()`	Возвращает `true`, если вызывающий объект содержит бесконечное значение. В противном случае возвращает `false`.
`static boolean isInfinite(double` *`num`*`)`	Возвращает `true`, если *`num`* специфицирует бесконечное значение. В противном случае возвращает `false`.
`boolean isNaN()`	Возвращает `true`, если вызывающий объект содержит нечисловое значение. В противном случае возвращает `false`.
`static boolean isNaN(double` *`num`*`)`	Возвращает `true`, если *`num`* специфицирует нечисловое значение. В противном случае возвращает `false`.
`static double longBitsToDouble(long` *`num`*`)`	Возвращает `double`-эквивалент совместимого с IEEE битового шаблона, специфицированного в *`num`*.
`long longValue()`	Возвращает значение вызывающего объекта как `long`.
`static double parseDouble(String` *`str`*`) throws NumberFormatException`	Возвращает `double`-эквивалент числа с основанием 10, содержащегося в строке *`str`*.
`short shortValue()`	Возвращает значение вызывающего объекта как `short`.
`static String toHexString(double` *`num`*`)`	Возвращает строку, содержащую значение *`num`* в шестнадцатеричном формате.
`String toString()`	Возвращает строковый эквивалент вызывающего объекта.
`static String toString(double` *`num`*`)`	Возвращает строковый эквивалент значения, специфицированного в *`num`*.
`static Double valueOf(double` *`num`*`)`	Возвращает объект `Double`, содержащий значение, переданное в *`num`*.
`static Double valueOf(String` *`str`*`) throws NumberFormatException`	Возвращает объект `Double`, содержащий значение, переданное в строке *`str`*.

В следующем примере создаются два объекта `Double` — один с использованием значения `double`, а второй — с использованием строки, которая может быть интерпретирована как `double`:

```
class DoubleDemo {
public static void main(String args[]) {
  Double d1 = new Double(3.14159);
  Double d2 = new Double("314159E-5");

  System.out.println(d1 + " = " + d2 + " -> " + d1.equals(d2));
}
}
```

Как вы можете видеть из следующего вывода, оба конструктора создают идентичные экземпляры Double, что доказывает метод equals(), возвращающий true:

```
3.14159 = 3.14159 -> true
```

isInfinite() и isNaN()

Float и Double предлагают методы isInfinite() и isNaN(), которые помогают манипулировать двумя специальными значениями double и float. Эти методы осуществляют проверку на предмет равенства двум уникальным значениям, определенным спецификациями плавающей точки стандарта IEEE: бесконечности и NaN (не число). isInfinite() возвращает true, если проверяемое число бесконечно велико или бесконечно мало по величине. isNaN() возвращает true, если проверяемое значение не является числовым.

В следующем примере создаются два объекта Double: один содержит бесконечность, а второй — нечисловое значение.

```
// Демонстрация применения 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());
}
}
```

Программа генерирует следующий вывод:

```
Infinity: true, false
NaN: false, true
```

Byte, Short, Integer и Long

Классы Byte, Short, Integer и Long — это оболочки для целочисленных типов byte, short, int и long соответственно. Ниже показаны конструкторы.

```
Byte(byte num)
Byte(String str) throws NumberFormatException

Short(short num)
Short(String str) throws NumberFormatException

Integer(int num)
Integer(String str) throws NumberFormatException

Long(long num)
Long(String str) throws NumberFormatException
```

Как видите, эти объекты могут быть сконструированы из числовых значений или из строк, которые содержат допустимые представления числовых значений.

Методы, определенные этими двумя классами, показаны в таблицах от 16.3 до 16.6. Как вы можете видеть, они определяют методы для разбора целых чисел из строк и преобразования строк обратно в целые. Вариации этих методов позволяют вам указать *radix* — основание числа для преобразования. Чаще всего применяются основание 2 — для двоичных, 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	Ширина помещенного в оболочку значения в битах.
TYPE	Объект Class для byte, short, int или long.

Таблица 16.3. Методы класса Byte

Метод	Описание
byte byteValue()	Возвращает значение вызывающего объекта, как byte.
int compareTo(Byte *b*)	Сравнивает числовое значение вызывающего объекта с *b*. Возвращает 0, если значения равны. Возвращает отрицательное число, если вызывающий объект имеет меньшее значение. Возвращает положительное число, если вызывающий объект имеет большее значение.
static Byte decode(String *str*) throws NumberFormatException	Возвращает объект Byte, который содержит значение, указанное в строке *str*.
double doubleValue()	Возвращает значение вызывающего объекта как double.
boolean equals(Object *ByteObj*)	Возвращает true, если вызывающий объект Byte эквивалентен ByteObj. В противном случае возвращает false.
float floatValue()	Возвращает значение вызывающего объекта как float.
int hashCode()	Возвращает хеш-код вызывающего объекта.
int intValue()	Возвращает значение вызывающего объекта как int.
long longValue()	Возвращает значение вызывающего объекта как long.
static byte parseByte(String *str*) throws NumberFormatException	Возвращает byte-эквивалент числа с основанием 10, переданного в строке *str*.
static byte parseByte(String *str*, int *radix*) throws NumberFormatException	Возвращает byte-эквивалент числа с основанием radix, переданного в строке *str*.
short shortValue()	Возвращает значение вызывающего объекта как short.
String toString()	Возвращает строку, которая содержит десятичный эквивалент вызывающего объекта.
static String toString(byte *num*)	Возвращает строку, которая содержит десятичный эквивалент *num*.
static Byte valueOf(byte *num*)	Возвращает объект Byte, содержащий значение, переданное в *num*.
static Byte valueOf(String *str*) throws NumberFormatException	Возвращает объект Byte, содержащий значение, специфицированное в строке *str*.
static Byte valueOf(String *str*, int *radix*) throws NumberFormatException	Возвращает объект Byte, содержащий значение, специфицированное в строке *str* с использованием основания *radix*.

Таблица 16.4. Методы класса `Short`

Метод	Описание
`byte byteValue()`	Возвращает значение вызывающего объекта как `byte`.
`int compareTo(Short s)`	Сравнивает числовое значение вызывающего объекта с *s*. Возвращает 0, если значения равны. Возвращает отрицательное число, если вызывающий объект имеет меньшее значение. Возвращает положительное число, если вызывающий объект имеет большее значение.
`static Short decode(String str) throws NumberFormatException`	Возвращает объект `Short`, который содержит значение, указанное в строке *str*.
`double doubleValue()`	Возвращает значение вызывающего объекта, как `double`
`boolean equals(Object ShortObj)`	Возвращает `true`, если вызывающий объект `Short` эквивалентен `ShortObj`. В противном случае возвращает `false`.
`float floatValue()`	Возвращает значение вызывающего объекта как `float`.
`int hashCode()`	Возвращает хеш-код вызывающего объекта.
`int intValue()`	Возвращает значение вызывающего объекта как `int`.
`long longValue()`	Возвращает значение вызывающего объекта как `long`.
`static short parseShort(String str) throws NumberFormatException`	Возвращает `short`-эквивалент числа с основанием 10, переданного в строке *str*.
`static short parseShort(String str, int radix) throws NumberFormatException`	Возвращает `short`-эквивалент числа с основанием `radix`, переданного в строке *str*.
`static short reverseBytes(short num)`	Меняет местами старший и младший байты *num* и возвращает результат.
`short shortValue()`	Возвращает значение вызывающего объекта как `short`.
`String toString()`	Возвращает строку, которая содержит десятичный эквивалент вызывающего объекта.
`static String toString(short num)`	Возвращает строку, которая содержит десятичный эквивалент *num*.
`static Short valueOf(short num)`	Возвращает объект `Short`, содержащий значение, переданное в *num*.
`static Short valueOf(String str) throws NumberFormatException`	Возвращает объект `Short`, содержащий значение, специфицированное в строке *str*.
`static Short valueOf(String str, int radix) throws NumberFormatException`	Возвращает объект `Short`, содержащий значение, специфицированное в строке *str* с использованием основания *radix*.

Таблица 16.5. Методы класса `Integer`

Метод	Описание
`static int bitCount(int num)`	Возвращает количество бит в *num*.
`byte byteValue()`	Возвращает значение вызывающего объекта как `byte`.
`int compareTo(Integer i)`	Сравнивает числовое значение вызывающего объекта с *i*. Возвращает 0, если значения равны. Возвращает отрицательное число, если вызывающий объект имеет меньшее значение. Возвращает положительное число, если вызывающий объект имеет большее значение.
`static Short decode(String str) throws NumberFormatException`	Возвращает объект `Short`, который содержит значение, указанное в строке *str*.
`double doubleValue()`	Возвращает значение вызывающего объекта как `double`.
`boolean equals(Object IntegerObj)`	Возвращает `true`, если вызывающий объект `Integer` эквивалентен *IntegerObj*. В противном случае возвращает `false`.
`float floatValue()`	Возвращает значение вызывающего объекта как `float`.
`static Integer getInteger(String propertyName)`	Возвращает значение, ассоциированное со свойством окружения, указанным в *propertyName*. В случае неудачи возвращается `null`.
`static Integer getInteger(String propertyName, int default)`	Возвращает значение, ассоциированное со свойством окружения, указанным в *propertyName*. В случае неудачи возвращается *default*.
`static Integer getInteger(String propertyName, Integer default)`	Возвращает значение, ассоциированное со свойством окружения, указанным в *propertyName*. В случае неудачи возвращается *default*.
`int hashCode()`	Возвращает хеш-код вызывающего объекта.
`static int highestOneBit(int num)`	Определяет позицию самого старшего бита в *num*. Возвращает значение, в котором установлен только этот бит. Если ни одного бита не установлено, возвращается ноль.
`int intValue()`	Возвращает значение вызывающего объекта как `int`.
`long longValue()`	Возвращает значение вызывающего объекта как `long`.
`static int lowestOneBit(int num)`	Определяет позицию самого младшего бита в *num*. Возвращает значение, в котором установлен только этот бит. Если установленных в 1 битов нет, возвращается ноль.
`static int numberOfLeadingZeros(int num)`	Возвращает количество старших бит, установленных в ноль, которые предшествуют первому установленному старшему биту в *num*. Если *num* равно 0, возвращается 32.
`static int numberOfTrailingZeros( int num)`	Возвращает количество младших бит, установленных в ноль, которые предшествуют первому установленному младшему биту в *num*. Если *num* равно 0, возвращается 32.
`static int parseInt(String str) throws NumberFormatException`	Возвращает целочисленный эквивалент числа с основанием 10, переданного в строке *str*.

Окончание табл. 16.5

Метод	Описание
`static int parseInt(String str, int radix) throws NumberFormatException`	Возвращает целочисленный эквивалент числа с основанием *radix*, переданного в строке *str*.
`static int reverse(int num)`	Изменяет порядок бит в *num* на противоположный и возвращает результат.
`static int reverseBytes(int num)`	Изменяет порядок байт в *num* на противоположный и возвращает результат.
`static int rotateLeft(int num, int n)`	Возвращает результат смещения *num* на *n* позиций влево.
`static int rotateRight(int num, int n)`	Возвращает результат смещения *num* на *n* позиций вправо.
`static int signum(int num)`	Возвращает –1, если *num* отрицательное, 0 — если ноль и 1 — если положительное.
`short shortValue()`	Возвращает значение вызывающего объекта как `short`.
`static String toBinaryString(int num)`	Возвращает строку, содержащую двоичный эквивалент *num*.
`static String toHexString(int num)`	Возвращает строку, содержащую шестнадцатеричный эквивалент *num*.
`static String toOctalString(int num)`	Возвращает строку, содержащую восьмеричный эквивалент *num*.
`String toString()`	Возвращает строку, которая содержит десятичный эквивалент вызывающего объекта.
`static String toString(int num)`	Возвращает строку, которая содержит десятичный эквивалент *num*.
`static String toString(int num, int radix)`	Возвращает строку, которая содержит десятичный эквивалент значения, специфицированного в строке *str* с использованием основания *radix*.
`static Integer valueOf(int num)`	Возвращает объект `Integer`, содержащий значение, переданное в *num*.
`static Integer valueOf(String str) throws NumberFormatException`	Возвращает объект `Integer`, содержащий значение, специфицированное в строке *str*.
`static Integer valueOf(String str, int radix) throws NumberFormatException`	Возвращает объект `Integer`, содержащий значение, специфицированное в строке *str* с использованием основания *radix*.

Таблица 16.6. Методы класса Long

Метод	Описание
`static int bitCount(long num)`	Возвращает количество бит в *num*.
`byte byteValue()`	Возвращает значение вызывающего объекта как `byte`.
`int compareTo(Long l)`	Сравнивает числовое значение вызывающего объекта с *l*. Возвращает 0, если значения равны. Возвращает отрицательное число, если вызывающий объект имеет меньшее значение. Возвращает положительное число, если вызывающий объект имеет большее значение.
`static Long decode(String str) throws NumberFormatException`	Возвращает объект `Long`, который содержит значение, указанное в строке *str*.
`double doubleValue()`	Возвращает значение вызывающего объекта, как `double`.
`boolean equals(Object LongObj)`	Возвращает `true`, если вызывающий объект `Long` эквивалентен *LongObj*. В противном случае возвращает `false`.
`float floatValue()`	Возвращает значение вызывающего объекта как `float`.
`static Long getLong(String propertyName)`	Возвращает значение, ассоциированное со свойством окружения, указанным в *propertyName*. В случае неудачи возвращается `null`.
`static Long getLong(String propertyName, long default)`	Возвращает значение, ассоциированное со свойством окружения, указанным в *propertyName*. В случае неудачи возвращается *default*.
`static Long getLong(String propertyName, Long default)`	Возвращает значение, ассоциированное со свойством окружения, указанным в *propertyName*. В случае неудачи возвращается *default*.
`int hashCode()`	Возвращает хеш-код вызывающего объекта.
`static int highestOneBit( long num)`	Определяет позицию самого старшего бита в *num*. Возвращает значение, в котором установлен только этот бит. Если ни одного бита не установлено, возвращается ноль.
`int intValue()`	Возвращает значение вызывающего объекта как `int`.
`long longValue()`	Возвращает значение вызывающего объекта как `long`.
`static int lowestOneBit( long num)`	Определяет позицию самого младшего бита в *num*. Возвращает значение, в котором установлен только этот бит. Если ни одного бита не установлено, возвращается ноль.
`static int numberOfLeadingZeros( long num)`	Возвращает количество старших бит, установленных в ноль, которые предшествуют первому установленному старшему биту в *num*. Если *num* равно 0, возвращается 64.
`static int numberOfTrailingZeros(long num)`	Возвращает количество младших бит, установленных в ноль, которые предшествуют первому установленному младшему биту в *num*. Если *num* равно 0, возвращается 64.
`static long parseLong( String str) throws NumberFormatException`	Возвращает `long`-эквивалент числа с основанием 10, переданного в строке *str*.

Окончание табл. 16.6

Метод	Описание
`static long parseInt(String str, int radix) throws NumberFormatException`	Возвращает `long`-эквивалент числа с основанием *radix*, переданного в строке *str*.
`static long reverse(long num)`	Изменяет порядок бит в *num* на противоположный и возвращает результат.
`static long reverseBytes(long num)`	Изменяет порядок байт в *num* на противоположный и возвращает результат.
`static long rotateLeft(long num, int n)`	Возвращает результат смещения *num* на *n* позиций влево.
`static long rotateRight(long num, int n)`	Возвращает результат смещения *num* на *n* позиций вправо.
`static int signum(int num)`	Возвращает -1, если *num* отрицательное, 0 — если ноль и 1 — если положительное.
`short shortValue()`	Возвращает значение вызывающего объекта как `short`.
`static String toBinaryString(long num)`	Возвращает строку, содержащую бинарный эквивалент *num*.
`static String toHexString(long num)`	Возвращает строку, содержащую шестнадцатеричный эквивалент *num*.
`static String toOctalString(long num)`	Возвращает строку, содержащую восьмеричный эквивалент *num*.
`String toString()`	Возвращает строку, которая содержит десятичный эквивалент вызывающего объекта.
`static String toString(long num)`	Возвращает строку, которая содержит десятичный эквивалент *num*.
`static String toString(long num, int radix)`	Возвращает строку, которая содержит десятичный эквивалент значения, специфицированного в строке *str* с использованием основания *radix*.
`static Long valueOf(long num)`	Возвращает объект `Long`, содержащий значение, переданное в *num*.
`static Long valueOf(String str) throws NumberFormatException`	Возвращает объект `Long`, содержащий значение, специфицированное в строке *str*.
`static Long valueOf(String str, int radix) throws NumberFormatException`	Возвращает объект `Long`, содержащий значение, специфицированное в строке *str* с использованием основания *radix*.

Преобразование чисел в строки и обратно

Одной из наиболее часто выполняемых рутинных операций в программировании является преобразование строкового представления чисел во внутренний двоичный формат. К счастью, в Java имеется простой способ осуществления этого. Классы Byte, Short, Integer и Long предлагают методы parseByte(), parseShort(), parseInt() и parseLong() соответственно. Эти методы возвращают byte, short, int или long — эквиваленты числовой строки, с которой они были вызваны (аналогичные методы также предусмотрены в классах Float и Double).

В следующей программе демонстрируется применение parseInt(). Она суммирует список целых, введенных пользователем. Для этого программа читает целые значения с помощью readLine() и использует parseInt() для преобразования этих строк в их int-эквиваленты.

```
/* Эта программа суммирует список целых чисел, введенных пользователем.
   Она преобразует строковое представление каждого числа в целое,
   используя parseInt()
*/
import java.io.*;
class ParseDemo {
public static void main(String args[])
throws IOException
{
  // Создать BufferedReader, используя 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);
}
}
```

Чтобы преобразовать число в десятичную строку, используйте версии toString(), определенные в классах Byte, Short, Integer или Long. Классы Integer и Long также предоставляют методы toBinaryString(), toHexString() и toOctalString(), которые преобразуют значение в бинарную, шестнадцатеричную и восьмеричную строки соответственно:

```
/* Преобразует целое в бинарный, шестнадцатеричный и восьмеричный формат */
class StringConversions {
public static void main(String args[]) {
  int num = 19648;
  System.out.println(num + " в бинарной форме: " + Integer.toBinaryString(num));
  System.out.println(num + " в восьмеричной форме: " +
                     Integer.toOctalString(num));
```

```
    System.out.println(num + " в шестнадцатеричной форме: " +
                       Integer.toHexString(num));
  }
  }
```

Вывод этой программы показан ниже:

```
19648 в бинарной форме: 100110011000000
19648 в восьмеричной форме: 46300
19648 в шестнадцатеричной форме: 4cc0
```

Character

`Character` — это простая оболочка для `char`. Конструктор `Character` выглядит следующим образом:

```
Character(char ch)
```

Здесь *ch* определяет символ, который будет помещен в оболочку создаваемого объекта `Character`. Чтобы получить значение `char`, содержащееся в объекте `Character`, вызовите метод `charValue()`, показанный ниже:

```
char charValue()
```

Этот метод вернет символ.

В классе `Character` определено несколько констант, включая следующие:

`MAX_RADIX`	Максимальное основание.
`MIN_RADIX`	Минимальное основание.
`MAX_VALUE`	Максимальное значение.
`MIN_VALUE`	Минимальное значение.
`TYPE`	Объект `Class` для `char`.

Класс `Character` включает несколько статических методов, которые категоризируют символы и изменяют их регистр. Они описаны в табл. 16.7. В следующем примере демонстрируется применение некоторые из этих методов.

```
// Демонстрация применения некоторых методов 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] + " - десятичное число.");
    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] + " - символ верхнего регистра.");
    if(Character.isLowerCase(a[i]))
      System.out.println(a[i] + " - символ нижнего регистра.");
  }
}
}
```

Вывод этой программы выглядит так:

```
a - буква.
a - символ нижнего регистра.
b - буква.
b - символ нижнего регистра.
5 - десятичное число.
A - буква.
A - символ верхнего регистра.
  - пробельный символ.
```

Таблица 16.7. Различные методы `Character`

Метод	Описание
`static boolean isDefined(char ch)`	Возвращает `true`, если *ch* определен в Unicode. В противном случае возвращает `false`.
`static boolean isDigit(char ch)`	Возвращает `true`, если *ch* является десятичной цифрой. В противном случае возвращает `false`.
`static Boolean isIdentifierIgnorable( char ch)`	Возвращает `true`, если *ch* должен быть проигнорирован в идентификаторе. В противном случае возвращает `false`.
`static boolean isISOControl(char ch)`	Возвращает `true`, если *ch* является управляющим символом ISO. В противном случае возвращает `false`.
`static Boolean isJavaIdentifierPart( char ch)`	Возвращает `true`, если *ch* может быть частью идентификатора Java. В противном случае возвращает `false`.
`static Boolean isJavaIdentifierStart( char ch)`	Возвращает `true`, если *ch* может быть первым символом идентификатора Java. В противном случае возвращает `false`.
`static boolean isLetter(char ch)`	Возвращает `true`, если *ch* — буква. В противном случае возвращает `false`.
`static Boolean isLetterOrDigit( char ch)`	Возвращает `true`, если *ch* — буква или цифра. В противном случае возвращает `false`.
`static boolean isLowerCase(char ch)`	Возвращает `true`, если *ch* — буква в нижнем регистре. В противном случае возвращает `false`.
`static boolean isMirrored(char ch)`	Возвращает `true`, если *ch* — зеркальный Unicode-символ. Зеркальный символ означает один их зарезервированных для текстов, отображаемых справа налево.
`static boolean isSpaceChar(char ch)`	Возвращает `true`, если *ch* — пробельный символ Unicode. В противном случае возвращает `false`.
`static boolean isTitleCase(char ch)`	Возвращает `true`, если *ch* — титульный символ. В противном случае возвращает `false`.
`static Boolean isUnicodeIdentifierPart (char ch)`	Возвращает `true`, если *ch* допустим в качестве части идентификатора Unicode (кроме первого символа). В противном случае возвращает `false`.

Окончание табл. 16.7

Метод	Описание
`static Boolean isUnicodeIdentifierStart(char ch)`	Возвращает `true`, если *ch* допустим в качестве первого символа идентификатора Unicode. В противном случае возвращает `false`.
`static boolean isUpperCase(char ch)`	Возвращает `true`, если *ch* — символ верхнего регистра. В противном случае возвращает `false`.
`static boolean isWhitespace(char ch)`	Возвращает `true`, если *ch* — пробельный символ. В противном случае возвращает `false`.
`static char toLowerCase(char ch)`	Возвращает эквивалент *ch* в нижнем регистре.
`static char toTitleCase(char ch)`	Возвращает эквивалент *ch* в титульном регистре.
`static char toUpperCase(char ch)`	Возвращает эквивалент *ch* в верхнем регистре.

В `Character` определены два метода — `forDigits()` и `digit()`, которые позволяют выполнять преобразования между целыми значениями и цифрами, их представляющими. Выглядят они следующим образом:

```
static char forDigit(int num, int radix)
static int digit(char digit, int radix)
```

`forDigit()` возвращает десятичную цифру, ассоциированную со значением *num*. Основание для преобразования задается параметром *radix*. *digit()* возвращает целое, ассоциированное с указанным символом (предполагается, цифрой) в соответствии с заданным основанием.

Другой метод, определенный в классе `Character` — это `compareTo()`, который имеет следующую форму:

```
int compareTo(Character c)
```

Возвращает ноль, если вызывающий объект и *c* эквивалентны. Возвращает отрицательное значение, если вызывающий объект содержит меньшее значение. В противном случае возвращает положительное значение.

`Character` содержит метод по имени `getDirectionality()`, который может быть использован для определения направления символа. Несколько констант добавлены для описания направления написания символа. Большинство программ в этом не нуждаются.

`Character` также переопределяет методы `equals()` и `hashCode()`.

Два других ориентированных на символы класса — это `Character.Subset`, используемый для описания подмножества Unicode, и `Character.UnicodeBlock`, содержащий блоки символов Unicode.

Дополнения к `Character` для поддержки кодовых точек Unicode

В последнее время в `Character` появились существенные дополнения. Начиная с JDK 5, класс `Character` обеспечивают поддержку 32-битных символов Unicode. В прошлом все символы Unicode состояли из 16 бит, что равно размеру `char` (и размеру значения, инкапсулированного в `Character`), поскольку эти символы находятся в диапазоне от 0 до FFFF. Однако набор символов Unicode был расширен, и понадобилось более 16 бит. Теперь символы расположены в диапазоне от 0 до 10FFFF.

Появилось три важных термина: кодовая точка (code point), кодовая единица (code unit) и дополнительный символ (supplemental character). Что касается Java, *кодовая точка* — это символ в диапазоне от 0 до 10FFFF. Java использует термин *кодовая единица* для ссылки на 16-битные символы. Символы, имеющие код свыше FFFF, называются *дополнительными символами*.

Расширение набора символов Unicode создает фундаментальные проблемы для Java. Поскольку дополнительные символы имеют значение, большее, чем умещается в `char`, для их поддержки требуются дополнительные средства. В Java эта проблема решается двумя способами. Во-первых, Java использует два `char` для представления дополнительных символов. Первый из них называется *старшей сигнатурой* (high signature), а второй — *младшей сигнатурой* (low signature). Для трансляции между кодовыми точками и дополнительными символами предусмотрены новые методы, такие как `codePointAt()`.

Во-вторых, Java переопределяет некоторые из ранее существовавших методов класса `Character`. Перегруженные формы используют данные типа `int` вместо `char`. Поскольку тип `int` достаточно велик, чтобы уместить любой символ как одно значение, его можно использовать для хранения любого символа. Например, все методы из табл. 16.7 имеют перегруженные формы, которые оперируют `int`. Вот примеры:

```
static boolean isDigit(int cp)
static boolean isLetter(int cp)
static int toLowerCase(int cp)
```

В дополнение к методам, перегруженным для работы с кодовыми точками, в `Character` также добавлены методы, которые предлагают дополнительную поддержку кодовых точек. Их примеры можно найти в табл. 16.8.

Таблица 16.8. Примеры методов, которые обеспечивают поддержку 32-битных кодовых точек Unicode

Метод	Описание
`static int charCount(int cp)`	Возвращает 1, если *cp* может быть представлен одним `char`. Возвращает 2, если требуется два `char`.
`static int codePointAt(CharSequence chars, int loc)`	Возвращает кодовую точку, находящуюся в позиции *loc*.
`static int codePointAt(char chars[], int loc)`	Возвращает кодовую точку, находящуюся в позиции *loc*.
`static int codePointBefore(CharSequence chars, int loc)`	Возвращает кодовую точку, находящуюся в позиции, предшествующей *loc*.
`static int codePointBefore(char chars[], int loc)`	Возвращает кодовую точку, находящуюся в позиции, предшествующей *loc*.
`static boolean isHighSurrogate(char ch)`	Возвращает `true`, если *ch* представляет корректный символ верхней сигнатуры.
`static boolean isLowSurrogate(char ch)`	Возвращает `true`, если *ch* представляет корректный символ нижней сигнатуры.
`static boolean isSupplementaryCodePoint(int cp)`	Возвращает `true`, если *ch* представляет собой дополнительный символ.
`static boolean isSurrogatePair(char highCh, char lowCh)`	Возвращает `true`, если *highCh* и *lowCh* формируют корректную суррогатную пару.

Окончание табл. 16.8

Метод	Описание
`static boolean isValidCodePoint(int cp)`	Возвращает `true`, если *`cp`* представляет корректную кодовую точку.
`static char[] toChars(int cp)`	Преобразует кодовую точку в *`cp`* в `char`-эквивалент, который может потребовать двух `char`. Возвращает массив, содержащий результат.
`static int toChars(int cp, char target[], int loc)`	Преобразует кодовую точку *`cp`* в `char`-эквивалент, сохраняя результат в *`target`*, начиная с позиции *`loc`*. Возвращает 1, если *`cp`* может быть представлена одним *`char`*. В противном случае возвращает 2.
`static int toCodePoint(char highCh, char lowCh)`	Преобразует *`highCh`* и *`lowCh`* в эквивалентную кодовую точку.

Boolean

`Boolean` — это очень тонкая оболочка вокруг значений типа `boolean`, что удобно в основном тогда, когда вы хотите передавать значения `boolean` по ссылке. Этот класс содержит константы `TRUE` и `FALSE`, определяющие объекты `Boolean`, которые соответствуют истинному и ложному значениям. `Boolean` также определяет поле `TYPE`, являющееся объектом `Class` для `boolean`. `Boolean` определяет следующие конструкторы:

```
Boolean(boolean boolValue)
Boolean(String boolString)
```

В первой версии *`boolValue`* должно быть либо `true`, либо `false`. Во второй версии конструктора, если *`boolString`* содержит строку "true" (в верхнем или нижнем регистре), то новый объект `Boolean` будет `true`. В противном случае — `false`.

В `Boolean` определены методы, перечисленные в табл. 16.9.

Таблица 16.9. Методы класса `Boolean`

Метод	Описание
`boolean booleanValue()`	Возвращает `boolean`-эквивалент.
`int compareTo(Boolean b)`	Возвращает ноль, если вызывающий объект и *`b`* содержат одинаковые значения. Возвращает положительное значение, если вызывающий объект равен `true`, а `b` — `false`. В противном случае возвращает отрицательное значение.
`boolean equals(Object boolObj)`	Возвращает `true`, если вызывающий объект эквивалентен *`boolObj`*. В противном случае возвращает `false`.
`static boolean getBoolean( String propertyName)`	Возвращает `true`, если системное свойство, указанное в *`propertyName`* является `true`. В противном случае возвращает `false`.
`int hashCode()`	Возвращает хеш-код вызывающего объекта.
`static boolean parseBoolean(String str)`	Возвращает `true`, если *`str`* содержит строку "true". Регистр символов не важен. В противном случае возвращает `false`.
`String toString()`	Возвращает строковый эквивалент вызывающего объекта.

Окончание табл. 16.9

Метод	Описание
`static String toString(boolean boolVal)`	Возвращает строковый эквивалент *boolVal*.
`static Boolean valueOf(boolean boolVal)`	Возвращает `Boolean`-эквивалент *boolVal*.
`static Boolean valueOf(String boolString)`	Возвращает `true`, если *boolString* содержит строку "true" (в верхнем или нижнем регистре). В противном случае возвращает `false`.

Void

Класс `Void` имеет одно поле `TYPE`, которое содержит ссылку на объект `Class` для типа `void`. Экземпляры этого класса не создаются.

Process

Абстрактный класс `Process` инкапсулирует *процесс*, то есть выполняющуюся программу. Он используется в основном как суперкласс для типа объектов, созданных методом `exec()` класса `Runtime` или методом `start()` класса `ProcessBuilder`. `Process` содержит абстрактные методы, описанные в табл. 16.10.

Таблица 16.10. Методы класса `Process`

Метод	Описание
`void destroy()`	Прерывает процесс.
`int extValue()`	Возвращает код выхода подпроцесса.
`InputStream getErrorStream()`	Возвращает входной поток, который читает ввод из выходного потока `err` процесса.
`InputStream getOutputStream()`	Возвращает входной поток, который читает ввод из выходного потока `out` процесса.
`OutputStream getOutputStream()`	Возвращает выходной поток, который записывает вывод во входной поток `in` процесса.
`Int waitFor() throws InterruptedException`	Возвращает выходной код процесса. Этот метод не возвращает управление до тех пор, пока процесс, на котором он вызван, не завершится.

Runtime

Класс `Runtime` инкапсулирует среду времени выполнения. Создать объект `Runtime` невозможно. Однако можно получить ссылку на текущий объект `Runtime`, вызвав статический метод `Runtime.getRuntime()`. Получив ссылку на текущий объект `Runtime`, вы можете вызвать несколько методов, контролирующих состояние и поведение виртуальной машины Java (Java Virtual Machine — JVM). Аплеты и другой не доверенный код не может вызвать ни одного метода `Runtime` без возбуждения исключения `SecurityException`. Часто используемые методы класса `Runtime`, перечислены в табл. 16.11.

Таблица 16.11. Примеры методов, определенных в классе `Runtime`

Метод	Описание
`void addShutdownHook(Thread thrd)`	Регистрирует *thrd* как поток, который должен быть запущен при останове виртуальной машины Java.
`Process exec(String progName) throws IOException`	Выполняет программу, указанную в параметре *progName*, как отдельный процесс. Возвращается объект типа `Process`, описывающий новый процесс.
`Process exec(String progName, String environment[]) throws IOException`	Выполняет программу, указанную в параметре *progName*, как отдельный процесс в окружении, определенном параметром *environment*. Возвращается объект типа `Process`, описывающий новый процесс.
`Process exec(String comLineArray[]) throws IOException`	Выполняет командную строку, переданную в параметре *comLineArray*, как отдельный процесс. Возвращается объект типа `Process`, описывающий новый процесс.
`Process exec(String comLineArray[], String environment[]) throws IOException`	Выполняет командную строку, переданную в параметре *comLineArray*, как отдельный процесс в окружении, описанном параметром *environment*. Возвращается объект типа `Process`, описывающий новый процесс.
`void exit(int exitCode)`	Прерывает выполнение и возвращает значение *exitCode* родительскому процессу. По соглашению 0 означает нормальное завершение. Все другие значения символизируют различные типы ошибок.
`long freeMemory()`	Возвращает приблизительное количество байт свободной памяти, доступной системе времени выполнения Java.
`void gc()`	Инициирует сборку мусора.
`static Runtime getRuntime()`	Возвращает текущий объект `Runtime`.
`void halt(int code)`	Немедленно прерывает работу виртуальной машины Java. Никакие завершающие потоки или финализация не выполняются. Вызывающему процессу возвращается значение *code*.
`void load(String libraryFileName)`	Загружает динамическую библиотеку, файл которой специфицирован параметром *libraryFileName*, где должен быть указан полный путь к нему.
`void loadLibrary(String libraryName)`	Загружает динамическую библиотеку, имя которой ассоциируется с *libraryName*.
`boolean removeShutdownHook(Thread thrd)`	Удаляет *thrd* из списка потоков, подлежащих запуску при останове виртуальной машины Java. Возвращает `true` в случае успеха, то есть если поток удален.

Окончание табл. 16.11

Метод	Описание
`void runFinalization()`	Инициирует вызовы методов `finalize()` для неиспользованных, но еще не возвращенных объектов.
`long totalMemory()`	Возвращает общее количество байт памяти, доступной программе.
`void traceInstructions(boolean traceOn)`	Включает и отключает трассировку инструкций в зависимости от значения *traceOn*. Если *traceOn* равно `true`, то трассировка включается, а если `false` — отключается.
`void traceMethodCalls(boolean traceOn)`	Включает и отключает трассировку вызовов методов в зависимости от значения *traceOn*. Если *traceOn* равно `true`, то трассировка включается, а если `false` — отключается.

Давайте взглянем на два наиболее частых способа применения класса `Runtime`: управление памятью и выполнение дополнительных процессов.

Управление памятью

Несмотря на то что Java предлагает автоматическую сборку мусора, иногда вы захотите узнать, насколько велика объектная куча и сколько свободной памяти осталось. Вы можете использовать эту информацию, например, чтобы проверить свой код на эффективность, или для того, чтобы предположить, сколько еще объектов определенного типа может быть инициализировано. Для получения этих значений служат методы `totalMemory()` и `freeMemory()`.

Как упоминалось в первой части книги, сборщик мусора Java запускается периодически для утилизации неиспользуемых объектов. Однако иногда может потребоваться собрать отброшенные объекты до того, как сборщик мусора будет запущен в очередной раз. Вы можете запускать его по требованию, вызывая метод `gc()`. Неплохо попробовать запустить `gc()` и вслед за ним `freeMemory()`, чтобы получить представление об использовании памяти. Далее выполняйте свой код и снова вызывайте `freeMemory()`, чтобы увидеть, сколько памяти он распределяет. В следующей программе иллюстрируется описанная идея.

```
// Демонстрация применения 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 вначале: " + mem1);
  r.gc();
  mem1 = r.freeMemory();
  System.out.println("Свободной памяти после сборки мусора: " + mem1);
```

```
    for(int i=0; i<1000; i++)
        someints[i] = new Integer(i); // Распределить Integer

    mem2 = r.freeMemory();
    System.out.println("Свободной памяти после распределения: "
                       + mem2);
    System.out.println("Использовано памяти для распределения: "
                       + (mem1-mem2));

    // отбросить Integers
    for(int i=0; i<1000; i++) someints[i] = null;

    r.gc(); // запуск сборщика мусора

    mem2 = r.freeMemory();
    System.out.println("Свободной памяти после сборки" +
                        " отброшенных Integer: " + mem2);
  }
  }
```

Пример вывода этой программы показан здесь (конечно, ваши реальные результаты могут отличаться):

```
Всего памяти: 1048568
Свободной памяти вначале: 751392
Свободной памяти после сборки мусора: 841424
Свободной памяти после распределения: 824000
Использовано памяти для распределения: 17424
Свободной памяти после сборки отброшенных Integer: 842640
```

Выполнение других программ

В безопасных средах вы можете использовать Java для выполнения других тяжеловесных процессов (то есть, программ) в многозадачной операционной системе. Некоторые формы метода `exec()` позволяют указывать программу, которую вы хотите выполнить, а также передавать ей входные параметры. Метод `exec()` возвращает объект `Process`, который затем может использоваться для управления взаимодействием вашей Java-программы с этим вновь запущенным процессом. Поскольку Java может функционировать на множестве платформ в средах разнообразных операционных систем, как следствие, `exec()` сильно зависит от среды.

В следующем примере используется `exec()` для запуска `notepad` — простого текстового редактора Windows. Очевидно, что этот пример должен выполняться на платформе операционной системы Windows.

```
// Демонстрация 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 этом примере, используется наиболее часто. Объектом Process, возвращенным exec(), можно манипулировать методами класса Process после того, как программа запущена. Вы можете уничтожить подпроцесс методом destroy(). Метод waitFor() заставит вашу программу ожидать завершения подпроцесса. Метод exitValue() возвратит значение, которое вернет подпроцесс по завершению. Обычно это будет 0, если никаких проблем не возникнет. Ниже представлен предыдущий пример exec(), модифицированный таким образом, чтобы он ожидал завершения запущенного процесса.

```
// Ожидает завершения 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 notepad.");
  }

  System.out.println("Notepad возвратил " + p.exitValue());
}
}
```

Пока выполняется подпроцесс, вы можете писать в его стандартный ввод и читать его стандартный вывод. Методы getOutputStream() и getInputStream() возвращают дескрипторы стандартных потоков in и out подпроцесса. (Ввод-вывод детально рассматривается в главе 19.)

ProcessBuilder

Класс ProcessBuilder обеспечивает другой способ запуска и управления процессами (т.е. программами). Как объяснялось ранее, все процессы представляются классом Process, и процесс может быть запущен методом Runtime.exec(). ProcessBuilder предлагает более развитые средства управления процессами. Например, вы можете установить текущий рабочий каталог и изменять параметры окружения. В ProcessBuilder определены следующие конструкторы:

```
ProcessBuilder(List<String> args)
ProccessBuilder(String ... args)
```

Здесь *args* — список аргументов, указывающих имя программы, которую нужно запустить, со всеми необходимыми аргументами командной строки. В первом конструкторе аргументы передаются через List. Во втором они указываются через vararg-параметр. В табл. 16.12 описаны методы, определенные в ProcessBuilder.

Чтобы создать процесс, используя ProcessBuilder, нужно просто создать экземпляр ProcessBuilder, указав имя программы и все необходимые аргументы. Чтобы начать выполнение программы, вызовите start() для этого созданного экземпляра. Ниже показан пример, который запускает notepad — текстовый редактор Windows. Обратите внимание, что в качестве аргумента передается имя файла, который нужно редактировать.

```
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.");
  }
}
}
```

Таблица 16.12. Методы, определенные в классе `ProcessBuilder`

Метод	Описание
`List<String> command()`	Возвращает ссылку на `List`, которая содержит имя программы и ее аргументы. Изменения в этом списке касаются вызванного процесса.
`ProcessBuilder command(List<String> args)`	Устанавливает имя программы и ее аргументы через параметр *`args`*. Изменения в этом списке касаются вызванного процесса. Возвращает ссылку на вызванный объект.
`ProcessBuilder command(String ... args)`	Устанавливает имя программы и ее аргументы через параметр *`args`*. Возвращает ссылку на вызванный объект.
`File directory()`	Возвращает текущий рабочий каталог вызванного объекта. Это значение может быть `null`, если каталог тот же, что и у Java-программы, которая запустила процесс.
`ProcessBuilder directory(File dir)`	Устанавливает текущий каталог для вызванного объекта. Возвращает ссылку на вызванный объект.
`Map<String, String> environment()`	Возвращает переменные окружения, ассоциированные с вызванным объектом, в виде пар "ключ-значение".
`boolean redirectErrorStream()`	Возвращает `true`, если стандартный поток ошибок перенаправлен на стандартный выходной поток. Возвращает `false`, если потоки различны.
`ProcessBuilder redirectErrorStream( boolean merge)`	Если *`merge`* равно `true`, то стандартный поток ошибок перенаправляется на стандартный вывод. Если *`merge`* равно `false`, то потоки разделены, что является состоянием по умолчанию. Возвращает ссылку на вызванный объект.
`Process start() throws IOException`	Запускает процесс, указанный в вызванном объекте. Другими словами, запускает заданную программу.

System

Класс System содержит коллекцию статических методов и переменных. Стандартный ввод, вывод и вывод ошибок исполняющей системы Java хранятся в переменных in, out и err. Методы, определенные в классе System, перечислены в табл. 16.13. Многие из них возбуждают исключение SecurityException, если операция не допускается диспетчером безопасности.

Давайте взглянем на некоторые случаи обычного применения System.

Таблица 16.13. Методы, определенные в классе System

Метод	Описание
`static void arraycopy(Object source, int sourceStart, Object target, int targetStart, int size)`	Копирует массив. Массив, подлежащий копированию, передается в *source*, а индекс позиции, с которой начинается копирование из *source*, передается в *sourceStart*. Массив, принимающий копию, передается в *target*, а индекс позиции в нем, куда нужно начать копирование — в параметре *targetStart*. *size* задает количество копируемых элементов.
`static String clearProperty(String which)`	Удаляет переменную окружения, указанную в *which*. Возвращается предыдущее значение, ассоциированное с *which*.
`static Console console()`	Возвращает консоль, ассоциированную с JVM. В случае отсутствия у JVM текущей консоли возвращается null (добавлен в Java SE 6).
`static long currentTimeMillis()`	Возвращает текущее время в миллисекундах, прошедших с полуночи 1 января 1970 года.
`static void exit(int exitCode)`	Прерывает выполнение и возвращает значение *exitCode* родительскому процессу (обычно операционной системе). По соглашению 0 означает нормальное завершение. Все другие значения обозначают разные варианты ошибок.
`static void gc()`	Инициирует сборку мусора.
`static Map<String, String> getenv()`	Возвращает Map, содержащий текущие переменные окружения и их значения.
`static String getenv(String which)`	Возвращает значение, ассоциированное с переменной окружения, имя которой передано в *which*.
`static Properties getProperties()`	Возвращает свойства, ассоциированные с системой времени выполнения Java (класс Properties описан в главе 17).
`static String getProperty(String which)`	Возвращает свойство, ассоциированное с *which*. Если описанный объект не найден, возвращается null.
`static String getProperty(String which, String default)`	Возвращает свойство, ассоциированное с *which*. Если описанный объект не найден, возвращается *default*.

Окончание табл. 16.13

Метод	Описание
`static SecurityManager getSecurityManager()`	Возвращает текущий диспетчер безопасности или объект `null`, если никакого диспетчера не установлено.
`static int identityHashCode(Object obj)`	Возвращает идентификационный хеш-код для *obj*.
`static Channel inheritedChannel() throws IOException`	Возвращает канал, унаследованный виртуальной машиной Java. Возвращает `null`, если никакого канала не унаследовано.
`static void load(String libraryFileName)`	Загружает динамическую библиотеку, файл которой специфицирован в *libraryFileName*, включая полный путь доступа.
`static void loadLibrary(String libraryName)`	Загружает динамическую библиотеку, имя которой ассоциировано с *libraryName*.
`static void mapLibraryName(String lib)`	Возвращает зависящее от платформы имя библиотеки *lib*.
`static long nanoTime()`	Получает наиболее точный таймер системы и возвращает его значение в наносекундах, прошедших от какого-то определенного начального момента. Точность таймера не известна.
`static void runFinalization()`	Инициирует вызов методов `finalize()` неиспользованных, но еще не утилизированных объектов.
`static void setErr(PrintStream eStream)`	Устанавливает стандартный поток ошибок `err` в *eStream*.
`static void setIn(InputStream iStream)`	Устанавливает стандартный входной поток `in` в *iStream*.
`static void setOut(PrintStream oStream)`	Устанавливает стандартный выходной поток `out` в *oStream*.
`static void setProperties(Properties sysProperties)`	Устанавливает текущие системные свойства в *sysProperties*.
`static String setProperty(String which, String v)`	Присваивает значение *v* свойству по имени *which*.
`static void setSecurityManager( SecurityManager secMan)`	Устанавливает диспетчер безопасности *secMan*.

Использование `currentTimeMills()` для измерения времени выполнения программы

Одно применение класса `System`, которое, возможно, вы сочтете интересным — это применение `currentTimeMills()` для фиксации времени выполнения различных частей вашей программы. Метод `currentTimeMills()` возвращает текущее время в миллисекундах, прошедшее с полуночи 1 января 1970 года. Чтобы хронометрировать часть вашей программы, сохраните это значение непосредственно перед началом выполнения этой части. Немедленно после выполнения вызовите `currentTimeMills()` еще раз. Временем

выполнения будет разница между конечным и начальным временем. Сказанное демонстрируется в следующей программе.

```
// Измерение времени выполнения программы.
class Elapsed {
public static void main(String args[]) {
  long start, end;

  System.out.println("Время прохода по циклу от 0 до 1 000 000");

  // Время прохода по циклу от 0 до 1 000 000
  start = System.currentTimeMillis(); // получить начальное время
  for(int i=0; i < 1000000; i++) ;
  end = System.currentTimeMillis(); // получить конечное время

  System.out.println("Время выполнения: " + (end-start));
}
}
```

Пример запуска этой программы (имейте в виду, что ваши результаты могут отличаться):

```
Время прохода по циклу от 0 до 1 000 000
Время выполнения: 10
```

Если таймер вашей системы имеет точность уровня наносекунд, вы можете переписать предшествующую программу с использованием `nanoTime()` вместо `currentTimeMills()`. Например, ниже показано ключевое изменение программы, переписанной для использования `nanoTime()`:

```
start = System.nanoTime();    // получить начальное время
for(int i=0; i < 1000000; i++) ;
end = System.nanoTime();      // получить конечное время
```

Использование `arraycopy()`

Метод `arraycopy()` может применяться для быстрого копирования массива любого типа из одного места в другое. Это намного быстрее, чем эквивалентный цикл, написанный чисто на Java. Ниже показан пример двух массивов, копируемых методом `arraycopy()`. Вначале `a` копируется в `b`. Затем все элементы `a` сдвигаются вниз на единицу. Затем `b` сдвигается вверх на одну позицию.

```
// Использование 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 вы можете видеть из следующего вывода, копировать можно один и тот же источник и приемник в обоих направлениях:

```
a = ABCDEFGHIJ
b = MMMMMMMMM
a = ABCDEFGHIJ
b = ABCDEFGHIJ
a = AABCDEFGHI
b = BCDEFGHIJJ
```

Свойства окружения

Доступны следующие свойства:

```
file.separator
java.class.path
java.class.version
java.compiler
java.ext.dirs
java.home
java.io.tmpdir
java.library.path
java.specification.name
java.specification.vendor
java.specification.version
java.vendor
java.vendor.url
java.version
java.vm.name
java.vm.specification.name
java.vm.specification.vendor
java.vm.specification.version
java.vm.vendor
java.vm.version
line.separator
os.arch
os.name
os.version
path.separator
user.dir
user.home
user.name
```

Вы можете получить значения различных переменных окружения, вызвав метод `System.getProperty()`. Например, следующая программа отображает путь к текущему пользовательскому каталогу:

```
class ShowUserDir {
public static void main(String args[]) {
  System.out.println(System.getProperty("user.dir"));
}
}
```

Object

Как упоминалось в первой части книги, `Object` — это суперкласс для всех других классов. В `Object` определены методы, перечисленные в табл. 16.14, которые применимы к любому объекту.

Таблица 16.14. Методы, определенные в классе `Object`

Метод	Описание
`Object clone() throws CloneNotSupportedException`	Создает новый объект, который повторяет вызывающий объект.
`boolean equals(Object` *`object`*`)`	Возвращает `true`, если вызывающий объект эквивалентен *`object`*.
`void finalize() throws Throwable`	Метод `finalize()` по умолчанию. Обычно переопределяется в подклассах.

Окончание табл. 16.14

Метод	Описание
`final Class <? > getClass()`	Получает объект `Class`, который описывает вызывающий объект.
`int hashCode()`	Возвращает хеш-код, ассоциированный с вызывающим объектом.
`final void notify()`	Прерывает выполнение потока, ожидающего вызывающего объекта.
`final void notifyAll()`	Прерывает выполнение всех потоков, ожидающих вызывающего объекта.
`String toString()`	Возвращает строку, описывающую объект.
`final void wait() throws InterruptedException`	Ожидает завершения выполнения другого потока.
`final void wait(long` *`milliseconds`*`) throws InterruptedException`	Ожидает до указанного числа миллисекунд завершения выполнения другого потока.
`final void wait(long` *`milliseconds`*`, int` *`nanoseconds`*`) throws InterruptedException`	Ожидает до указанного числа миллисекунд плюс наносекунд до завершения выполнения другого потока.

Использование `clone()` и интерфейса `Cloneable`

Большинство методов, определенных в `Object`, описываются в других местах настоящей книги. Однако один из них требует особого внимания: `clone()`. Метод `clone()` генерирует дубликат объекта, который его вызвал. Клонироваться могут только те классы, которые реализуют интерфейс `Cloneable`.

Интерфейс `Cloneable` не объявляет членов. Он служит для указания того факта, что класс допускает побитовое копирование объектов (то есть *клонирование*). Если вы попытаетесь вызвать `clone()` для класса, который не реализует интерфейс `Cloneable`, будет сгенерировано исключение `CloneNotSupportedException`. Когда создается клон, конструктор клонируемого объекта *не вызывается*. Клон — это просто точная копия оригинала.

Клонирование — потенциально опасное действие, поскольку оно может вызвать нежелательные побочные эффекты. Например, если объект, подвергаемый клонированию, содержит переменную-ссылку по имени *`objRef`*, то *`objRef`* в клоне будет ссылаться на тот же объект, что и *`objRef`* в оригинале. Если клон проведет изменения в содержимом объекта, на который ссылается *`objRef`*, то это также изменит исходный объект. Вот другой пример: если объект открывает поток ввода-вывода, а затем клонируется, то два объекта будут иметь дело с одним и тем же потоком. Более того, если один из этих объектов закроет поток, второй может все еще пытаться писать в него, что приведет к ошибке. В некоторых случаях вам понадобится переопределить метод `clone()`, определенный в `Object`, для того, чтобы справиться с проблемами подобного рода.

Поскольку клонирование может вызывать проблемы, `clone()` объявляется внутри `Object` как `protected`. Это значит, что он может быть вызван либо из метода, опреде-

ленного в классе, реализующем интерфейс Cloneable, либо он должен быть явно переопределен в классе как public. Давайте рассмотрим примеры каждого подхода.

В следующей программе реализуется Cloneable и определяется метод cloneTest(), который вызывает clone() в Object.

```
// Демонстрация применения метода clone().
class TestClone implements Cloneable {
int a;
double b;
// Этот метод вызывает clone() из Object.
TestClone cloneTest() {
  try {
     // вызвать clone() из 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 x1
  System.out.println("x1: " + x1.a + " " + x1.b);
  System.out.println("x2: " + x2.a + " " + x2.b);
}
}
```

Здесь метод cloneTest() вызывает clone() из Object и возвращает результат. Обратите внимание, что объект, возвращенный методом clone(), должен быть приведен к соответствующему типу (TestClone).

В следующем примере clone() перегружается таким образом, что он может быть вызван извне класса. Чтобы сделать это, спецификатором доступа должен быть public, как показано ниже:

```
// Переопределение метода clone().
class TestClone implements Cloneable {
int a;
double b;
// clone() переопределен как public.
public Object clone() {
  try {
     // вызов clone() из 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 clone() вызывается непосредственно.
  x2 = (TestClone) x1.clone();
  System.out.println("x1: " + x1.a + " " + x1.b);
  System.out.println("x2: " + x2.a + " " + x2.b);
}
}
```

Сторонние эффекты от клонирования иногда поначалу трудно обнаружить. Очень легко решить, что класс безопасен для клонирования, когда на самом деле это не так. Вообще говоря, вы не должны реализовывать `Cloneable` для любого класса без серьезной на то причины.

Class

`Class` инкапсулирует состояние времени выполнения объекта или интерфейса. Объекты типа `Class` создаются автоматически при загрузке класса. Вы не можете явно объявлять объект `Class`. В общем случае вы получаете объект `Class`, вызывая метод `getClass()`, определенный в `Object`. `Class` — это обобщенный тип, который объявлен, как показано ниже:

```
Class Class<T>
```

Здесь `T` — тип представленного класса или интерфейса. Примеры часто используемых методов, определенных в классе `Class`, приведены в табл. 16.15.

Таблица 16.15. Примеры методов, определенные в классе `Class`

Метод	Описание
`static Class<?> forName(String name) throws ClassNotFoundException`	Возвращает объект `Class` по его полному имени.
`static Class<?> forName(String name, boolean how, ClassLoader ldr) throws ClassNotFoundException`	Возвращает объект `Class` по его полному имени. Объект загружается с помощью загрузчика, указанного в *ldr*. Если параметр *how* равен `true`, объект инициализируется, в противном случае — нет.
`<A extends Annotation> A getAnnotation(Class<A> annoType)`	Возвращает объект `Annotation`, содержащий аннотацию, ассоциированную с *annoType*, для вызывающего объекта.
`Annotation[ ] getAnnotations()`	Получает аннотацию, ассоциированную с вызывающим объектом, и сохраняет ее в массиве объектов `Annotation`. Возвращает ссылку на массив.
`Class<?>[ ] getClasses()`	Возвращает объекты `Class` для каждого из общедоступных классов и интерфейсов, которые являются членами вызывающего объекта.
`ClassLoader getClassLoader()`	Возвращает объект `ClassLoader`, который загружал класс или интерфейс, использованный для создания экземпляра вызывающего объекта.

Продолжение табл. 16.15

Метод	Описание
`Constructor<T> getConstructor(Class ... paramTypes) throws NoSuchMethodException, SecurityException`	Возвращает объект `Constructor`, представляющий конструктор вызывающего объекта, который имеет типы параметров, указанные в *paramTypes*.
`Constructor<?>[] getConstructors() throws SecurityException`	Получает объекты `Constructor` для каждого из общедоступных конструкторов вызывающего объекта и сохраняет их в массиве. Возвращает ссылку на этот массив.
`Annotation[] getDeclaredAnnotations()`	Получает объекты `Annotation` для всех аннотаций, объявленных в вызывающем объекте, и сохраняет их в массиве. Возвращает ссылку на этот массив (унаследованные аннотации игнорируются).
`Constructor<?>[] getDeclaredConstructors() throws SecurityException`	Получает объекты `Constructor` для каждого из объявленных конструкторов вызывающего объекта и сохраняет их в массиве. Возвращает ссылку на этот массив (конструкторы суперклассов игнорируются).
`Field[] getDeclaredFields() throws SecurityException`	Получает объекты `Field` для каждого из полей, объявленных классом, и сохраняет их в массиве. Возвращает ссылку на массив. (Унаследованные поля игнорируются.)
`Method[] getDeclaredMethods() throws SecurityException`	Получает объекты `Method` для каждого из методов, объявленных в классе или интерфейсе, и сохраняет их в массиве. Возвращает ссылку на этот массив. (Унаследованные методы игнорируются.)
`Field getField(String fieldName) throws NoSuchMethodException, SecurityException`	Возвращает объект `Field`, который представляет поле, специфицированное в параметре *fieldName* для вызывающего объекта.
`Field[] getFields() throws SecurityException`	Получает объект `Field` для каждого из общедоступных полей вызывающего объекта и сохраняет их в массиве. Возвращает ссылку на этот массив.
`Class<?>[] getInterfaces()`	Когда вызывается объект, этот метод возвращает массив интерфейсов, реализованных типом класса объекта. Когда вызывается интерфейсом, этот метод возвращает массив интерфейсов, которые расширяет данный интерфейс.
`Method getMethod(String methName, Class<?> ... paramTypes) throws NoSuchMethodException, SecurityException`	Возвращает объект `Method`, который представляет метод, специфицированный в *methName* с параметрами типов, указанных в *paramTypes*.
`Method[] getMethods() throws SecurityException`	Получает объект `Method` для каждого общедоступного метода вызывающего объекта и сохраняет их в массиве. Возвращает ссылку на этот массив.
`String getName()`	Возвращает полное имя класса или интерфейса вызывающего объекта.

Окончание табл. 16.15

Метод	Описание
`ProtectionDomain getProtectionDomain()`	Возвращает защитный домен, ассоциированный с вызывающим объектом.
`Class<? super T> getSuperclass()`	Возвращает суперкласс вызывающего объекта. Возвращаемое значение равно `null`, если вызывающий объект относится к типу `Object`.
`boolean isInterface()`	Возвращает `true`, если вызывающий объект является интерфейсом. В противном случае возвращает `false`.
`T newInstance() throws IllegalAccessException, InstantiationException`	Создает новый экземпляр (то есть новый объект), имеющий тот же тип, что и вызывающий объект. Это — эквивалент применения операции `new` с конструктором класса по умолчанию. Возвращается новый объект.
`String toString()`	Возвращает строковое представление вызывающего объекта или интерфейса.

Методы, определенные в `Class`, часто применяются в ситуациях, когда требуется информация об объекте времени выполнения. Как показано в табл. 16.15, вам предоставляются методы, которые позволяют получить дополнительную информацию об определенном классе, такую как его общедоступные конструкторы, поля и методы. Наравне с прочим, это важно для обеспечения функциональности Java Beans, которая обсуждается в настоящей книге далее.

В следующей программе демонстрируется применение `getClass()` (унаследованного от `Object`) и `getSuperclass()` (из `Class`).

```
// Работа с информацией о типе времени 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 Class
  System.out.println("x — объект типа: " +
  clObj.getName());
  clObj = y.getClass(); // Получить ссылку на Class
  System.out.println("y — объект типа: " +
  clObj.getName());
  clObj = clObj.getSuperclass();
  System.out.println("Суперкласс y: " +
  clObj.getName());
}
}
```

Ниже показан вывод этой программы.

```
x — объект типа: X
y — объект типа: Y
Суперкласс y: X
```

ClassLoader

Абстрактный класс `ClassLoader` определяет, как загружаются классы. Ваше приложение может создавать подклассы, расширяющие `ClassLoader`, реализуя его методы. Это позволяет загружать классы другими способами, нежели тот, которым выполняется обычная загрузка в системе времени выполнения Java. Однако обычно вы не должны этого делать.

Math

Класс `Math` содержит все функции с плавающей точкой, которые используются в геометрии и тригонометрии, а также некоторые методы общего назначения. В `Math` определены две константы `double`: `E` (равную приблизительно 2,72) и `PI` (равную примерно 3,14).

Трансцендентные функции

Методы, перечисленные в табл. 16.16, принимают параметр типа `double`, выражающий угол в радианах и возвращающий результат соответствующей трансцендентной функции.

Таблица 16.16. Прямые трансцендентные функции класса `Math`

Метод	Описание
`static double sin(double arg)`	Возвращает синус угла *arg*, переданного в радианах.
`static double cos(double arg)`	Возвращает косинус угла *arg*, переданного в радианах.
`static double tan(double arg)`	Возвращает тангенс угла *arg*, переданного в радианах.

Методы, перечисленные в табл. 16.17, принимают в качестве параметра результат трансцендентной функции и возвращают угол в радианах, который порождает такой результат. Они представляют собой инверсию соответствующих функций из предыдущей таблицы.

Таблица 16.17. Обратные трансцендентные функции класса `Math`

Метод	Описание
`static double asin(double arg)`	Возвращает угол, синус которого равен *arg*.
`static double acos(double arg)`	Возвращает угол, косинус которого равен *arg*.
`static double atan(double arg)`	Возвращает угол, тангенс которого равен *arg*.
`static double atan2(double x, double y)`	Возвращает угол, тангенс которого равен *x/y*.

Методы, перечисленные в табл. 16.18, вычисляют гиперболический синус, косинус и тангенс угла.

Таблица 16.18. Гиперболические функции класса Math

Метод	Описание
`static double sinh(double arg)`	Возвращает гиперболический синус угла *arg*, переданного в радианах.
`static double cosh(double arg)`	Возвращает гиперболический косинус угла *arg*, переданного в радианах.
`static double tanh(double arg)`	Возвращает гиперболический тангенс угла *arg*, переданного в радианах.

Экспоненциальные функции

В классе Math определен набор экспоненциальных методов, которые описаны в табл. 16.19.

Таблица 16.19. Экспоненциальные функции класса Math

Метод	Описание
`static double cbrt(double arg)`	Возвращает кубический корень из *arg*.
`static double exp(double arg)`	Возвращает экспоненту *arg*.
`static double expm1(double arg)`	Возвращает экспоненту *arg*-1.
`static double log(double arg)`	Возвращает натуральный алгоритм *arg*.
`static double log10(double arg)`	Возвращает логарифм по основанию 10 от *arg*.
`static double log1p(double arg)`	Возвращает натуральный алгоритм *arg*+1.
`static double pow(double y, double x)`	Возвращает *y* в степени *x*, например `pow(2.0, 3.0)` вернет `8.0`.
`static double scalb(double arg, int factor)`	Возвращает $arg \times 2^{factor}$. (Добавлен Java SE 6.)
`static float scalb(float arg, int factor)`	Возвращает $arg \times 2^{factor}$. (Добавлен Java SE 6.)
`static double sqrt(double arg)`	Возвращает квадратный корень из *arg*.

Функции округления

В классе Math определены некоторые методы, которые предназначены для выполнения различного рода операций округления. Они перечислены в табл. 16.20. Обратите внимание на два метода `ulp()` в конце таблицы. В данном контексте “ulp” означает “units in the last place” (единицы на последнем месте). Это указывает число единиц между значением и ближайшим большим значением. Это можно использовать для получения точности результата.

Таблица 16.20. Методы округления класса `Math`

Метод	Описание
`static int abs(int arg)`	Возвращает абсолютное значение *arg*.
`static long abs(long arg)`	Возвращает абсолютное значение *arg*.
`static float abs(float arg)`	Возвращает абсолютное значение *arg*.
`static double abs(double arg)`	Возвращает абсолютное значение *arg*.
`static double ceil(double arg)`	Возвращает наименьшее целое число, которое больше *arg*.
`static double floor(double arg)`	Возвращает наибольшее целое число, которое меньше или равно *arg*.
`static int max(int x, int y)`	Возвращает большее из двух чисел *x* и *y*.
`static long max(long x, long y)`	Возвращает большее из двух чисел *x* и *y*.
`static float max(float x, float y)`	Возвращает большее из двух чисел *x* и *y*.
`static double max(double x, double y)`	Возвращает большее из двух чисел *x* и *y*.
`static int min(int x, int y)`	Возвращает меньшее из двух чисел *x* и *y*.
`static long min(long x, long y)`	Возвращает меньшее из двух чисел *x* и *y*.
`static float min(float x, float y)`	Возвращает меньшее из двух чисел *x* и *y*.
`static double min(double x, double y)`	Возвращает меньшее из двух чисел *x* и *y*.
`static double nextAfter(double arg, double toward)`	Начиная со значения *arg*, возвращает следующее значение в направлении *toward*. Если *arg==toward*, то возвращается *toward*. (Добавлен в Java SE 6.)
`static float nextAfter(float arg, double toward)`	Начиная со значения *arg*, возвращает следующее значение в направлении *toward*. Если *arg==toward*, то возвращается *toward*. (Добавлен в Java SE 6.)
`static double nextUp(double arg)`	Возвращает следующее значение в положительном направлении от *arg*. (Добавлен в Java SE 6.)
`static float nextUp(float arg)`	Возвращает следующее значение в положительном направлении от *arg*. (Добавлен в Java SE 6.)
`static double rint(double arg)`	Возвращает ближайшее целое к *arg*.
`static int round(float arg)`	Возвращает *arg*, округленное вверх до ближайшего `int`.
`static long round(double arg)`	Возвращает *arg*, округленное вверх до ближайшего `long`.
`static float ulp(float arg)`	Возвращает ulp для *arg*.
`static double ulp(double arg)`	Возвращает ulp для *arg*.

Прочие методы Math

В дополнение к методам, приведенным в таблицах выше, в Math определено еще несколько методов, которые перечислены в табл. 16.21.

Таблица 16.21. Прочие методы класса Math

Метод	Описание
static double copySign(double *arg*, double *signarg*)	Возвращает *arg* с тем же знаком, что у *signarg*. (Добавлен в Java SE 6.)
static float copySign(float *arg*, float *signarg*)	Возвращает *arg* с тем же знаком, что у *signarg*. (Добавлен в Java SE 6.)
static int getExponent(double *arg*)	Возвращает экспоненту по основанию 2, используемую для двоичного представления *arg*. (Добавлен в Java SE 6.)
static int getExponent(float *arg*)	Возвращает экспоненту по основанию 2, используемую для двоичного представления *arg*. (Добавлен в Java SE 6.)
static double IEEEremainder(double *dividend*, double *divisor*)	Возвращает остаток от деления *divident/divisor*.
static hypot(double *side1*, double *side2*)	Возвращает длину гипотенузы прямоугольного треугольника по длине двух противоположных сторон.
static double random()	Возвращает псевдослучайное число между 0 и 1.
static float signum(double *arg*)	Определяет знак значения. Возвращает 0, если *arg* равен 0, 1 — если *arg* больше 0, и –1 — если *arg* меньше 0.
static float signum(float *arg*)	Определяет знак значения. Возвращает 0, если *arg* равен 0, 1 — если *arg* больше 0, и –1 — если *arg* меньше 0.
static double toDegrees(double *angle*)	Преобразует радианы в градусы. Переданный в *angle* угол должен быть указан в радианах. Возвращается результат в градусах.
static double toRadians(double *angle*)	Преобразует градусы в радианы. Переданный в *angle* угол должен быть указан в градусах. Возвращается результат в радианах.

В следующей программе демонстрируется использование toRadians() и toDegrees().

```
// Демонстрация применения 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.");
}
}
```

Вывод этой программы показан ниже:

```
120.0 градусов равно 2.0943951023931953 радиан.
1.312 радиан равно 75.17206272116401 радиан.
```

StrictMath

Класс `StrictMath` определяет полный набор математических методов, параллельный доступному в `Math`. Отличие в том, что версия `StrictMath` гарантирует точность, идентичную на всех реализациях Java, в то время как `Math` имеет большую свободу в целях достижения производительности.

Compiler

Класс `Compiler` поддерживает создание окружений Java, в которых байт-код Java компилируется в исполняемый код вместо интерпретируемого. При нормальном программировании не используется.

Thread, ThreadGroup и Runnable

Интерфейс `Runnable`, а также классы `Thread` и `ThreadGroup` поддерживают многопоточное программирование. Они рассматриваются ниже.

Интерфейс Runnable

Интерфейс `Runnable` должен быть реализован любым классом, который инициирует отдельный поток выполнения. `Runnable` только определяет один абстрактный метод по имени `run()`, который является "точкой входа" потока. Он определен следующим образом:

```
void run()
```

Потоки, которые вы создаете, должны реализовывать этот метод.

Thread

`Thread` создает новый поток выполнения. Он определяет следующие часто используемые конструкторы:

```
Thread()
Thread(Runnable threadOb)
Thread(Runnable threadOb, String threadName)
Thread(String threadName)
Thread(ThreadGroup groupOb, Runnable threadOb)
Thread(ThreadGroup groupOb, Runnable threadOb, String threadName)
Thread(ThreadGroup groupOb, String threadName)
```

threadOb — это экземпляр класса, реализующего интерфейс `Runnable` и определяющий, где начинается выполнение потока. Имя потока задается в *threadName*. Когда имя не указано, оно создается виртуальной машиной Java. *groupOb* специфицирует группу потоков, к которой новый поток будет относиться. Когда никакая группа не указана, новый поток относится к той же группе, что и родительский.

В классе Thread определены следующие константы:

```
MAX_PRIORITY
MIN_PRIORITY
NORM_PRIORITY
```

Как и можно было ожидать, эти константы определяют максимальный, минимальный и нормальный уровень приоритета потоков. Методы, определенные в классе Thread, перечислены в табл. 16.22. В ранних версиях Java Thread также включал методы stop(), suspend() и resume(). Однако, как уже объяснялось в главе 11, они являются нежелательными, поскольку нестабильны. Также нежелательными является метод countStackFrames(), поскольку он вызывает suspend() и destroy(), и может привести к взаимной блокировке.

Таблица 16.22. Методы, определенные в классе Thread

Метод	Описание
`static int activeCount()`	Возвращает количество потоков в группе, к которой относится данный поток.
`void checkAccess()`	Принуждает диспетчер безопасности проверять, что текущий поток может иметь доступ и/или изменять поток, в котором checkAccess() был вызван.
`static Thread currentThread()`	Возвращает объект Thread, который инкапсулирует поток, вызвавший этот метод.
`static void dumpStack()`	Отображает стек вызовов потока.
`static int enumerate(Thread threads[])`	Помещает копию объектов Thread из текущей группы потоков в массив *threads*. Возвращается количество потоков.
`static Map<Thread, StackTraceElement[]> getAllStackTraces()`	Возвращает Map, который содержит трассировку стека для всех активных потоков в группе. Каждый элемент в Map состоит из ключа, который является объектом Thread, и его значения, которое представляет собой массив StackTraceElement.
`ClassLoader getContextClassLoader()`	Возвращает загрузчик класса, используемый для загрузки классов и ресурсов для текущего потока.
`static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler()`	Возвращает обработчик неперехваченных исключений по умолчанию.
`Thread.UncaughtExceptionHandler getUncaughtExceptionHandler()`	Возвращает обработчик неперехваченных исключений данного потока.
`long getID()`	Возвращает идентификатор вызывающего потока.
`final String getName()`	Возвращает имя потока.
`final int getPriority()`	Возвращает установки приоритета потока.
`StackTraceElement[] getStackTrace()`	Возвращает массив, содержащий трассировку стека вызывающего потока.
`Thread.State getState()`	Возвращает состояние вызывающего потока.
`final ThreadGroup getThreadGroup()`	Возвращает объект ThreadGroup, членом которого является текущий поток.

Окончание табл. 16.22

Метод	Описание
`static boolean holdsLock(Object ob)`	Возвращает `true`, если вызывающий поток владеет блокировкой на *ob*. В противном случае возвращает `false`.
`void interrupt()`	Прерывает поток.
`static boolean interrupted()`	Возвращает `true`, если вызывающий поток запланирован на прерывание. В противном случае возвращает `false`.
`final Boolean isAlive()`	Возвращает `true`, если вызывающий поток еще активен. В противном случае возвращает `false`.
`final Boolean isDaemon()`	Возвращает `true`, если вызывающий поток является потоком-демоном (одним из потоков исполняющей системы Java). В противном случае возвращает `false`.
`boolean isInterrupted()`	Возвращает `true`, если вызывающий поток прерван. В противном случае возвращает `false`.
`final void join() throws InterruptedException`	Ожидает завершения потока.
`final void join(long milliseconds) throws InterruptedException`	Ожидает до заданного числа миллисекунд завершения потока, в котором метод вызван.
`final void join(long milliseconds, int nanoseconds) throws InterruptedException`	Ожидает до заданного числа миллисекунд плюс наносекунд завершения потока, в котором метод вызван.
`void run()`	Начинает выполнение потока.
`void setContextClassLoader(ClassLoader cl)`	Устанавливает загрузчик класса, который будет использован вызывающим объектом .
`final void setDaemon(boolean state)`	Помечает поток как поток-демон.
`static void setDefaultUncaughtExceptionHandler( Thread.UncaughtExceptionHandler e)`	Устанавливает обработчик неперехваченных прерываний по умолчанию.
`void setUncaughtExceptionHandler( Thread.UncaughtExceptionHandler e)`	Устанавливает обработчик неперехваченных прерываний по умолчанию для вызывающего потока.
`final void setName(String threadName)`	Устанавливает имя потока в *threadName*.
`final void setPriority(int priority)`	Устанавливает приоритет потока в *priority*.
`static void sleep(long milliseconds) throws InterruptedException`	Прерывает выполнение потока на заданное число миллисекунд.
`static void sleep(long milliseconds, int nanoseconds) throws InterruptedException`	Прерывает выполнение потока на заданное число миллисекунд плюс наносекунд.
`void start()`	Запускает выполнение потока.
`String toString()`	Возвращает строковое представление потока.
`static void yield()`	Вызывающий поток уступает ресурс центрального процессора другому потоку.

ThreadGroup

ThreadGroup создает группу потоков. В классе определены следующие два конструктора:

```
ThreadGroup(String groupName)
ThreadGroup(ThreadGroup parentOb, String groupName)
```

Для обеих форм *groupName* указывает имя группы потоков. Первая версия создает новую группу, родителем которой будет текущий поток. Во второй форме родитель группы задается параметром *parentOb*. Не устаревшие методы, определенные в ThreadGroup, перечислены в табл. 16.23.

Таблица 16.23. Методы, определенные в ThreadGroup

Метод	Описание
`int activeCount()`	Возвращает количество потоков в группе, плюс любые группы, для которых этот поток является родителем.
`int activeGroupCount()`	Возвращает количество групп, для которых вызывающий поток является родителем.
`final void checkAccess()`	Вынуждает диспетчер безопасности проверять, может ли вызывающий поток иметь доступ и/или изменять группу, для которой вызван метод `checkAccess()`.
`final void destroy()`	Уничтожает группу потоков (и ее дочерние группы), для которой метод вызывался.
`int enumerate(Thread group[])`	Потоки, которые включены в вызывающую группу, помещаются в массив *group*.
`int enumerate(Thread group[], boolean all)`	Потоки, которые включены в вызывающую группу, помещаются в массив *group*. Если *all* равно `true`, то в *group* помещаются также все подгруппы потока.
`int enumerate(ThreadGroup group[])`	Подгруппы вызывающей группы помещаются в массив *group*.
`int enumerate(ThreadGroup group[], boolean all)`	Подгруппы вызывающей группы помещаются в массив *group*. Если *all* равно `true`, то все подгруппы всех подгрупп также помещаются в *group*.
`final int getMaxPriority()`	Возвращает максимальный приоритет, установленный для группы.
`final String getName()`	Возвращает имя группы.
`final ThreadGroup getParent()`	Возвращает `null`, если вызывающий объект ThreadGroup не имеет родителя. В противном случае возвращается родитель вызывающего объекта.
`final void interrupt()`	Вызывает метод `interrupt()` для всех потоков в группе.
`final boolean isDaemon()`	Возвращает `true`, если текущая группа является группой-демоном. В противном случае возвращается `false`.
`boolean isDestroyed()`	Возвращает `true`, если текущая группа разрушена. В противном случае возвращается `false`.
`void list()`	Отображает информацию о группе.

Окончание табл. 16.23

Метод	Описание
`final boolean parentOf(ThreadGroup group)`	Возвращает `true`, если вызывающий поток является родителем группы (или самой группой). В противном случае возвращает `false`.
`final void setDaemon(Boolean isDaemon)`	Если *isDaemon* равно `true`, то вызывающая группа помечается, как группа-демон.
`final void setMaxPriority(int priority)`	Устанавливает максимальный приоритет для вызывающей группы равным *priority*.
`String toString()`	Возвращает строковое представление группы.
`void uncaughtException(Thread thread, Throwable e)`	Метод вызывается, когда возникает необработанное исключение.

Группы потоков предоставляют удобный способ управления группами потоков как одним целым. В частности, это бывает важно в ситуациях, когда вы хотите приостановить или продолжить выполнение множества взаимосвязанных потоков. Например, предположим, что имеется программа, в которой один набор потоков используется для печати документов, другой — для отображения документа на экране, а еще один сохраняет документ в дисковом файле. Если печать прерывается, потребуется прервать все потоки, имеющие отношение к печати. Сказанное иллюстрируется в следующей программе, которая создает две группы процессов, по два процесса в каждой.

```
// Демонстрация применения групп потоков.
class NewThread extends Thread {
boolean suspendFlag;
NewThread(String threadname, ThreadGroup tgOb) {
  super(tgOb, threadname);
  System.out.println("Новый поток: " + this);
  suspendFlag = false;
  start(); // Запуск потока
}
// Точка входа потока.
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.");
}
void mysuspend() {
  suspendFlag = true;
}
```

```
  synchronized void myresume() {
    suspendFlag = false;
    notify();
  }
  }

  class ThreadGroupDemo {
  public static void main(String args[]) {
      ThreadGroup groupA = new ThreadGroup("Группа A");
      ThreadGroup groupB = new ThreadGroup("Группа B");
      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 list():");
      groupA.list();
      groupB.list();
      System.out.println();
      System.out.println("Прерывается группа A");
      Thread tga[] = new Thread[groupA.activeCount()];
      groupA.enumerate(tga);              // получить потоки группы

      for(int i = 0; i < tga.length; i++) {
        ((NewThread)tga[i]).mysuspend();  // приостановить каждый поток
      }

      try {
        Thread.sleep(4000);
      } catch (InterruptedException e) {
        System.out.println("Главный поток прерван.");
      }

      System.out.println("Возобновление Group A");

      for(int i = 0; i < tga.length; i++) {
        ((NewThread)tga[i]).myresume();   // возобновить все потоки в группе
      }

      // Ожидать завершения потоков
      try {
        System.out.println("Ожидание останова потоков.");
        ob1.join();
        ob2.join();
        ob3.join();
        ob4.join();
      } catch (Exception e) {
        System.out.println("Исключение в основном потоке");
      }
      System.out.println("Основной поток завершен.");
  }
  }
```

Пример вывода этой программы показан ниже (точный вывод, который вы увидите, может несколько отличаться).

```
Новый поток: Thread[Один,5,Группа A]
Новый поток: Thread[Два,5,Группа A]
Новый поток: Thread[Три,5,Группа B]
Новый поток: Thread[Четыре,5,Группа B]
```

```
Вывод из list():
java.lang.ThreadGroup[name=Group A,maxpri=10]
Thread[Один,5,Группа A]
Thread[Два,5,Группа A]
java.lang.ThreadGroup[name=Group B,maxpri=10]
Thread[Три,5,Группа B]
Thread[Четыре,5,Группа B]
Прерывается группа A
Три: 5
Четыре: 5
Три: 4
Четыре: 4
Три: 3
Четыре: 3
Три: 2
Четыре: 2
Возобновление группы A
Ожидание останова потоков.
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.
Основной поток завершен.
```

Обратите внимание, что внутри программы группа A приостанавливается на четыре секунды. Как подтверждает вывод программы, это приостанавливает выполнение потоков “Один” и “Два”, но потоки “Три” и “Четыре” продолжают выполняться. По истечении четырех секунд выполнение потоков “Один” и “Два” возобновляется. Следует отметить также, как останавливается и возобновляет выполнение группа A. Во-первых, потоки из группы A извлекаются вызовом метода `enumerate()` для этой группы. Затем каждый поток приостанавливается в процессе итерации по результирующему массиву. Чтобы продолжить выполнение потоков в A, опять осуществляется проход по списку потоков, и каждый поток запускается для продолжения работы. И последний момент: в этом примере применяется рекомендованный подход для приостановки и возобновления потоков. Это не касается устаревших (не рекомендованных) методов `suspend()` и `resume()`.

ThreadLocal и InheritableThreadLocal

В пакете `java.lang` определены два дополнительных класса, имеющих отношение к потокам:

- ThreadLocal — используется для создания локальных переменных потоков. Каждый поток будет иметь свою собственную копию локальной переменной потока.
- InheritableThreadLocal — создает локальные переменные потоков, которые могут наследоваться.

Package

Package инкапсулирует данные о версии, ассоциированные с пакетом. Информация о версии пакета приобретает особую ценность из-за профилирования пакетов, а также потому, что Java-программы могут нуждаться в знании о том, какая версия пакета доступна. Методы, определенные в классе Package, перечислены в табл. 16.24. В следующей программе демонстрируется использование класса Package, отображающего список пакетов, с которыми имеет дело программа в данный момент.

```
// Демонстрация применения 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 16.24. Методы, определенные в Package

Метод	Описание
<A extends Annotation> A getAnnotation(Class<A> *annoType*)	Возвращает объект Annotation, который содержит аннотацию, ассоциированную с *annoType*, для вызывающего объекта.
Annotation[ ] getAnnotations()	Возвращает все аннотации, ассоциированные с вызывающим объектом в массиве объектов Annotation. Возвращает ссылку на этот массив.
Annotation[ ] getDeclaredAnnotations()	Возвращает объект Annotation для всех аннотаций, объявленных в вызывающем объекте (унаследованные аннотации игнорируются).
String getImplementationTitle()	Возвращает заголовок вызывающего пакета.
String getImplementationVendor()	Возвращает имя реализатора вызывающего пакета.
String getImplementationVersion()	Возвращает номер версии вызывающего пакета.
String getName()	Возвращает имя вызывающего пакета.
static Package getPackage(String *pkgName*)	Возвращает объект Package по имени, указанном в *pkgName*.

Окончание табл. 16.24

Метод	Описание
`static Package[] getPackages()`	Возвращает все пакеты, о которых осведомлена текущая выполняющаяся программа.
`String getSpecificationTitle()`	Возвращает титул спецификации вызывающего пакета.
`String getSpecificationVendor()`	Возвращает имя владельца из спецификации вызывающего пакета.
`String getSpecificationVersion()`	Возвращает номер версии спецификации вызывающего пакета.
`int hashCode()`	Возвращает хеш-код вызывающего пакета.
`boolean isAnnotationPresent(` `Class<? extends Annotation> anno)`	Возвращает `true`, если аннотация, описанная *anno*, ассоциируется с вызывающим объектом. В противном случае возвращает `false`.
`boolean isCompatibleWith(String verNum)` `throws NumberFormatException`	Возвращает `true`, если *verNum* меньше или равно номеру версии вызывающего пакета.
`boolean isSealed()`	Возвращает `true`, если вызывающий пакет "запечатан" (sealed). В противном случае возвращает `false`.
`boolean isSealed(URL url)`	Возвращает `true`, если вызывающий пакет "запечатан" (sealed) относительно *url*. В противном случае возвращает `false`.
`String toString()`	Возвращает строковый эквивалент вызывающего пакета.

RuntimePermission

Класс `RuntimePermission` относится к механизму безопасности Java и подробно здесь не рассматривается.

Throwable

Класс `Throwable` поддерживает систему обработки исключений Java и представляет собой класс, от которого происходят все классы исключений. Этот класс рассматривался в главе 10.

SecurityManager

`SecurityManager` — это класс, от которого можно наследовать подклассы для создания диспетчера безопасности. В общем случае, как правило, нет необходимости реализовывать свой собственный диспетчер безопасности. Если вы все же решите это делать, проконсультируйтесь с документацией, поставляемой с системой разработки Java.

StackTraceElement

Класс StackTraceElement описывает единственный *стековый фрейм*, который представляет собой индивидуальный элемент трассировки стека при возникновении исключения. Каждый стековый фрейм представляет *точку выполнения*, которая включает такие вещи, как имя класса, имя метода, имя файла и номер строки исходного кода. Массив элементов StackTraceElement возвращается методом getStackTrace() класса Throwable.

StackTraceElement имеет один конструктор:

```
StackTraceElement(String className, String methName, String fileName, int line)
```

Здесь имя класса указано в *className*, имя метода — в *methName*, имя файла — в *fileName*, а номер строки передается в *line*. Если нет допустимого номера строки, используйте отрицательное значение *line*. Более того, значение –2 для *line* означает, что этот фрейм ссылается на родной (native) метод.

Методы, поддерживаемые StackTraceElement, перечислены в таблице 16.25. Они дают доступ программе к ее трассировке стека.

Таблица 16.25. Методы, определенные в StackTraceElement

Метод	Описание
boolean equals(Object *ob*)	Возвращает true, если вызывающий StackTraceElement тот же самый, что передан в *ob*. В противном случае возвращает false.
String getClassName()	Возвращает имя класса точки выполнения, описанной вызывающим объектом StackTraceElement.
String getFileName()	Возвращает имя файла точки выполнения, описанной вызывающим объектом StackTraceElement.
int getLineNumber()	Возвращает номер строки исходного кода точки выполнения, описанной вызывающим объектом StackTraceElement. В некоторых ситуациях номер строки не будет доступен, тогда возвращается отрицательное значение.
String getMethodName()	Возвращает имя метода в точке выполнения, описанной вызывающим объектом StackTraceElement.
int hashCode()	Возвращает хеш-код вызывающего StackTraceElement.
boolean isNativeMethod()	Возвращает true, если вызывающий объект StackTraceElement описывает родной метод. В противном случае возвращает false.
String toString()	Возвращает строковый эквивалент вызывающей последовательности.

Enum

Как было описано в главе 12, недавно в язык Java были добавлены перечисления (enumeration). (Вспомните, что перечисление создается ключевым словом enum.) Все перечисления автоматически наследуются от Enum. Enum — это обобщенный класс, объявленный следующим образом:

```
class Enum<E extends Enum<E>>
```

Здесь E обозначает перечислимый тип. Enum не имеет общедоступных конструкторов.

В Enum определено несколько методов, доступных для использования всеми перечислителями. Они описаны в таблице 16.26.

Таблица 16.26. Методы, определенные в Enum

Метод	Описание
`protected final Object clone() throws CloneNotSupportedException`	Вызов этого метода инициирует исключение CloneNotSupportedException. Это предотвращает клонирование перечислений.
`final int compareTo(E e)`	Сравнивает порядковое значение двух констант одного перечисления. Возвращает отрицательное значение, если вызывающая константа имеет порядковое значение меньше *e*, ноль — если совпадающее и больше нуля — если больше *e*.
`final boolean equals(Object obj)`	Возвращает true, если *obj* 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 неизменное имя вызывающей константы.
`final int ordinal()`	Возвращает значение, которое указывает позицию перечислимой константы в списке констант.
`String toString()`	Возвращает имя вызывающей константы. Это имя может отличаться от того, которое использовалось при объявлении перечисления.
`static <T extends Enum<T>> T valueOf(Class<T> e-type, String name)`	Возвращает константу, ассоциированную с *name*, в типе перечисления, специфицированном *e-type*.

Интерфейс CharSequence

Интерфейс CharSequence определяет методы, которые предоставляют доступ только по чтению к последовательности символов. Методы описаны в таблице 16.27. Этот интерфейс реализован в классах String, StringBuffer и StringBuilder. Кроме того, его реализует CharBuffer, содержащийся в пакете java.nio (рассматривается далее в настоящей главе).

Таблица 16.27. Методы, определенные в CharSequence

Метод	Описание
`char charAt(int idx)`	Возвращает символ, находящийся в позиции, указанной индексом *idx*.
`int length()`	Возвращает количество символов в вызывающей последовательности.
`CharSequence subSequence(int startIdx, int stopIdx)`	Возвращает подмножество вызывающей последовательности, начиная с *startIndex* и заканчивая *stopIndex*-1.
`String toString()`	Возвращает String-эквивалент вызывающей последовательности.

Интерфейс Comparable

Объекты классов, реализующих интерфейс Comparable, могут быть упорядочены. Другими словами, классы, реализующие Comparable, содержат объекты, которые можно сравнивать некоторым осмысленным образом. Comparable — обобщенный интерфейс и объявлен следующим образом:

```
interface Comparable<T>
```

Здесь T представляет собой тип сравниваемых объектов.

Интерфейс Comparable объявляет один метод, который используется для определения того, что в Java называется *натуральным порядком* экземпляров класса. Сигнатура метода показана ниже:

```
int compareTo(T obj)
```

Этот метод сравнивает вызывающий объект с *obj*. Возвращает 0, если значения эквивалентны. Отрицательное значение возвращается, если вызывающий объект имеет меньшее значение. В противном случае возвращается положительное значение.

Этот интерфейс реализован в нескольких классах, уже рассмотренных в книге. В частности, классы Byte, Character, Double, Float, Long, Short, String и Integer определяют метод compareTo(). В дополнение, как показано в следующей главе, объекты, которые реализуют этот интерфейс, могут быть использованы в разных коллекциях.

Интерфейс Appendable

Объекты классов, реализующих интерфейс Appendable, позволяют добавлять к себе символы или символьные последовательности. Appendable определяет следующие три метода:

```
Appendable append(char ch) throws IOException
Appendable append(CharSequence chars) throws IOException
Appendable append(CharSequence chars, int begin, int end) throws IOException
```

В первой форме символ *ch* добавляется к вызывающему объекту. Во второй форме к вызывающему объекту добавляется последовательность символов *chars*. Третья форма позволяет указать фрагмент (заданную через *start* и *end*) последовательности *chars*. Во всех случаях возвращается ссылка на вызывающий объект.

Интерфейс Iterable

Интерфейс Iterable должен быть реализован всеми классами, чьи объекты будут использованы в версии “for-each” цикла for. Другими словами, для того, чтобы объект использовался внутри цикла “for-each”, его класс должен реализовывать интерфейс цикла Iterable. Iterable — это обобщенный интерфейс, имеющий следующее объявление:

```
interface Iterable<T>
```

Здесь T представляет собой тип объектов, по которым выполняется итерация. Он определяет один метод, iterator(), показанный ниже:

```
Iterator<T> iterator( )
```

Метод возвращает итератор, связанный с элементами, содержащимися в вызывающем объекте.

На заметку! *Итераторы рассматриваются в главе 17.*

Интерфейс `Readable`

Интерфейс `Readable` указывает, что объект может быть использован в качестве источника символов. В этом интерфейсе определен один метод — `read()`, представленный ниже:

```
int read(CharBuffer buf) throws IOException
```

Этот метод читает символы в *buf*. Возвращает количество прочитанных символов, или –1 в случае достижения символа конца файла (EOF).

Вложенные пакеты `java.lang`

В Java определено несколько вложенных пакетов:

- `java.lang.annotation`
- `java.lang.instrument`
- `java.lang.management`
- `java.lang.ref`
- `java.lang.reflect`

Каждый из них кратко описан ниже

`java.lang.annotation`

Средства аннотирования Java поддерживаются с помощью `java.lang.annotation`. В этом пакете определен интерфейс `Annotation`, а также перечисления `ElementType` и `RetentionPolicy`. (`Annotation` описан в главе 12.)

`java.lang.instrument`

Пакет `java.lang.instrument` определяет средства, которые могут быть использованы для добавления инструментария для разных аспектов выполнения программ. Он определяет интерфейсы `Instrumentation` и `ClassFileTransformer`, а также класс `ClassDefinition`.

`java.lang.management`

Пакет `java.lang.management` предоставляет поддержку управления виртуальной машиной Java и исполняющим окружением. Используя средства `java.lang.management`, вы можете просматривать и управлять различными аспектами выполнения программы.

java.lang.ref

Ранее уже упоминалось, что средства сборки мусора в Java автоматически определяют, когда не остается ссылок на объект. Затем предполагается, что этот объект более не нужен и занятую им память можно утилизировать. Классы пакета `java.lang.ref` предлагают более тонкие возможности управления процессом сборки мусора. Например, представим, что ваша программа создает множество объектов, которые вы хотели бы использовать в более позднее время. Вы можете продолжать удерживать ссылки на них, но это может потребовать много памяти.

Взамен вы можете создавать "мягкие" ссылки на эти объекты. Объект, который "мягко доступен", может быть утилизирован сборщиком мусора, если объем свободной памяти станет слишком мал. В этом случае сборщик мусора устанавливает "мягкие" ссылки на эти объекты равными `null`. В противном случае сборщик мусора сохраняет объект для позднего повторного использования.

Программист имеет возможность определить, когда "мягко ссылаемые" объекты утилизированы. Если это случилось, они могут быть инициированы повторно. В противном случае сборщик мусора сохраняет их для возможного использования в будущем. Вы можете создать также "слабые" (weak) или "фантомные" ссылки на объекты. Дискуссия об этих и других средствах пакета `java.lang.ref` выходит за рамки настоящей книги.

java.lang.reflect

Рефлексия (reflection) — это свойство программы анализировать себя саму. Пакет `java.lang.reflect` предоставляет возможности получать информацию о полях, конструкторах, методах и модификаторах класса. Помимо других причин, эта информация может понадобиться для создания программных инструментов, которые позволяют работать с компонентами Java Beans. Инструмент использует рефлексию для динамического определения характеристик компонента. Рефлексия была представлена в главе 12 и также рассматривается в главе 27.

`java.lang.reflect` определяет несколько классов, включая `Method`, `Field` и `Constructor`. Также этот пакет содержит несколько интерфейсов, в числе которых `AnnotatedElement`, `Member` и `Type`. В дополнение пакет `java.lang.reflect` включает класс `Array`, позволяющий динамически создавать и оперировать массивами.

17 ГЛАВА

java.util: каркас коллекций

Настоящая глава посвящена классам и интерфейсам, определенным в пакете java.util. Этот важный пакет содержит большой ассортимент классов и интерфейсов, поддерживающих широкий диапазон функциональности. Например, java.util включает классы, генерирующие псевдослучайные числа, управляющие датами и временем, просмотром событий, манипулирующие наборами бит, разбирающие строки и управляющие форматированными данными. Пакет java.util также включает одну из наиболее мощных подсистем Java — коллекции. Каркас коллекций — это сложная иерархия интерфейсов и классов, предоставляющих изящную технологию управления группами объектов. Она заслуживает пристального внимания всех программистов.

Поскольку java.util содержит широкий диапазон функциональности, этот пакет достаточно объемен. Ниже дается список его классов.

AbstractCollection
AbstractList
AbstractMap
AbstractQueue
AbstractSequentialList
AbstractSet
ArrayDeque (Добавлен в Java SE 6)
ArrayList
Arrays
BitSet
Calendar
Collections
Currency
Date
Dictionary
EnumMap
EnumSet
EventListenerProxy
EventObject
FormattableFlags
Formatter
GregorianCalendar
HashMap
HashSet
Hashtable
IdentityHashMap
LinkedHashMap
LinkedHashSet
LinkedList
ListResourceBundle
Locale
Observable
PriorityQueue
Properties
PropertyPermission
PropertyResourceBundle
Random
ResourceBundle
Scanner
ServiceLoader (Добавлен в Java SE 6)
SimpleTimeZone
Stack
StringTokenizer
Timer
TimerTask
TimeZone
TreeMap
TreeSet
UUID
Vector
WeakHashMap

В java.util определены следующие интерфейсы:

Collection	List	Queue
Comparator	ListIterator	RandomAccess
Deque (Добавлен в Java SE 6)	Map	Set
Enumeration	Map.Entry	SortedMap
EventListener	NavigableMap (Добавлен в Java SE 6)	SortedSet
Formattable	NavigableSet (Добавлен в Java SE 6)	
Iterator	Observer	

Из-за большого размера пакета java.util его описание разбито на две главы. Эта глава посвящена той части java.util, которая является частью каркаса коллекций (Collections Framework). В главе 18 рассматриваются остальные классы и интерфейсы этого пакета.

Обзор коллекций

Каркас коллекций Java стандартизирует способы управления группами объектов для ваших программ. Коллекции не были частью исходной версии Java, но были добавлены в J2SE 1.2. До появления каркаса коллекций для хранения и манипулирования группами объектов Java предлагал такие специальные классы, как Dictionary, Vector, Stack и Properties. Хотя эти классы были достаточно удобны, им недоставало централизованной, универсальной идеи. Так, например, способ применения Vector отличался от способа использования Properties. Кроме того, этот ранний, специализированный подход не был спроектирован в расчете на дальнейшее расширение и адаптацию. Коллекции стали решением этой и ряда других проблем.

Каркас коллекций был разработан для достижения нескольких целей. Во-первых, он должен был быть высокопроизводительным. Реализация основных коллекций (динамических массивов, связных списков, деревьев и хеш-таблиц) отличается высокой эффективностью. Очень редко вам понадобится (если вообще понадобится) кодировать один из таких "механизмов данных" вручную. Во-вторых, эта система должна была позволить разным типам коллекций работать в единой манере и с высокой степенью взаимодействия. В-третьих, расширение и/или адаптация коллекций должна была быть простой. И, наконец, весь каркас коллекций построен на едином наборе стандартных интерфейсов. Некоторые стандартные реализации (такие как LinkedList, HashSet и TreeSet) этих интерфейсов вы можете использовать "как есть". Вы также можете реализовать свои собственные коллекции, если хотите. Для вашего удобства предусмотрены различные реализации специального назначения, а также частичные реализации, которые облегчают создание ваших собственных коллекций. Наконец, были добавлены механизмы интеграции стандартных массивов в систему коллекций.

Алгоритмы — это другая важная часть каркаса коллекций. Алгоритмы оперируют коллекциями и определены как статические методы класса Collections. Таким образом, они доступны всем коллекциям. Каждый класс коллекции не нуждается в реализации своей собственной версии. Алгоритмы представляют стандартные способы манипулирования коллекциями.

Другая сущность, плотно ассоциированная с системой коллекций — это интерфейс итератора Iterator. *Итератор* предоставляет общий, стандартизированный способ доступа к элементам коллекций по одному. То есть итератор предлагает способ *перечисления содержимого коллекций*. Поскольку каждая коллекция реализует интерфейс Iterator,

элементы любого класса коллекций могут быть доступны через методы, определенные в Iterator. Таким образом, код, проходящий в цикле набор (set), с минимальными изменениями можно применить, например, к списку (list).

В дополнение к коллекциям в каркасе определено также несколько интерфейсов и классов карт (map). *Карта* хранит пары "ключ-значение". Хотя карты являются частью системы коллекций, они, строго говоря, коллекциями не являются. Тем не менее, вы можете получить доступ к картам в виде коллекций. Такое представление содержит элементы карты, помещенные в коллекцию. Таким образом, вы, при желании, можете обрабатывать содержимое карты как коллекцию.

Каркас коллекций был модифицирован для некоторых классов, изначально определенных в пакете java.util таким образом, что они также могут быть интегрированы в новую систему. Важно понимать: несмотря на то, что добавление коллекций изменило архитектуру многих оригинальных служебных классов, это не отменило ни одного из них. Коллекции просто предлагают лучший способ выполнения некоторых вещей.

На заметку! *Если вы знакомы с языком C++, то, возможно, обнаружите, что вам поможет сходство между технологией коллекций Java и идеологией стандартной библиотеки шаблонов (Standard Template Library — STL), определенной в C++. То, что в C++ называется контейнером, в Java называется коллекцией. Однако есть существенные отличия между системой коллекций и STL. Поэтому важно не делать поспешных выводов.*

Последние изменения в коллекциях

В последнее время каркас коллекций претерпел существенные изменения, значительно повышающие ее мощность и расширяющие направления ее использования. Изменения вызваны добавлением возможности обобщенных (generic) определений, автоматической упаковки-распаковки, а также стиля "for-each" цикла for. Несмотря на то что мы будем повторно возвращаться к этим темам на протяжении всей настоящей главы, все же краткий обзор представим сразу.

Обобщенные определения фундаментально изменяют систему коллекций

Добавление обобщенных определений — существенное изменение в каркасе коллекций, поскольку для этого он полностью был перепроектирован. Все коллекции теперь обобщенные, и многие из методов, оперирующих коллекциями, также принимают обобщенные параметры. Простое добавление этого свойства коснулось всех частей каркаса коллекций.

Обобщенные определения — это то свойство, которого не хватало коллекциям: безопасность типов. Раньше все коллекции хранили ссылки на Object, а это означало, что любая коллекция могла хранить объекты любого типа. Таким образом, было возможно непреднамеренно сохранить несовместимые типы в одной коллекции. Это могло привести к ошибкам несовместимости типов во время выполнения. С обобщенными определениями можно явно указать тип сохраняемых данных и таких ошибок времени выполнения можно избежать.

Несмотря на то что добавление обобщенных определений изменило объявления большинства классов и интерфейсов, а также некоторых их методов, в общем, каркас коллекций по-прежнему работает так же, как и ранее. Однако если вы знакомы со старыми

версиями каркаса коллекций, новый синтаксис может показаться несколько пугающим. Не беспокойтесь, со временем обобщенный синтаксис станет привычным.

Еще один момент: чтобы получить выгоду от этого нововведения в коллекциях, старый код должен быть переписан. Это также важно, поскольку в противном случае старый код при компиляции современным компилятором Java будет генерировать предупреждения. Чтобы исключить эти сообщения, вам придется добавить информацию о типе везде, где у вас встречается код, работающий с коллекциями.

Средства автоматической упаковки используют примитивные типы

Сохранение примитивных типов в коллекциях облегчает автоматическая упаковка. Как вы увидите, коллекции могут сохранять только ссылки, но не примитивные значения. Раньше, если вы хотели сохранять в коллекции значение примитивного типа вроде `int`, то должны были вручную упаковать его в объект-оболочку типа. Когда значение извлекалось, нужно было его вручную распаковать (применяя явное приведение) в корректный примитивный тип. Благодаря автоматической упаковке-распаковке, Java теперь может делать это автоматически, когда необходимо сохранять или извлекать примитивные типы. Нет необходимости вручную делать эти операции.

Стиль цикла “for-each”

Все классы коллекций модифицированы таким образом, что реализуют интерфейс `Iterable`, что означает, что можно пройти циклом по коллекции, используя стиль “for-each” цикла `for`. Раньше для прохода по коллекциям необходимо было использовать итератор (описанный далее в настоящей главе), программно конструируя цикл. Хотя итераторы все еще применяются для некоторых целей, во многих случаях циклы на основе итераторов могут быть заменены циклами `for`.

Интерфейсы коллекций

Каркас коллекций определяет несколько интерфейсов. В этом разделе представлен обзор каждого из них. Начинать с интерфейсов коллекций необходимо потому, что они определяют фундаментальную природу классов коллекций. Взятые по отдельности, конкретные классы просто представляют различные реализации стандартных интерфейсов. Интерфейсы, которые поддерживают коллекции, перечислены в табл. 17.1.

В дополнение к этим интерфейсам, коллекции также используют интерфейсы `Comparator`, `RandomAccess`, `Iterator` и `ListIterator`, которые детально рассматриваются далее в этой главе. Кратко говоря, `Comparator` определяет два сравниваемых объекта; `Iterator` и `ListIterator` перечисляют объекты в коллекции. Реализуя `RandomAccess`, список поддерживает эффективный произвольный доступ к своим элементам.

Чтобы обеспечить максимальную гибкость в применении, интерфейсы коллекций позволяют некоторым методам быть необязательными. Необязательные методы позволяют модифицировать содержимое коллекций. Коллекции, которые поддерживают эти методы, называются *модифицируемыми*. Коллекции, которые не позволяют изменять собственное содержимое, называются *не модифицируемыми*. Если предпринимается попытка вызвать один из этих методов для не модифицируемой коллекции, возбуждается исключение `UnsupportedOperationException`. Все встроенные коллекции модифицируемы.

Таблица 17.1. Интерфейсы, поддерживаемые коллекциями

Интерфейс	Описание
`Collection`	Позволяет работать с группами объектов. Это — вершина иерархии коллекций.
`Deque`	Расширяет `Queue` для обработки двунаправленных очередей. (Добавлен в Java SE 6.)
`List`	Расширяет `Collection` для управления последовательностями (списками объектов).
`NavigableSet`	Расширяет `SortedSet` для обработки извлечения элементов на основе поисков по ближайшему соответствию. (Добавлено в Java SE 6.)
`Queue`	Расширяет `Collection` для управления специальными типами списков, в которых элементы удаляются только с начала.
`Set`	Расширяет `Collection` для управления множествами (наборами), которые должны содержать уникальные элементы.
`SortedSet`	Расширяет `Set` для управления сортированными множествами.

Следующий раздел посвящен интерфейсам коллекций.

Интерфейс `Collection`

Интерфейс `Collection` — это фундамент, на котором построен весь каркас коллекций, поскольку он должен быть реализован всеми классами коллекций. `Collection` — обобщенный интерфейс, имеющий следующее объявление:

```
interface Collection<E>
```

Здесь `E` указывает тип объектов, которые будет содержать коллекция. `Collection` расширяет интерфейс `Iterable`. Это означает, что по всем коллекциям можно проходить циклами вида “for-each”. (Вспомните, что только те классы, которые реализуют `Iterable`, позволяют проходить по их элементам циклами `for`.)

`Collection` определяет основные методы, которые будут иметь все коллекции. Эти методы собраны в табл. 17.2.

Таблица 17.2. Методы, определенные в `Collection`

Метод	Описание
`boolean add(E obj)`	Добавляет *obj* к вызывающей коллекции. Возвращает `true`, если *obj* был добавлен к коллекции.
`boolean addAll(Collection<? extends E> c)`	Добавляет все элементы *c* к вызывающей коллекции. Возвращает `true`, если операция удалась (то есть все элементы добавлены). В противном случае возвращает `false`.
`void clear()`	Удаляет все элементы вызывающей коллекции.
`boolean contains(Object obj)`	Возвращает `true`, если *obj* является элементом вызывающей коллекции. В противном случае возвращает `false`.
`boolean containsAll(Collection<?> c)`	Возвращает `true`, если вызывающая коллекция содержит все элементы *c*. В противном случае возвращает `false`.

Окончание табл. 17.1

Метод	Описание
`boolean equals(Object obj)`	Возвращает `true`, если вызывающая коллекция и *obj* эквивалентны. В противном случае возвращает `false`.
`int hashCode()`	Возвращает хеш-код вызывающей коллекции.
`boolean isEmpty()`	Возвращает `true`, если вызывающая коллекция пуста. В противном случае возвращает `false`.
`Iterator<E> iterator()`	Возвращает итератор для вызывающей коллекции.
`boolean remove(Object obj)`	Удаляет один экземпляр *obj* из вызывающей коллекции. Возвращает `true`, если элемент удален. В противном случае возвращает `false`.
`boolean removeAll(Collection<?> c)`	Удаляет все элементы *c* из вызывающей коллекции. Возвращает `true`, если в результате коллекция изменяется (то есть элементы удалены). В противном случае возвращает `false`.
`boolean retainAll(Collection<?> c)`	Удаляет все элементы кроме входящих в *c* из вызывающей коллекции. Возвращает `true`, если в результате коллекция изменяется (то есть элементы удалены). В противном случае возвращает `false`.
`int size()`	Возвращает количество элементов, содержащихся в коллекции.
`Object[] toArray()`	Возвращает массив, содержащий все элементы вызывающей коллекции. Элементы массива являются копиями элементов коллекции.
`<T> T[] toArray(T array[])`	Возвращает массив, содержащий элементы вызывающей коллекции. Элементы массива являются копиями элементов коллекции. Если размер массива *array* равен количеству элементов, он возвращается. Если размер *array* меньше количества элементов, создается и возвращается новый массив нужного размера. Если размер *array* больше количества элементов, то элементы, следующие за последним из коллекции, устанавливаются равными `null`. Если любой из элементов коллекции имеет тип, не являющийся подтипом *array*, возбуждается исключение `ArrayStoreException`.

Поскольку все коллекции реализуют `Collection`, знакомство с его методами необходимо для четкого понимания каркаса коллекций. Некоторые из этих методов могут инициировать исключение `UnsupportedOperationException`. Как уже объяснялось, это происходит, если коллекция не может быть модифицирована. Исключение `ClassCastException` генерируется, когда объекты несовместимы между собой, как, например, в случае, когда предпринимается попытка добавить несовместимый объект в коллекцию. Исключение `NullPointerException` возбуждается при попытке вставить значение `null` в коллекцию, не допускающую `null`-элементов. Исключение `IllegalArgumentException` возбуждается, когда используется неправильный аргумент.

Исключение IllegalStateException возбуждается, когда осуществляется попытка вставить новый элемент в заполненную коллекцию фиксированной длины.

Объекты добавляются в коллекции методом add(). Отметим, что add() принимает аргумент типа E, что означает, что добавляемые в коллекцию объекты должны быть совместимыми с ожидаемым типом данных коллекции. Вы можете добавить все содержимое одной коллекции к другой вызовом addAll().

Вы можете удалить объект, используя remove(). Чтобы удалить группу объектов, вызывайте removeAll(). Можно также удалить все объекты, кроме указанных, применив метод retainAll(). Для очистки коллекции потребуется вызвать clear().

Вы можете определить, содержит ли коллекция определенный объект, вызвав метод contains(). Чтобы определить, содержит ли одна коллекция все члены другой, вызывайте containsAll(). Определить, пуста ли коллекция, можно с помощью метода isEmpty(). Количество элементов, содержащихся в данный момент в коллекции, возвращает метод size().

Методы toArray() возвращают массив, который содержит элементы, хранящиеся в коллекции. Первый из них возвращает массив Object. Второй — массив элементов того типа, что и массив, указанный в параметре. Обычно второй метод более предпочтителен, поскольку он возвращает массив элементов нужного типа. Эти методы важнее, чем может показаться на первый взгляд. Часто обработка содержимого коллекции с применением синтаксиса массивов выгодна. Имея простой способ превращения коллекций в массивы, вы имеете доступ к преимуществам обоих представлений.

Две коллекции можно сравнить на эквивалентность, вызывая equals(). Точный смысл "эквивалентности" может изменяться от коллекции к коллекции. Например, вы можете реализовать метод equals() так, чтобы он сравнивал значения элементов, хранимых в коллекции. В качестве альтернативы equals() может сравнивать ссылки на эти элементы.

Еще один очень важный метод — это iterator(), который возвращает итератор коллекции. Итераторы очень часто используются при работе с коллекциями.

Интерфейс List

Интерфейс List расширяет Collection и определяет такое поведение коллекций, которое сохраняет последовательность элементов. Элементы могут быть вставлены или извлечены по их позиции в списке, используя начинающийся с нуля индекс. Список может содержать повторяющиеся элементы. List — обобщенный интерфейс, объявленный следующим образом:

```
interface List<E>
```

Здесь E указывает тип объектов, которые должен содержать список.

В дополнение к методам, объявленным в Collection, List определяет некоторые свои, которые приведены в таблице 17.3. Еще раз отметим, что некоторые из этих методов генерируют исключение UnsupportedOperationException, если коллекция не может быть модифицирована, а исключение ClassCastException генерируется, когда один из объектов несовместим с другим, как, например, когда осуществляется попытка добавить в список элемент несовместимого типа. Кроме того, некоторые методы инициируют исключение IndexOutOfBounsException, если используется неправильный индекс. Исключение NullPointerException возбуждается, когда предпринимается попытка вставить в список объект null, а null-элементы в данном списке не допускаются. Исключение IllegalArgumentException возбуждается, когда передается неверный аргумент.

Таблица 17.3. Методы, определенные в `List`

Метод	Описание
`void add(int index, E obj)`	Вставляет `obj` в вызывающий список в позицию, указанную в `index`. Любые ранее вставленные элементы за указанной позицией вставки смещаются вверх. То есть никакие элементы не перезаписываются.
`boolean addAll(int index, Collection<? extends E> c)`	Вставляет все элементы `c` в вызывающий список, начиная с позиции, переданной в `index`. Все ранее существовавшие элементы за точкой вставки смещаются вверх. То есть никакие элементы не перезаписываются. Возвращает `true`, если вызывающий список изменяется, и `false` — в противном случае.
`E get(int index)`	Возвращает объект, сохраненный в указанной позиции вызывающего списка.
`int indexOf(Object obj)`	Возвращает индекс первого экземпляра `obj` в вызывающем списке. Если `obj` не содержится в списке, возвращается `-1`.
`int lastIndexOf(Object obj)`	Возвращает индекс последнего экземпляра `obj` в вызывающем списке. Если `obj` не содержится в списке, возвращается `-1`.
`ListIterator<E> listIterator()`	Возвращает итератор, указывающий на начало списка.
`ListIterator<E> listIterator(int index)`	Возвращает итератор, указывающий на заданную позицию в списке.
`E remove(int index)`	Удаляет элемент из вызывающего списка в позиции `index` и возвращает удаленный элемент. Результирующий список уплотняется, то есть элементы, следующие за удаленным, сдвигаются на одну позицию назад.
`E set(int index, E obj)`	Присваивает `obj` элементу, находящемуся в списке в позиции `index`.
`List<E> subList(int start, int end)`	Возвращает список, включающий элементы от `start` до `end-1` из вызывающего списка. Элементы из возвращаемого списка также сохраняют ссылки в вызывающем списке.

К версиям методов `add()` и `addAll()`, определенным в `Collection`, класс `List` добавляет методы `add(int, E)` и `addAll(int, Collection)`. Эти методы вставляют элементы в позицию, указанную индексом.

Также семантика методов `add(E)` и `addAll(Collection)`, определенная в `Collection`, изменяется в `List` таким образом, что они добавляют элементы в конец списка.

Чтобы получить объект, сохраненный в определенной позиции, вызывайте метод `get()` с индексом объекта. Чтобы присвоить значение элементу списка, вызывайте `set()`, указав индекс объекта, который требуется изменить. Чтобы найти индекс объекта, применяйте `indexOf()` или `lastIndexOf()`.

Вы можете получить подсписок данного списка, вызвав метод `subList()` и указав начальный и конечный индексы этого подсписка. Как вы можете представить, `subList()` обеспечивает значительное удобство работы со списками.

Интерфейс Set

Интерфейс Set определяет множество (набор). Он расширяет Collection и определяет поведение коллекций, не допускающих дублирования элементов. Таким образом, метод add() возвращает false, если делается попытка добавить дублированный элемент в набор. Он не определяет никаких собственных дополнительных методов. Set — обобщенный интерфейс, который объявлен следующим образом:

```
interface Set<E>
```

Здесь E указывает тип объектов, которые должен содержать набор.

Интерфейс SortedSet

Интерфейс SortedSet расширяет Set и объявляет поведение множеств, сортированных в порядке возрастания. SortedSet — обобщенный интерфейс, который имеет следующее объявление:

```
interface SortedSet<E>
```

Здесь E указывает тип объектов, которые должен содержать набор.

В дополнение к методам, определенным в Set, интерфейс SortedSet объявляет методы, перечисленные в таблице 17.4. Некоторые из них инициируют исключение NoSuchElementException, когда никаких элементов в вызывающем множестве не содержится. Исключение ClassCastException инициируется, когда объект не совместим с элементами множества. Исключение NullPointerException генерируется, когда предпринимается попытка использовать null-объект, и null — недопустимое значение множества. При использовании неправильного аргумента возбуждается исключение IllegalArgumentException.

Таблица 17.4. Методы, определенные в SortedSet

Метод	Описание
`Comparator<? super E> comparator()`	Возвращает компаратор сортированного множества. Если для множества применяется естественный порядок сортировки, возвращается null.
`E first()`	Возвращается первый элемент вызывающего сортированного множества.
`SortedSet<E> headSet(E end)`	Возвращается SortedSet, содержащий элементы из вызывающего множества, которые предшествуют *end*. Элементы из возвращенного множества имеют также ссылки в вызывающем объекте.
`E last()`	Возвращается последний элемент вызывающего сортированного множества.
`SortedSet<E> subSet(E start, E end)`	Возвращается SortedSet, который включает элементы, находящиеся между *start* и *end*-1. Элементы из возвращенного множества имеют также ссылки в вызывающем объекте.
`SortedSet<E> tailSet(E start)`	Возвращается SortedSet, содержащий элементы из вызывающего множества, которые следуют за *end*. Элементы из возвращенного множества имеют также ссылки в вызывающем объекте.

В SortedSet определено несколько методов, которые облегчают обработку. Чтобы получить первый объект в сортированном множестве, вызывайте first(), а чтобы последний — метод last(). Вы можете получить подмножество сортированного множества, вызвав subSet() и передав ему первый и последний объекты в подмножестве. Если вам нужно подмножество, которое начинается с первого элемента множества, используйте headSet(). Если же требуется подмножество из конца множества, тогда вызывайте tailSet().

Интерфейс NavigableSet

Интерфейс NavigableSet появился в Java SE 6. Он расширяет SortedSet и объявляет поведение коллекции, которая поддерживает извлечение элементов на основе ближайшего соответствия заданному значению или значениям. NavigableSet — обобщенный интерфейс, имеющий следующее объявление:

```
interface NavigableSet<E>
```

Здесь E специфицирует тип содержащихся в наборе объектов. В дополнение к методам, унаследованным от SortedSet, NavigableSet включает также и те, что перечислены в табл. 17.5.

Таблица 17.5. Методы, определенные в NavigableSet

Метод	Описание
`E ceiling(E obj)`	Ищет в наборе наименьший элемент *e*, для которого истинно *e* >= *obj*. Если такой элемент найден, он возвращается. В противном случае возвращается null.
`Iterator<E> descendingIterator()`	Возвращает итератор, перемещающийся от большего к меньшему, другими словами, обратный итератор.
`NavigableSet<E> descendingSet()`	Возвращает NavigableSet, представляющий собой обратную версию вызывающего набора. Результирующий набор поддерживается вызывающим набором.
`E floor(E obj)`	Ищет в наборе наибольший элемент *e*, для которого истинно *e* <= *obj*. Если такой элемент найден, он возвращается. В противном случае возвращается null.
`NavigableSet<E> headSet( E upperBound, boolean incl)`	Возвращает NavigableSet, включающий все элементы вызывающего набора, меньшие *upperBound*. Результирующий набор поддерживается вызывающим набором.
`E higher(E obj)`	Ищет в наборе наибольший элемент *e*, для которого истинно *e* > *obj*. Если такой элемент найден, он возвращается. В противном случае возвращается null.
`E lower(E obj)`	Ищет в наборе наименьший элемент *e*, для которого истинно *e* < *obj*. Если такой элемент найден, он возвращается. В противном случае возвращается null.
`E pollFirst()`	Возвращает первый элемент, удаляя его в процессе. Поскольку набор сортирован, это будет элемент с наименьшим значением. Возвращает null в случае пустого набора.

Окончание табл. 17.5

Метод	Описание
`E pollLast()`	Возвращает последний элемент, удаляя его в процессе. Поскольку набор сортирован, это будет элемент с наибольшим значением. Возвращает `null` в случае пустого набора.

`NavigableSet<E> subSet(`
`E lowerBound, boolean lowIncl,`
`E upperBound, boolean highIncl)`
	Возвращает `NavigableSet`, включающий все элементы вызывающего набора, которые больше *`lowerBound`* и меньше *`upperBound`*. Если *`lowIncl`* равно `true`, то элемент, равный *`lowerBound`*, включается. Если *`highIncl`* равно `true`, также включается элемент, равный *`upperBound`*.

`NavigableSet<E> tailSet(` `E lowerBound, boolean incl)`	Возвращает `NavigableSet`, включающий все элементы из вызывающего набора, которые больше *`lowerBound`*. Если *`incl`* равно `true`, в результат включается элемент, равный *`lowerBound`*. Результирующий набор поддерживается вызывающим набором.

Исключение `ClassCastException` возбуждается, когда объект несовместим с элементами набора. Исключение `NullPointerException` возникает, если предпринимается попытка вставить `null`-объект, а данный набор значений `null` не допускает. При использовании неправильного аргумента генерируется исключение `IllegalArgumentException`.

Интерфейс `Queue`

Интерфейс `Queue` расширяет `Collection` и объявляет поведение очередей, которые представляют собой список с дисциплиной "первый вошел — первый вышел". Однако существуют разные типы очередей, в которых порядок основан на некотором критерии. `Queue` — обобщенный интерфейс со следующим объявлением:

```
interface Queue<E>
```

Здесь `E` указывает тип объектов, которые будут храниться в очереди. Методы, определенные в `Queue`, представлены в табл. 17.6.

Таблица 17.6. Методы, определенные в `Queue`

Метод	Описание
`E element()`	Возвращает элемент из головы очереди. Элемент не удаляется. Если очередь пуста, инициируется исключение `NoSuchElementException`.
`boolean offer(E obj)`	Пытается добавить *`obj`* в очередь. Возвращает `true`, если *`obj`* добавлен, и `false` — в противном случае.
`E peek()`	Возвращает элемент из головы очереди. Возвращает `null`, если очередь пуста. Элемент не удаляется.
`E poll()`	Возвращает элемент из головы очереди и удаляет его. Возвращает `null`, если очередь пуста.
`E remove()`	Удаляет элемент из головы очереди, возвращая его. Инициирует исключение `NoSuchElementException`, если очередь пуста.

Несколько методов возбуждают исключение ClsssCastException, когда объект не совместим с элементами очереди. NullPointerException генерируется при попытке сохранения null-объекта, когда null-элементы в очереди не разрешены. IllegalArgimentException инициируется при использовании неверного аргумента. IllegalStateException возбуждается, когда предпринимается попытка вставки в полную очередь фиксированной длины. Исключение NoSuchElementException возникает при попытке удалить элемент из пустой очереди.

Несмотря на свою простоту, Queue представляет интерес с нескольких точек зрения. Во-первых, элементы могут удаляться только из начала очереди. Во-вторых, есть два метода, которыми можно получать и удалять элементы: poll() и remove().

Разница между ними состоит в том, что poll() возвращает null, если очередь пуста, а remove() возбуждает исключение. В-третьих, есть два метода, element() и peek(), которые получают элемент из головы очереди, но не удаляют его. Отличаются они тем, что при пустой очереди element() инициирует исключение, а peek() возвращает null. И, наконец, отметим, что offer() только пытается добавить элемент в очередь. Поскольку некоторые очереди имеют фиксированную длину и могут быть полны, offer() может завершиться неудачно.

Интерфейс Dequeue

Интерфейс Dequeue появился в Java SE 6. Он расширяет Queue и описывает поведение двунаправленной очереди. Двунаправленная очередь может функционировать как стандартная очередь "первый вошел — первый вышел" либо как стек "последний вошел — первый вышел". Dequeue — обобщенный интерфейс со следующим объявлением:

```
interface Deque<E>
```

Здесь E специфицирует тип объектов, которые будет содержать двусторонняя очередь. В дополнение к методам, унаследованным от Queue, Dequeue добавляет методы, перечисленные в табл. 17.7. Несколько методов возбуждают ClassCastException, когда объект несовместим с элементами в двунаправленной очереди. NullPointerException генерируется при попытке сохранения null-объекта, когда null-элементы в двунаправленной очереди не допускаются. IllegalArgimentException инициируется при использовании неверного аргумента. IllegalStateException возбуждается, когда осуществляется попытка вставки в полную двунаправленную очередь фиксированной длины. Исключение NoSuchElementException возникает, когда предпринимается попытка удалить элемент из пустой очереди.

Обратите внимание, что Deque включает методы push() и pop(). Эти методы позволяют Deque функционировать в качестве стека. Кроме того, следует обратить внимание на метод descendingIterator(). Он возвращает итератор, который возвращает элементы в обратном порядке. Другими словами, итератор, перемещающийся от конца коллекции к ее началу. Реализация Deque может быть ограниченной по емкости, то есть в него может быть добавлено ограниченное количество элементов. В этом случае попытка добавления элемента может вызвать исключение. Deque позволяет обрабатывать такие сбои двумя способами. Во-первых, методы вроде addFirst() и addLast() возбуждают исключение IllegalStateException, если двунаправленная очередь имеет ограниченную емкость. Во-вторых, такие методы, как offerFirst() и offerLast(), возвращают false, когда элемент не может быть добавлен.

Таблица 17.7. Методы, определенные в Dequeue

Метод	Описание
`void addFirst(E obj)`	Добавляет *obj* в голову двунаправленной очереди. Возбуждает исключение `IllegalStateException`, если в очереди ограниченной емкости нет места.
`void addLast(E obj)`	Добавляет *obj* в хвост двунаправленной очереди. Возбуждает исключение `IllegalStateException`, если в очереди ограниченной емкости нет места.
`Iterator<E> descendingIterator()`	Возвращает итератор, перемещающийся от хвоста к голове двунаправленной очереди. То есть, возвращает обратный итератор.
`E getFirst()`	Возвращает первый элемент двунаправленной очереди. Объект из очереди не удаляется. В случае пустой двунаправленной очереди возбуждает исключение `NoSuchElementException`.
`E getLast()`	Возвращает последний элемент двунаправленной очереди. Объект из очереди не удаляется. В случае пустой двунаправленной очереди возбуждает исключение `NoSuchElementException`.
`boolean offerFirst(E obj)`	Пытается добавить *obj* в голову двунаправленной очереди. Возвращает `true`, если *obj* добавлен, и `false` — в противном случае. Таким образом, этот метод возвращает `false` при попытке добавить *obj* в полную двунаправленную очередь ограниченной емкости.
`boolean offerLast(E obj)`	Пытается добавить *obj* в хвост двунаправленной очереди. Возвращает `true`, если *obj* добавлен, и `false` — в противном случае.
`E peekFirst()`	Возвращает элемент, находящийся в голове двунаправленной очереди. Возвращает `null`, если очередь пуста. Объект из очереди не удаляется.
`E peekLast()`	Возвращает элемент, находящийся в хвосте двунаправленной очереди. Возвращает `null`, если очередь пуста. Объект из очереди не удаляется.
`E pollFirst()`	Возвращает элемент, находящийся в голове двунаправленной очереди, одновременно удаляя его из очереди. Возвращает `null`, если очередь пуста.
`E pollLast()`	Возвращает элемент, находящийся в хвосте двунаправленной очереди, одновременно удаляя его из очереди. Возвращает `null`, если очередь пуста.
`E pop()`	Возвращает элемент, находящийся в голове двунаправленной очереди, одновременно удаляя его из очереди. Возбуждает исключение `NoSuchElementException`, если очередь пуста.
`void push(E obj)`	Добавляет элемент в голову двунаправленной очереди. Если в очереди фиксированного объема нет места, возбуждает исключение `IllegalStateException`.
`E removeFirst()`	Возвращает элемент, находящийся в голове двунаправленной очереди, одновременно удаляя его из очереди. Возбуждает исключение `NoSuchElementException`, если очередь пуста.

Окончание табл. 17.7

Метод	Описание
`boolean removeFirstOccurrence(Object obj)`	Удаляет первое вхождение *obj* из двунаправленной очереди. Возвращает `true` в случае успеха и `false`, если очередь не содержала *obj*.
`E removeLast()`	Возвращает элемент, находящийся в конце двунаправленной очереди, удаляя его в процессе. Возбуждает исключение `NoSuchElementException`, если очередь пуста.
`boolean removeLastOccurrence(Object obj)`	Удаляет последнее вхождение *obj* из двунаправленной очереди. Возвращает `true` в случае успеха и `false` — если очередь не содержала *obj*.

Классы коллекций

Теперь, когда вы знакомы с интерфейсами коллекций, вы готовы приступить к изучению стандартных классов, их реализующих. Некоторые классы представляют полную реализацию и могут применяться "как есть". Другие же являются абстрактными, представляя только скелетные реализации, которые используются в качестве начальных точек при создании конкретных коллекций. Ни один из классов коллекций не синхронизирован, но как будет показано далее в настоящей главе, при необходимости можно получить их синхронизированные версии.

Стандартные классы коллекций перечислены в табл. 17.8.

Таблица 17.8. Стандартные классы коллекций

Класс	Описание
`AbstractCollection`	Реализует большую часть интерфейса `Collection`.
`AbstractList`	Расширяет `AbstractCollection` и реализует большую часть интерфейса `List`.
`AbstractQueue`	Расширяет `AbstractCollection` и реализует часть интерфейса `Queue`.
`AbstractSequentalList`	Расширяет `AbstractList` для использования в коллекциях, использующих последовательности вместо случайного доступа к элементам.
`LinkedList`	Реализует связный список, расширяя `AbstractSequentalList`.
`ArrayList`	Реализует динамический массив, расширяя `AbstractList`.
`AbstractSet`	Расширяет `AbstractCollection` и реализует большую часть интерфейса `Set`.
`EnumSet`	Расширяет `AbstractSet` для использования с элементами `enum`.
`HashSet`	Расширяет `AbstractSet` для использования с хеш-таблицами.
`LinkedHashSet`	Расширяет `HashSet`, разрешая итерации порядковой вставки.
`PriorityQueue`	Расширяет `AbstractQueue` для поддержки очередей, основанных на приоритетах.
`TreeSet`	Реализует множество, хранимое в дереве. Расширяет `AbstractSet`.

В следующих разделах рассматриваются конкретные классы коллекций и иллюстрируется их применение.

На заметку! *В дополнение к классам коллекций некоторые унаследованные от прежних версий классы, такие как* `Vector`, `Stack` *и* `HashTable`, *были перепроектированы для поддержки коллекций. Они также рассматриваются далее в настоящей главе.*

Класс ArrayList

Класс `ArrayList` расширяет `AbstractList` и реализует интерфейс `List`. `ArrayList` — это обобщенный класс со следующим объявлением:

```
class ArrayList<E>
```

Здесь `E` указывает тип сохраняемых объектов.

`ArrayList` поддерживает динамические массивы, которые могут расти по мере необходимости. Стандартные массивы Java имеют фиксированную длину. После того, как массив создан, он не может расти или уменьшаться, а это означает, что вы должны заранее знать, сколько элементов нужно в нем хранить. Но иногда вы не можете точно знать до момента выполнения программы, насколько большой массив понадобится. Чтобы справиться с этой ситуацией, в каркасе коллекций определен класс `ArrayList`. По сути, `ArrayList` — это массив объектных ссылок переменной длины. То есть `ArrayList` может динамически увеличиваться или уменьшаться в размере. Массивы-списки (ArrayList) создаются с некоторым начальным размером. Когда этот первоначальный размер становится недостаточным, коллекция автоматически увеличивается. Когда объекты удаляются, коллекция может уменьшаться.

На заметку! *Динамические массивы также поддерживаются унаследованным классом* `Vector`, *который будет описан далее в настоящей главе.*

`ArrayList` имеет следующие конструкторы:

```
ArrayList()
ArrayList(Collection <? extends E> c)
ArrayList(int capacity)
```

Первый конструктор создает пустой массив-список. Второй строит массив-список, который инициализируется элементами коллекции *c*. Третий конструктор создает массив-список, который имеет начальную емкость *capacity*. Емкость — это размер лежащего в основе массива, используемого для хранения элементов. Емкость растет автоматически по мере добавления элементов в массив-список.

В следующей программе демонстрируется простое применение `ArrayList`. Создается массив-список для объектов типа `String`, затем в него добавляются несколько строк. (Вспомните, что строки в кавычках транслируются в объекты `String`.) Затем список отображается. Некоторые элементы удаляются, и список отображается снова.

```
// Демонстрация использования ArrayList.
import java.util.*;

class ArrayListDemo {
public static void main (String args[]) {
  // Создать массив-список.
  ArrayList<String> al = new ArrayList<String>();

  System.out.println("Начальный размер al: " + al.size());
```

```
    // Добавить элементы в массив-список.
    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 al после вставки: " +
    al.size());

    // отобразить массив-список.
    System.out.println("Содержимое al: " + al);

    // Удалить элементы из массива-списка.
    al.remove("F");
    al.remove(2);

    System.out.println("Размер al после удалений: " + al.size());

    System.out.println("Содержимое al: " + al);
  }
  }
```

Вывод программы показан ниже:

```
Начальный размер al: 0
Размер al после вставки: 7
Содержимое al: [C, A2, A, E, B, D, F]
Размер al после удалений: 5
Содержимое al: [C, A2, E, B, D]
```

Обратите внимание, что `al` изначально пуст и растет по мере добавления в него элементов. Когда элементы удаляются, его размер уменьшается.

В предыдущем примере содержимое коллекции отображается с использованием преобразования типов по умолчанию, предоставляемого методом `toString()`, который унаследован от `AbstractCollection`. Несмотря на то что это удобно при написании коротких примеров программ, вы редко будете пользоваться этим методом для отображения содержимого реальных коллекций. Обычно вы будете предусматривать свои собственные процедуры вывода. Но для нескольких следующих примеров вывод по умолчанию, выполняемый методом `toString()`, вполне подойдет.

Невзирая на то что емкость объектов `ArrayList` растет автоматически, по мере сохранения в них объектов, вы также можете увеличивать эту емкость вручную, вызывая метод `ensureCapacity()`. Это может потребоваться, если вы заранее знаете, что собираетесь сохранить в коллекции намного больше элементов, чем она содержит в данный момент. Увеличивая емкость однажды, в начале работы, вы тем самым предотвращаете несколько дополнительных распределений памяти позднее. Поскольку перераспределения памяти — дорогостоящие операции в смысле временных затрат, предотвращение таких излишних операций увеличивает производительность. Сигнатура метода `ensureCapacity()` показана ниже:

```
void ensureCapacity(int cap)
```

Здесь *cap* — новая емкость коллекции.

И наоборот, когда вы хотите уменьшить размер массива объектов, лежащего в основе ArrayList, до текущего реального количества хранимых объектов, вызывайте метод trimToSize():

```
void trimToSize()
```

Получение массива из ArrayList

При работе с ArrayList иногда необходимо получить обыкновенный массив, содержащий все элементы списка. Это можно сделать, вызвав метод toArray(), который определен в Collection.

Существует несколько причин, по которым вам может понадобиться преобразовать коллекцию в массив, например:

- для ускорения выполнения некоторых операций;
- для передачи массива в качестве параметра методам, не перегруженным для непосредственного использования коллекций;
- для интеграции нового кода, основанного на коллекциях с унаследованным кодом, который не понимает коллекций.

Независимо от причины, преобразование ArrayList в массив — задача тривиальная.

Как объяснялось ранее, существует две версии метода toArray(), которые показаны ниже:

```
Object[] toArray()
<T> T[] toArray(T array[])
```

Первая версия возвращает массив объектов типа Object. Вторая возвращает массив элементов, имеющих тип T. Обычно вторая форма более удобна, поскольку возвращает правильный тип массива. В следующей программе демонстрируется их применение.

```
// Преобразование ArrayList в массив.
import java.util.*;

class ArrayListToArray {
public static void main(String args[]) {
  // Создать массив-список.
  ArrayList<Integer> al = new ArrayList<Integer>();

  // Добавить элементы в массив-список.
  al.add(1);
  al.add(2);
  al.add(3);
  al.add(4);

  System.out.println("Содержимое al: " + al);

  // Получить массив.
  Integer ia[] = new Integer[al.size()];
  ia = al.toArray(ia);

  int sum = 0;

  // Суммировать массив.
  for(int i : ia) sum += i;

  System.out.println("Сумма: " + sum);
}
}
```

Вывод программы будет выглядеть так:

```
Содержимое al: [1, 2, 3, 4]
Сумма: 10
```

Программа начинается с создания коллекции целых чисел. Затем вызывается `toArray()` и получается массив элементов типа `Integer`. Далее содержимое массива суммируется в цикле `for` вида "for-each".

Есть еще кое-что интересное в этой программе. Как вы знаете, коллекции могут содержать только ссылки, а не значения примитивных типов. Однако автоматическая упаковка делает возможным передавать методу `add()` значения типа `int`, без необходимости осуществлять помещение их в оболочку `Integer`, как это демонстрируется в программе. Это осуществляет автоматическая упаковка. Таким образом, она ощутимо облегчает сохранение в коллекциях значений примитивных типов.

Класс `LinkedList`

Класс `LinkedList` расширяет `AbstractSequentalList` и реализует интерфейсы `List`, `Dequeue` и `Queue`. Он представляет структуру данных связного списка. `LinkedList` — обобщенный класс со следующим объявлением:

```
class LinkedList<E>
```

Здесь `E` указывает тип сохраняемых в списке объектов. `LinkedList` имеет следующие два конструктора:

```
LinkedList()
LinkedList(Collection<? extends E> c)
```

Первый конструктор создает пустой связный список. Второй — строит связный список и инициирует его содержимым коллекции *c*.

Поскольку `LinkedList` реализует интерфейс `Deque`, вы имеете доступ к методам, определенным в `Deque`. Например, чтобы добавить элементы в начало списка, вы можете использовать `addFirst()` или `offerFirst()`. Для добавления элементов в конец списка применяйте `addLast()` или `offerLast()`. Чтобы получить первый элемент, используйте `getLast()` или `peekLast()`. Чтобы удалить первый элемент, вызывайте `removeFirst()` или `pollFirst()`. Для удаления последнего элемента используется `removeLast()` или `pollLast()`.

В следующей программе иллюстрируется использование `LinkedList`.

```
// Демонстрация применения LinkedList.
import java.util.*;

class LinkedListDemo {
public static void main(String args[]) {
  // Создать связный список.
  LinkedList<String> ll = new LinkedList<String>();
  // Добавить 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 ll: " + ll);

    // Удалить элементы из связного списка.
    ll.remove("F");
    ll.remove(2);
    System.out.println("Содержимое ll после удаления: " + ll);

    // Удалить первый и последний элементы.
    ll.removeFirst();
    ll.removeLast();
    System.out.println("ll после удаления первого и последнего: "+ ll);

    // Получить и присвоить значение.
    String val = ll.get(2);
    ll.set(2, val + " Изменен");
    System.out.println("ll после изменения: " + ll);
  }
  }
```

Ниже показан вывод этой программы:

```
Исходное содержимое ll: [A, A2, F, B, D, E, C, Z]
Содержимое ll после удаления: [A, A2, D, E, C, Z]
ll после удаления первого и последнего: [A2, D, E, C]
ll после изменения: [A2, D, E Changed, C]
```

Поскольку `LinkedList` реализует интерфейс `List`, вызов `add(E)` добавляет элементы в конец списка, как это делает и `addLast()`. Чтобы вставить элементы в определенное место, используйте форму `add(int, E)`, как демонстрируется в примере вызовом `add(1, "A2")`.

Обратите внимание на то, как изменяется третий элемент в `ll` с помощью методов `get()` и `set()`. Чтобы получить текущее значение элемента методу `get()` передается индекс позиции, в которой расположен элемент. Чтобы присвоить новое значение элементу в этой позиции, методу `set()` передается этот индекс и новое значение.

Класс HashSet

Класс `HashSet` расширяет `AbstractSet` и реализует интерфейс `Set`. Он создает коллекцию, которая использует для хранения хеш-таблицу. `HashSet` — обобщенный класс, имеющий следующее объявление:

```
class HashSet<E>
```

Здесь `E` указывает тип объектов, которые будут храниться в наборе.

Как большинство читателей, вероятно, знает, хеш-таблица хранит информацию, используя так называемый механизм *хеширования*, в котором содержимое ключа используется для определения уникального значения, называемого *хеш-кодом*. Этот хеш-код затем применяется в качестве индекса, с которым ассоциируются данные, доступные по этому ключу. Преобразование ключа в хеш-код выполняется автоматически — вы никогда не увидите самого хеш-кода. Также ваш код не может напрямую индексировать хеш-таблицу. Выгода от хеширования состоит в том, что оно обеспечивает константное время выполнения операций `add()`, `contains()`, `remove()` и `size()`, даже для больших наборов.

Определены следующие конструкторы:

```
HashSet()
HashSet(Collection<? extends E> c)
HashSet(int capacity)
HashSet(int capacity, float fillRatio)
```

Первая форма конструктора создает хеш-набор по умолчанию. Вторая форма инициирует его содержимым коллекции *c*. Третья форма устанавливает емкость хеш-набора равной *capacity*. (Емкость по умолчанию — 16.) Четвертая форма инициирует и емкость, и отношение наполнения (*fill ratio*, также называемое емкостью загрузки, *load capacity*) из аргументов конструктора. Отношение наполнения должно быть от 0,0 до 1,0, и оно определяет, насколько заполненным должен быть хеш-набор, прежде чем будет выполнено изменение его размера. Точнее говоря, когда количество элементов становится больше емкости хеш-набора, умноженной на отношение наполнения, такой хеш-набор увеличивается. В конструкторах, которые не принимают параметр отношения наполнения, принимается значение 0,75.

`HashSet` не определяет никаких дополнительных методов, помимо представленных его суперклассами и интерфейсами.

Важно отметить, что `HashSet` не гарантирует упорядоченности элементов, поскольку процесс хеширования сам по себе обычно не порождает сортированных наборов. Если вам нужны сортированные наборы, то лучшим выбором может быть другой тип коллекций, такой как `TreeSet`.

Ниже показан пример применения `HashSet`.

```
// Демонстрация применения HashSet.
import java.util.*;

class HashSetDemo {
public static void main(String args[]) {
  // Создать хеш-набор.
  HashSet<String> hs = new HashSet<String>();

  // Добавить элементы в хеш-набор.
  hs.add("B");
  hs.add("A");
  hs.add("D");
  hs.add("E");
  hs.add("C");
  hs.add("F");
  System.out.println(hs);
}
}
```

Вывод этой программы выглядит следующим образом:

```
[D, A, F, C, B, E]
```

Как объяснялось, элементы не сохраняются в сортированном порядке, поэтому порядок их вывода может варьироваться.

Класс `LinkedHashSet`

Класс `LinkedHashSet` расширяет `HashSet`, не добавляя никаких новых методов. Это обобщенный класс со следующим объявлением:

```
class LinkedHashSet<E>
```

Здесь `E` указывает тип объектов, которые будут храниться в наборе. Конструкторы этого класса параллельны тем, что имеются у `HashSet`.

`LinkedHashSet` поддерживает связный список элементов набора в том порядке, в котором они вставлялись. Это позволяет организовать упорядоченную итерацию вставки в

набор. То есть, когда идет цикл по LinkedHashSet с применением итератора, элементы извлекаются в том порядке, в каком они были вставлены. Это также тот порядок, в котором они будут возвращены методом toString() объекта LinkedHashSet. Чтобы увидеть эффект от применения LinkedHashSet, попробуйте подставить его в предыдущем примере вместо HashSet. Вывод программы после этого будет выглядеть так:

```
[B, A, D, E, C, F]
```

Это отражает порядок, в каком элементы вставлялись в набор.

Класс TreeSet

Класс TreeSet расширяет AbstractSet и реализует интерфейс NavigableSet. Он создает коллекцию, которая для хранения элементов применяет дерево. Объекты сохраняются в отсортированном порядке по возрастанию. Время доступа и извлечения элементов достаточно мало, что делает TreeSet блестящим выбором для хранения больших объемов сортированной информации, которая должна быстро находиться. TreeSet — обобщенный класс со следующим объявлением:

```
class TreeSet<E>
```

Здесь E указывает тип объектов, которые будут храниться в наборе.
TreeSet имеет следующие конструкторы:

```
TreeSet()
TreeSet(Collection<? extends E> c)
TreeSet(Comparator<? super E> comp)
TreeSet(SortedSet<E> ss)
```

Первая форма конструирует пустой набор-дерево, который будет сортировать элементы в естественном порядке возрастания. Вторая форма строит набор-дерево, содержащий элементы коллекции *c*. Третья форма конструирует пустой набор-дерево, элементы в котором будут отсортированы компаратором, указанным в параметре *comp*. (Компараторы рассматриваются далее в настоящей главе.) Четвертая форма строит набор-дерево, содержащий элементы из *ss*. Ниже показан пример использования TreeSet.

```
// Демонстрация TreeSet.
import java.util.*;

class TreeSetDemo {
public static void main(String args[]) {
  // Создать TreeSet.
  TreeSet<String> ts = new TreeSet<String>();
  // Добавить элементы в TreeSet.
  ts.add("C");
  ts.add("A");
  ts.add("B");
  ts.add("E");
  ts.add("F");
  ts.add("D");
  System.out.println(ts);
}
}
```

Вот как выглядит вывод этой программы:

```
[A, B, C, D, E, F]
```

Как уже объяснялось, поскольку TreeSet сохраняет элементы дерева, они автоматически располагаются в отсортированном порядке, что и подтверждает вывод программы.

Поскольку TreeSet реализует интерфейс NavigableSet (добавленный в Java SE 6), вы можете использовать методы, определенные в NavigableSet, для извлечения элементов TreeSet. Например, предположим, что в предыдущую программу был добавлен следующий оператор, который использует subSet() для получения подмножества ts, содержащего элементы между C (включительно) и F (исключительно). Затем отображается результирующий набор.

```
System.out.println(ts.subSet()("C", "F"));
```

Вывод этого оператора показан ниже:

```
[C, D, E]
```

При желании вы можете поэкспериментировать с другими методами, определенными в NavigableSet.

Класс PriorityQueue

Класс PriorityQueue расширяет AbstractQueue и реализует интерфейс Queue. Он создает очередь с приоритетами на базе компаратора очереди. PriorityQueue является обобщенным классом со следующим объявлением:

```
class PriorityQueue<E>
```

Здесь E указывает тип объектов, которые будут храниться в очереди. PriorityQueue является динамической коллекцией и при необходимости может увеличиваться.

PriorityQueue определяет шесть конструкторов:

```
PriorityQueue()
PriorityQueue(int capacity)
PriorityQueue(int capacity, Comparator<? super E> comp)
PriorityQueue(Collection<? extends E> c)
PriorityQueue(PriorityQueue<? extends E> c)
PriorityQueue(SortedSet<? extends E> c)
```

Первый конструктор строит пустую очередь. Его начальная емкость равна 11. Второй конструктор строит очередь с заданной начальной емкостью. Третий конструктор создает очередь заданной емкости с заданным компаратором. Последние три конструктора создают очереди, инициированные элементами коллекций, переданных в параметре *c*. Во всех случаях по мере добавления элементов емкость автоматически растет.

Если не указан никакой компаратор при конструировании PriorityQueue, то применяется компаратор по умолчанию для того типа данных, который сохраняется в очереди. Компаратор по умолчанию размещает элементы очереди в порядке возрастания. Таким образом, в начале (голове) очереди будет находиться наименьшее значение. Однако, предоставляя собственный компаратор, вы можете задать другую схему сортировки элементов. Например, когда сохраняются элементы, включающие временную метку, вы можете ввести приоритеты в очередь таким образом, чтобы самые "старые" элементы располагались в начале очереди.

Вы можете получить ссылку на компаратор, используемый PriorityQueue, вызвав его метод comparator(), показанный ниже:

```
Comparator<? super E> comparator()
```

Он возвращает компаратор. Если в данной очереди применяется естественный порядок сортировки, возвращается null. Одно предупреждение: несмотря на то, что вы можете пройти по элементам PriorityQueue, применяя итератор, порядок итерации не определен. Чтобы правильно использовать PriorityQueue, вы должны вызывать такие методы, как offer() и poll().

Класс ArrayDeque

В Java SE 6 появился класс ArrayDeque, расширяющий AbstractCollection и реализующий интерфейс Deque. Он не добавляет собственных методов. ArrayDeque создает динамический массив, не имеющий ограничений емкости. (Интерфейс Deque поддерживает реализации с ограниченной емкостью, но не накладывает такого требования.) ArrayDeque — обобщенный класс со следующим объявлением:

```
class ArrayDeque<E>
```

Здесь E специфицирует тип объекта, сохраняемого в коллекции. ArrayDeque определяет следующие конструкторы:

```
ArrayDeque()
ArrayDeque(int size)
ArrayDeque(Collection<? extends E> c)
```

Первый конструктор строит пустую двунаправленную очередь. Ее начальная емкость — 16 элементов. Второй конструктор строит двунаправленную очередь с указанной емкостью. Третий конструктор создает двунаправленную очередь, инициализируемую коллекцией, переданной в параметре *c*. Во всех случаях затем по мере необходимости емкость увеличивается при добавлении новых элементов в двунаправленную очередь.

Приведенная ниже программа демонстрирует применение ArrayDeque для организации стека.

```
// Демонстрация применения ArrayDeque.
import java.util.*;

class ArrayDequeDemo {
public static void main(String args[]) {
  // Создать стек.
  ArrayDeque<String> adq = new ArrayDeque<String>();

  // Использование ArrayDeque в виде стека.
  adq.push("A");
  adq.push("B");
  adq.push("D");
  adq.push("E");
  adq.push("F");

  System.out.print("Выталкиваем из стека: ");

  while(adq.peek() != null)
     System.out.print(adq.pop() + " ");

  System.out.println();
}
}
```

Вывод этой программы выглядит так:

```
Выталкиваем из стека: F E D B A
```

Класс EnumSet

EnumSet расширяет AbstractSet и реализует интерфейс Set. Он создает коллекцию, которая предназначена для применения с ключами типа enum. Это обобщенный класс со следующим объявлением:

```
class EnumSet<E extends Enum<E>>
```

Здесь E специфицирует элементы. Отметим, что E должно расширять Enum<E>, а это накладывает требование, что элементы должны относиться к определенному типу enum.

EnumSet не определяет конструкторов. Вместо этого для создания объектов он использует методы фабрики, перечисленные в табл. 17.9. Обратите внимание, что метод of() перегружен множество раз. Это делается из соображений эффективности. Передача известного количества аргументов может работать быстрее, чем применение параметра vararg, когда число аргументов не велико.

Таблица 17.9. Методы, определенные в EnumSet

Метод	Описание
static <E extends Enum<E>> EnumSet<E> allOf(Class<E> *t*)	Создает EnumSet, который содержит элементы перечисления, указанные в *t*.
static <E extends Enum<E>> EnumSet<E> complementOf(EnumSet<E> *e*)	Создает EnumSet, который дополняет элементы, отсутствующие в *e*.
static <E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> *c*)	Создает EnumSet, содержащий элементы из набора *c*.
static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> *c*)	Создает EnumSet, содержащий элементы из набора *c*.
static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> *t*)	Создает EnumSet, содержащий элементы, которые не входят в перечисление, заданное *t*, которое по определению является пустым набором.
static <E extends Enum<E>> EnumSet<E> of(E *v*, E ... *varargs*)	Создает EnumSet, содержащий элементы *v* и ноль или более дополнительных перечислимых значений.
static <E extends Enum<E>> EnumSet<E> of(E *v*)	Создает EnumSet, содержащий элементы *v*.
static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*)	Создает EnumSet, содержащий элементы *v1* и *v2*.
static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*, E *v3*)	Создает EnumSet, содержащий элементы от *v1* до *v3*.
static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*, E *v3*, E *v4*)	Создает EnumSet, содержащий элементы от *v1* до *v4*.
static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*, E *v3*, E *v4*, E *v5*)	Создает EnumSet, содержащий элементы от *v1* до *v5*.
static <E extends Enum<E>> EnumSet<E> range(E *start*, E *end*)	Создает EnumSet, содержащий элементы в диапазоне, заданном *start* и *end*.

Доступ к коллекциям через итератор

Часто вам понадобится проходить по всем элементам коллекции. Например, пусть необходимо отобразить каждый элемент коллекции. Один способ сделать это — использовать итератор, представляющий собой объект, реализующий один из двух интерфейсов: `Iterator` либо `ListIterator`. `Iterator` позволяет организовать цикл для прохода по коллекции, получая либо удаляя элементы. `ListIterator` расширяет `Iterator` для обеспечения двунаправленного прохода по списку и модификации элементов. `Iterator` и `ListIterator` — обобщенные интерфейсы, объявленные, как показано ниже:

```
interface Iterator<E>
interface ListIterator<E>
```

Здесь `E` представляет собой тип объектов, по которым буду выполняться итерации. Интерфейс `Iterator` объявляет методы, перечисленные в табл. 17.10. Методы, объявленные `ListIterator`, показаны в табл. 17.11.

В обоих случаях операции, которые модифицируют лежащую в основе коллекцию, необязательны. Например, `remove()` возбудит исключение `UnsupportedOperationException`, будучи примененным к коллекции, доступной только для чтения. Возможны также и другие исключения.

Таблица 17.10. Методы, определенные в `Iterator`

Метод	Описание
`boolean hasNext()`	Возвращает `true`, если есть еще элементы. В противном случае возвращает `false`.
`E next()`	Возвращает следующий элемент. Возбуждает исключение `NoSuchElementException`, если больше нет элементов.
`void remove()`	Удаляет текущий элемент. Возбуждает исключение `IllegalStateException`, если предпринимается попытка вызвать `remove()`, которой не предшествовал вызов `next()`.

Таблица 17.11. Методы, определенные в `ListIterator`

Метод	Описание
`void add(E obj)`	Вставляет *obj* перед элементом, который должен быть возвращен следующим вызовом `next()`.
`boolean hasNext()`	Возвращает `true`, если есть следующий элемент. В противном случае возвращает `false`.
`boolean hasPrevious()`	Возвращает `true`, если есть предыдущий элемент. В противном случае возвращает `false`.
`E next()`	Возвращает следующий элемент. Если следующего нет, инициируется исключение `NoSuchElementException`.
`int nextIndex()`	Возвращает индекс следующего элемента. Если следующего нет, возвращается размер списка.
`E previous()`	Возвращает предыдущий элемент. Если предыдущего нет, инициируется исключение `NoSuchElementException`.
`int previousIndex()`	Возвращает индекс предыдущего элемента. Если предыдущего нет, возвращается -1.
`void remove()`	Удаляет текущий элемент из списка. Если `remove()` вызван до `next()` или `previous()`, инициируется исключение `IllegalStateException`.
`void set(E obj)`	Присваивает *obj* текущему элементу. Это элемент, возвращенный последним вызовом `next()` или `previous()`.

Использование Iterator

Прежде чем получить доступ к коллекции через итератор, вы должны получить его. Каждый из классов коллекций предлагает метод iterator(), который возвращает итератор на начало коллекции. Используя объект итератора, вы можете получить доступ к каждому элементу коллекции — одному за другим. В общем случае, применение итератора для цикла по содержимому коллекции сводится к выполнению следующих шагов.

1. Установить итератор на начало коллекции, получив его от метода коллекции iterator().
2. Организовать цикл, вызывающий hasNext(). Выполнять циклическую итерацию до тех пор, пока hasNext() возвращает true.
3. Внутри цикла получать каждый элемент, вызывая метод next().

Для коллекций, реализующих List, вы также можете получить итератор, вызывая метод listIterator(). Как уже объяснялось, итератор списка дает возможность доступа к элементам коллекции, как в прямом, так и в обратном направлении, а также позволяет модифицировать элементы. Во всем остальном ListIterator применяется так же, как Iterator.

В следующем примере выполняются все перечисленные шаги с демонстрацией обоих интерфейсов — Iterator и ListIterator. Здесь используется объект ArrayList, но общие принципы применимы к коллекциям любого типа. Конечно, ListIterator доступен только тем коллекциям, которые реализуют интерфейс List.

```
// Демонстрация применения итераторов.
import java.util.*;

class IteratorDemo {
public static void main(String args[]) {
  // Создать массив-список.
  ArrayList<String> al = new ArrayList<String>();

  // Добавить элементы в массив-список.
  al.add("C");
  al.add("A");
  al.add("E");
  al.add("B");
  al.add("D");
  al.add("F");

  // Использовать итераторы для отображения содержимого al.
  System.out.print("Исходное содержимое al: ");
  Iterator<String> itr = al.iterator();
  while(itr.hasNext()) {
     String element = itr.next();
     System.out.print(element + " ");
  }
  System.out.println();

  // Модифицировать текущий объект итерации.
  ListIterator<String> litr = al.listIterator();
  while(litr.hasNext()) {
     String element = litr.next();
     litr.set(element + "+");
  }
```

```
    System.out.print("Модифицированное содержимое al: ");
    itr = al.iterator();
    while(itr.hasNext()) {
      String element = itr.next();
      System.out.print(element + " ");
    }
    System.out.println();

    // Теперь отображаем список в обратном порядке.
    System.out.print("Модифицированный список в обратном порядке: ");
    while(litr.hasPrevious()) {
      String element = litr.previous();
      System.out.print(element + " ");
    }
    System.out.println();
  }
}
```

Ниже показан вывод этой программы:

```
Исходное содержимое al: C A E B D F
Модифицированное содержимое al: C+ A+ E+ B+ D+ F+
Модифицированный список в обратном порядке: F+ D+ B+ E+ A+ C+
```

Обратите особое внимание на то, как список отображается в обратном порядке. После того, как список модифицирован, `litr` указывает на конец списка. (Помните, что `litr.hasNext()` возвращает `false`, когда достигнут конец списка.) Чтобы пройти список в обратном порядке, программа продолжает использовать `litr`, но на этот раз она проверяет, существует ли предшествующий элемент. Пока она это делает, извлеченный элемент отображается.

Версия “for-each” цикла `for` как альтернатива итераторам

Если вам не нужно модифицировать содержимое коллекции либо извлекать элементы в обратном порядке, в этом случае версия “for-each” цикла `for` может оказаться более удобной альтернативой итераторам. Вспомните, что цикл `for` может проходить через любую коллекцию объектов, реализующую интерфейс `Iterable`. Поскольку все классы коллекций реализуют этот интерфейс, ими можно оперировать с помощью `for`.

В следующем примере цикл `for` используется для суммирования содержимого коллекции.

```
// Применение цикла "for-each" для прохода по элементам коллекции.
import java.util.*;

class ForEachDemo {
public static void main(String args[]) {
   // Создать массива-списка для целых чисел.
   ArrayList<Integer> vals = new ArrayList<Integer>();

   // Добавить значения в массив-список.
   vals.add(1);
   vals.add(2);
   vals.add(3);
   vals.add(4);
   vals.add(5);
```

```
    // Использовать цикл для отображения значений.
    System.out.print("Исходное содержимое vals: ");
    for(int v : vals)
      System.out.print(v + " ");
    System.out.println();
    // Суммирование значений в цикле for.
    int sum = 0;
    for(int v : vals)
      sum += v;
    System.out.println("Сумма значений: " + sum);
  }
}
```

Вывод этой программы показан ниже:

```
Исходное содержимое vals: 1 2 3 4 5
Сумма значений: 15
```

Как видите, цикл `for` существенно проще и короче, чем подход на базе итераторов. Однако он может применяться для построения цикла для прохода по элементам коллекции только в прямом направлении, и вы не можете модифицировать элементы коллекции.

Использование пользовательских классов в коллекциях

Все примеры, которые были приведены до сих пор, в целях простоты сохраняли в коллекциях встроенные объекты, такие как `String` или `Integer`. Конечно, коллекции не ограничиваются сохранением встроенных объектов. Как раз наоборот. Мощность коллекций в том, что они могут хранить любой тип объектов, включая объекты классов, которые создаете вы сами. Например, рассмотрим следующий пример, в котором `LinkedList` используется для сохранения почтовых адресов.

```
// Простой пример работы со списком 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>();

  // Добавить 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"));

  // Отобразить список почтовых адресов.
  for(Address element : ml)
     System.out.println(element + "\n");

  System.out.println();
}
}
```

Вывод этой программы приведен ниже:

```
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 пользовательских классов в коллекции, другая важная вещь, на которую стоит обратить внимание в этой программе — это то, что она достаточно коротка. Когда вы оцените, что для сохранения, извлечения и обработки почтовых адресов понадобилось всего около 50 строк кода, станет очевидной мощь каркаса коллекций. Как большинство пользователей знает, если всю эту функциональность запрограммировать вручную, программа станет в несколько раз длиннее. Коллекции предлагают готовые решения для широкого диапазона программистских задач. Вы должны использовать их всякий раз, когда ситуация позволяет.

Интерфейс RandomAccess

Интерфейс `RandomAccess` не имеет членов. Однако, реализуя этот интерфейс, коллекция сообщает о том, что поддерживает эффективный случайный доступ к своим элементам. Несмотря на то что коллекция может поддерживать случайный доступ, он может быть не слишком эффективным. Проверяя интерфейс `RandomAccess`, клиентский код может определять во время выполнения, допускает ли конкретная коллекция некоторые типы операций случайного доступа — особенно, насколько они применимы к большим коллекциям. (Вы можете использовать `instanceof` для определения того, реализует ли класс данный интерфейс.) `RandomAccess` реализуется `ArrayList`, и среди прочих, унаследованным классом `Vector`.

Работа с картами

Карта (map) — это объект, который сохраняет ассоциации между ключами и значениями, или пары "ключ-значение". По заданному ключу вы можете найти его значение. И ключи, и значения являются объектами. Ключи могут быть уникальными, но значения могут дублироваться. Некоторые карты допускают `null`-ключи и `null`-значения, некоторые — нет.

Имеется один ключевой момент относительно карт, который важно упомянуть: они не реализуют интерфейс `Iterable`. Это означает, что вы не можете проходить в цикле по карте, используя форму "for-each" цикла `for`. Более того, вы не можете получить итератор карты. Однако, как вскоре будет показано, можно получить представление карты в виде коллекции, которое допускает использование и цикла, и итераторов.

Интерфейсы `Map`

Поскольку интерфейсы карт определяют их характер и природу, обсуждение начнем с них. Интерфейсы, перечисленные в табл. 17.12, поддерживают карты.

Таблица 17.12. Интерфейсы, которые поддерживают карты

Интерфейс	Описание
`Map`	Отображает уникальные ключи на значения.
`Map.Entry`	Описывает элемент карты (пару "ключ-значение"). Это — вложенный класс `Map`.
`NavigableMap`	Расширяет `SortedMap` для обработки извлечения элементов на основе поиска по ближайшему соответствию. (Добавлен в Java SE 6.)
`SortedMap`	Расширяет `Map` таким образом, что ключи располагаются в порядке по возрастанию.

Ниже каждый из этих интерфейсов рассматривается более подробно.

Интерфейс `Map`

Интерфейс `Map` отображает уникальные ключи на значения. Ключ — это объект, который вы используете для последующего извлечения данных. Задавая ключ и значение, вы можете помещать значения в объект `Map`. После того, как это значение сохранено, вы можете получить его по ключу. `Map` — обобщенный интерфейс, объявленный, как показано ниже:

```
interface Map<K, V>
```

Здесь `K` указывает тип ключей, а `V` — тип хранимых значений.

Методы, объявленные в `Map`, собраны в табл. 17.13. Несколько методов возбуждают исключение `ClassCastException`, когда объект оказывается несовместимым с элементами карты. Исключение `NullPointerException` инициируется, если предпринимается попытка использовать `null`-объект, когда данная карта этого не допускает. Исключение `UnsupportedOperationException` генерируется при попытке изменить не модифицируемую карту.

Карты вращаются вокруг двух основных операций: `get()` и `put()`. Чтобы поместить значение в карту, используйте `put()`, указав ключ и значение. Чтобы получить значение, вызывайте `get()`, передавая ключ в качестве аргумента. Значение будет возвращено.

Как упоминалось ранее, несмотря на то, что карты являются частью каркаса коллекций, сами по себе они не реализуют интерфейс Collection. Однако вы можете получить представление карт в виде коллекций. Для этого можно воспользоваться методом entrySet(). Он возвращает Set, содержащий элементы карты. Чтобы получить коллекционное представление ключей, используйте keySet().

Таблица 17.13. Методы, определенные в Map

Метод	Описание
void clear()	Удаляет все пары "ключ-значение" из вызывающей карты.
boolean containsKey(Object *k*)	Возвращает true, если вызывающая карта содержит ключ *k*. В противном случае возвращает false.
boolean containsValue(Object *v*)	Возвращает true, если вызывающая карта содержит значение *v*. В противном случае возвращает false.
Set<Map.Entry<K, V>> entrySet()	Возвращает Set, содержащий все значения карты. Набор содержит объекты типа Map.Entry. То есть этот метод представляет карту в виде набора.
boolean equals(Object *obj*)	Возвращает true, если *obj* — это Map, содержащая одинаковые значения. В противном случае возвращает false.
V get(Object *k*)	Возвращает значение, ассоциированное с ключом *k*. Возвращает null, если ключ не найден.
int hashCode()	Возвращает хеш-код вызывающей карты.
boolean isEmpty()	Возвращает true, если вызывающая карта пуста. В противном случае возвращает false.
Set<K> keySet()	Возвращает Set, который содержит ключи вызывающей карты. Этот метод представляет ключи вызывающей карты в виде набора.
V put(K *k*, V *v*)	Помещает элемент в вызывающую карту, перезаписывая любое предшествующее значение, ассоциированное с ключом. Ключ и значение — это *k* и *v* соответственно. Возвращает null, если ключ ранее не существовал. В противном случае возвращается предыдущее значение, связанное с ключом.
void putAll(Map<? extends K, ? extends V> *m*)	Помещает все значения из *m* в карту.
V remove(Object *k*)	Удаляет элемент, чей ключ равен *k*.
int size()	Возвращает число пар "ключ-значение" в карте.
Collection<V> values()	Возвращает коллекцию, содержащую значения карты. Этот метод представляет значения, содержащихся в карте, в виде коллекции.

Чтобы получить коллекционное представление значений, используйте values(). Коллекционные представления — это средства, которыми карты интегрируются в большой каркас коллекций.

Интерфейс SortedMap

Интерфейс SortedMap расширяет Map. Он гарантирует, что элементы размещаются в возрастающем порядке значений ключей. SortedMap — обобщенный интерфейс, объявленный, как показано ниже:

```
interface SortedMap<K, V>
```

Здесь K указывает тип ключей, а V — тип хранимых значений.

Методы, объявленные в SortedMap, собраны в табл. 17.14. Некоторые методы инициируют исключение NoSuchElementException, когда вызывающая карта пуста. Исключение ClassCastException возбуждается, когда объект несовместим с элементами, хранящимися в карте. Исключение NullPointerException генерируется при попытке использовать объект null, когда null в данной карте не допускается. Исключение IllegalArgumentException возникает при использовании неверного аргумента.

Таблица 17.14. Методы, определенные в SortedMap

Метод	Значение
`Comparator<? super K> comparator()`	Возвращает компаратор вызывающей сортированной карты. Если картой используется естественный порядок, возвращается null.
`K firstKey()`	Возвращает первый ключ вызывающей карты.
`SortedMap<K, V> headMap(K end)`	Возвращает сортированную карту, содержащую те элементы вызывающей карты, ключ которых меньше *end*.
`K lastKey()`	Возвращает последний ключ вызывающей карты.
`SortedMap<K, V> subMap(K start, K end)`	Возвращает карту, содержащую элементы вызывающей карты, чей ключ больше или равен *start* и меньше *end*.
`SortedMap<K, V> tailMap(K start)`	Возвращает сортированную карту, содержащую те элементы вызывающей карты, ключ которых больше *start*.

Сортированные карты делают возможным очень эффективное манипулирование *подкартами* (другими словами, подмножествами карты). Чтобы получить подкарту, используйте headMap(), tailMap() или subMap(). Чтобы получить первый ключ набора, вызывайте firstKey(). Чтобы получить последний ключ — применяйте lastKey().

Интерфейс NavigableMap

Интерфейс NavigableMap был добавлен в Java SE 6. Она расширяет SortedMap и определяет поведение карты, поддерживающей извлечение элементов на основе ближайшего соответствия заданному ключу или ключам. NavigableMap — обобщенный интерфейс со следующим объявлением:

```
interface NavigableMap<K,V>
```

Здесь K специфицирует тип ключей, а V — тип значений, ассоциированных с ключами. В дополнение к методам, унаследованным от SortedMap, NavigableMap добавляет методы, перечисленные в табл. 17.15. Несколько методов возбуждают исключение ClassCastException, когда объект несовместим с ключами карты. Исключение NullPointerException возникает при попытке использования null-объекта, когда

null-ключи не допускаются в наборе. Исключение IllegalArgumentException генерируется при неправильном аргументе.

Таблица 17.15. Методы, определенные в NavigableMap

Метод	Значение
Map.Entry<K,V> ceilingEntry(K *obj*)	Выполняет поиск в карте наименьшего ключа *k*, такого, что *k*>=*obj*. Если такой ключ найден, возвращается соответствующее ему значение, в противном случае возвращается null.
K ceilingKey(K *obj*)	Выполняет поиск в карте наименьшего ключа *k*, такого, что *k*>=*obj*. Если такой ключ найден, он возвращается, в противном случае возвращается null.
NavigableSet<K> descendingKeySet()	Возвращает NavigableSet, содержащий ключи вызывающей карты в обратном порядке. То есть он возвращает обратное представление ключей. Результирующий набор основан на вызывающей карте.
NavigableMap<K,V> descendingMap()	Возвращает NavigableSet, обратный вызывающей карте.
Map.Entry<K,V> firstEntry()	Возвращает первое вхождение в карте. Это будет вхождение с минимальным ключом. Результирующий набор основан на вызывающей карте.
Map.Entry<K,V> floorEntry(K *obj*)	Выполняет поиск в карте наибольшего ключа *k*, такого, что *k*<=*obj*. Если такой ключ найден, возвращается соответствующее ему значение, в противном случае возвращается null.
K floorKey(K *obj*)	Выполняет поиск в карте наибольшего ключа *k*, такого, что *k*<=*obj*. Если такой ключ найден, он возвращается, в противном случае возвращается null.
NavigableMap<K,V> headMap(K *upperBound*, boolean *incl*)	Возвращает NavigableSet, включающий все вхождения вызывающей карты, имеющей ключи, меньшие *upperBound*. Если *incl* равно true, то элемент, равный *upperBound*, включается. Результирующий набор основан на вызывающей карте.
Map.Entry<K,V> higherEntry(K *obj*)	Выполняет поиск в наборе наибольшего ключа, такого, что *k*>*obj*. Если такой ключ найден, возвращается соответствующее ему значение. В противном случае возвращается null.
K higherKey(K *obj*)	Выполняет поиск в наборе наибольшего ключа, такого, что *k*>*obj*. Если такой ключ найден, он возвращается. В противном случае возвращается null.
Map.Entry<K,V> lastEntry()	Возвращает последнее вхождение в карте. Это будет значение с наибольшим ключом.
Map.Entry<K,V> lowerEntry(K *obj*)	Выполняет поиск в наборе наибольшего ключа, такого, что *k*<*obj*. Если такой ключ найден, возвращается соответствующее ему значение. В противном случае возвращается null.
K lowerKey(K *obj*)	Выполняет поиск в наборе наибольшего ключа, такого, что *k*<*obj*. Если такой ключ найден, он возвращается. В противном случае возвращается null.

Окончание табл. 17.15

Метод	Значение
`NavigableSet<K> navigableKeySet()`	Возвращает `NavigableSet`, содержащий ключи вызывающей карты. Результирующий набор основан на вызывающей карте.
`Map.Entry<K,V> pollFirstEntry()`	Возвращает первое вхождение, удаляя его в процессе. Поскольку карта сортирована, это будет вхождение с наименьшим ключом. При пустой карте возвращается `null`.
`Map.Entry<K,V> pollLastEntry()`	Возвращает последнее вхождение, удаляя его в процессе. Поскольку карта сортирована, это будет вхождение с наименьшим ключом. При пустой карте возвращается `null`.
`NavigableMap<K,V> subMap(K` *`lowerBound`*`, boolean` *`lowIncl`*`, K` *`upperBound`*`, boolean` *`highIncl`*`)`	Возвращает `NavigableSet`, включающий все вхождения вызывающей карты, которая имеет ключи, меньшие *`upperBound`* и большие *`lowerBound`*. Если *`lowincl`* равно `true`, то элемент, равный *`lowerBound`*, включается. Если *`highincl`* равно `true`, то элемент, равный *`upperBound`*, включается. Результирующий набор основан на вызывающей карте.
`NavigableMap<K,V> tailMap(K` *`lowerBound`*`, boolean` *`incl`*`)`	Возвращает `NavigableSet`, включающий все вхождения вызывающей карты, имеющей ключи, большие *`lowerBound`*. Если *`incl`* равно `true`, то элемент, равный *`lowerBound`*, включается. Результирующий набор основан на вызывающей карте.

Интерфейс `Map.Entry`

Интерфейс `Map.Entry` позволяет работать с элементом карты. Вспомните, что метод `entrySet()`, объявленный в интерфейсе `Map`, возвращает `Set`, содержащий элементы карты. Каждый из элементов этого набора представляет собой объект типа `Map.Entry` Интерфейс `Map.Entry` является обобщенным и объявлен следующим образом:

```
interface Map.Entry<K, V>
```

Здесь `K` указывает тип ключей, а `V` — тип хранимых значений. В табл. 17.16 перечислены методы, объявленные в `Map.Entry`.

Таблица 17.16. Методы, определенные в `Map.Entry`

Метод	Значение
`boolean equals(Object` *`obj`*`)`	Возвращает `true`, если *`obj`* — это `Map.Entry`, чей ключ и значение эквивалентны вызывающему объекту.
`K getKey()`	Возвращает ключ данного элемента карты.
`V getValue()`	Возвращает значение данного элемента карты.
`int hashCode()`	Возвращает хеш-код данного элемента карты.
`V setValue(V` *`v`*`)`	Устанавливает значение данного элемента карты равным *`v`*. Если *`v`* не относится к типу, допустимому для данной карты, генерируется исключение `ClassCastException`. `IllegalArgumentException` инициируется, если возникают проблемы с *`v`*. `NullPointerException` возбуждается, если *`v`* равно `null`, а карта не допускает хранения `null`-ключей. `UnsupportedOperationException` генерируется, если карта не может быть модифицирована.

Классы Map

Реализацию интерфейсов карт предлагают несколько классов. Классы, которые могут быть использованы для карт, перечислены в табл. 17.17.

Таблица 17.17. Классы, которые могут использоваться для карт

Класс	Описание
AbstractMap	Реализует большую часть интерфейса Map.
EnumMap	Расширяет AbstractMap для использования с ключами enum.
HashMap	Расширяет AbstractMap для использования хеш-таблицы.
TreeMap	Расширяет AbstractMap для использования дерева.
WeakHashMap	Расширяет AbstractMap для использования хеш-таблицы со слабыми ключами.
LinkedHashMap	Расширяет HashMap, разрешая итераторы в порядке вставки.
IdentityHashMap	Расширяет AbstractMap и использует проверку ссылочной эквивалентности при сравнении документов.

Следует отметить, что AbstractMap — это суперкласс для всех конкретных реализаций карт.

WeakHashMap реализует карту, которая использует "слабые ключи", что позволяет элементу карты быть объектом, подлежащим сборке мусора, когда его ключ никак не используется. Этот класс подробно здесь не обсуждается. Прочие классы карт описаны ниже.

Класс HashMap

Класс HashMap расширяет AbstractMap и реализует интерфейс Map. Он использует хеш-таблицу для хранения карты. Это позволяет обеспечить константное время выполнения методов get() и put() даже при больших наборах. HashMap — обобщенный класс со следующим объявлением:

```
class HashMap<K, V>
```

Здесь K указывает тип ключей, а V — тип хранимых значений.

В классе определены следующие конструкторы:

```
HashMap()
HashMap(Map<? extends K, ? extends V> m)
HashMap(int capacity)
HashMap(int capacity, float fillRatio)
```

Первая форма конструирует хеш-карту по умолчанию. Вторая форма инициирует хеш-карту элементами *m*. Третья форма инициирует емкость хеш-карты величиной *capacity*. Четвертая форма инициализирует и емкость, и отношение наполнения хеш-карты, используя аргументы конструктора. Значение емкости и отношения наполнения — то же самое, что и в HashSet, описанном ранее. Емкость по умолчанию — 16, коэффициент наполнения — 0,75.

HashMap реализует Map и расширяет AbstractMap. Он не добавляет никаких собственных методов.

Вы должны отметить, что хеш-карта не гарантирует порядка элементов. Таким образом, порядок, в котором элементы добавляются к хеш-карте, не обязательно повторяют

порядок, в котором они читаются итератором. В следующей программе иллюстрируется применение HashMap. Она отображает имена на балансовые счета. Обратите внимание на то, как получается и используется представление в виде набора.

```
import java.util.*;

class HashMapDemo {
public static void main(String args[]) {
  // Создать хеш-карту.
  HashMap<String, Double> hm = new HashMap<String, Double>();

  // Поместить элементы в карту
  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 набор элементов.
  Set<Map.Entry<String, Double>> set = hm.entrySet();

  // Отобразить набор.
  for(Map.Entry<String, Double> me : set) {
     System.out.print(me.getKey() + ": ");
     System.out.println(me.getValue());
  }
  System.out.println();

  // Добавить 1000 на свет Джона Доу.
  double balance = hm.get("Джон Доу");
  hm.put("Джон Доу ", balance + 1000);
  System.out.println("Новый баланс Джона Доу: " + hm.get("Джон Доу"));
}
}
```

Вывод этой программы показан здесь (точный порядок может отличаться):

```
Ральф Смит: -19.08
Том Смит: 123.22
Джон Доу: 3434.34
Тод Холл: 99.22
Джейн Бейкер: 1378.0
Новый баланс Джона Доу: 4434.34
```

Программа начинается с создания хеш-карты с последующим добавлением отображения имен на балансы. Далее содержимое карты отображается с применением представления карты в виде набора, полученного от entrySet(). Ключи и значения отображаются вызовом методов getKey() и getValue(), которые определены в Map.Entry. Обратите особое внимание на то, как депозит помещается на счет Джона Доу. Метод put() автоматически замещает любое предварительно существовавшее значение, ассоциированное с указанным ключом, новым значением. Таким образом, после того, как счет Джона Доу обновлен, хеш-карта по-прежнему содержит только один счет Джона Доу.

Класс TreeMap

Класс TreeMap расширяет AbstractMap и реализует интерфейс NavigableMap. Он создает карту, размещенную в древовидной структуре. TreeMap предлагает эффективный способ хранения пар "ключ-значение" в сортированном порядке и позволяет быстрое из-

влечение. Вы должны отметить, что в отличие от хеш-карты, карта-дерево (tree-map) гарантирует, что ее элементы будут отсортированы в порядке возрастания ключей. TreeMap является обобщенным классом со следующим объявлением:

```
class TreeMap<K, V>
```

Здесь K указывает тип ключей, а V — тип хранимых значений.

В TreeMap определены следующие конструкторы:

```
TreeMap()
TreeMap(Comparator<? super K> comp)
TreeMap(Map<? extends K, ? extends V> m)
TreeMap(SortedMap<K, ? extends V> sm)
```

Первая форма конструирует пустую карту-дерево, которая будет отсортирована с использованием естественного порядка ключей. Вторая форма конструирует пустую карту, основанную на дереве, которая будет отсортирована применением Comparator *comp*. (Компараторы будут обсуждаться далее в настоящей главе.) Третья форма инициализирует карту-дерево с элементами из *m*, которые будут отсортированы по естественному порядку ключей. Четвертая форма создает карту-дерево с элементами из *sm*, которые будут отсортированы в том же порядке, что и *sm*.

TreeMap не определяет дополнительных методов, помимо тех, что имеются в интерфейсе NavigableMap и классе AbstractMap.

В следующей программе предыдущий пример переделывается для использования TreeMap.

```
import java.util.*;

class TreeMapDemo {
public static void main(String args[]) {
  // Создать карту-дерево.
  TreeMap<String, Double> tm = new TreeMap<String, Double>();

  // Поместить элементы в карту.
  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 набор элементов.
  Set<Map.Entry<String, Double>> set = tm.entrySet();

  // Отобразить элементы.
  for(Map.Entry<String, Double> me : set) {
     System.out.print(me.getKey() + ": ");
     System.out.println(me.getValue());
  }
  System.out.println();

  // Добавить 1000 на счет Джона Доу.
  double balance = tm.get("Джон Доу");
  tm.put("Джон Доу", balance + 1000);
  System.out.println("Новый баланс Джона Доу: " +
  tm.get("Джон Доу"));
}
}
```

Вот как выглядит вывод программы:

```
Джейн Бейкер: 1378.0
Джон Доу: 3434.34
Ральф Смит: -19.08
Тод Халл: 99.22
Том Смит: 123.22

Текущий баланс Джона Доу: 4434.34
```

Обратите внимание, что `TreeMap` сортирует ключи. Однако в данном случае они сортируются по имени вместо фамилии. Вы можете изменить это поведение, указав компаратор при создании карты, как уже кратко упоминалось.

Класс LinkedHashMap

Класс `LinkedHashMap` расширяет `HashMap`. Он создает связный список элементов в карте, расположенных в том порядке, в котором они вставлялись. Это позволяет организовать итерацию по карте в порядке вставки. То есть, когда происходит итерация по коллекционному представлению `LinkedHashMap`, элементы будут возвращаться в том порядке, в котором они вставлялись. Вы также можете создать `LinkedHashMap`, возвращающий свои элементы в том порядке, в котором к ним в последний раз осуществлялся доступ. `LinkedHashMap` является обобщенным классом, имеющим следующее объявление:

```
class LinkedHashMap<K, V>
```

Здесь `K` указывает тип ключей, а `V` — тип хранимых значений.
`LinkedHashMap` определяет следующие конструкторы:

```
LinkedHashMap()
LinkedHashMap(Map<? extends K, ? extends V> m)
LinkedHashMap(int capacity)
LinkedHashMap(int capacity, float fillRatio)
LinkedHashMap(int capacity, float fillRatio, boolean Order)
```

Первая форма конструирует `LinkedHashMap` по умолчанию. Вторая форма инициализирует `LinkedHashMap` элементами *m*. Третья форма инициирует емкость. Четвертая форма инициализирует и емкость, и отношение наполнения. Смысл этих параметров тот же, что и у `HashMap`. Емкость по умолчанию составляет 16, коэффициент наполнения — 0,75. Последняя форма позволяет специфицировать, в каком порядке в связном списке будут размещаться элементы — в порядке вставки или порядке последнего доступа. Если *Order* равно `true`, используется порядок доступа. Если *Order* равно `false`, то используется порядок вставки.

`LinkedHashMap` добавляет только один новый метод к тем, что определены `HashMap`. Этот метод — `removeEldestEntry()` и он показан ниже:

```
protected boolean removeEldestEntry(Map.Entry<K, V> e)
```

Этот метод вызывается из `put()` и `putAll()`. Самый старый элемент передается в *e*. По умолчанию этот метод возвращает `false` и не делает ничего. Однако если вы переопределите этот метод, то сможете иметь `LinkedHashMap`, который удаляет самый старый элемент в карте. Чтобы сделать это, переопределенный метод должен возвращать `true`. Чтобы сохранять самый старый элемент, возвращайте `false`.

Класс IdentityHashMap

Класс IdentityHashMap расширяет AbstractMap и реализует интерфейс Map. Он похож на HashMap всем, за исключением того, что при сравнении элементов использует проверку эквивалентности ссылок. IdentityHashMap — это обобщенный класс со следующим объявлением:

```
class IdentityHashMap<K, V>
```

Здесь K указывает тип ключей, а V — тип хранимых значений. Документация по API явно устанавливает, что

IdentityHashMap не предназначен для общего применения.

Класс EnumMap

Класс EnumMap расширяет AbstractMap и реализует Map. Он специально предназначен для использования с ключами типа enum. Это обобщенный класс, имеющий следующее объявление:

```
class EnumMap<K extends Enum<K>, V>
```

Здесь K указывает тип ключей, а V — тип хранимых значений. Отметим, что K должен расширять Enum<K>, а это накладывает требование, чтобы ключи были типа enum.

В EnumMap определены следующие конструкторы:

```
EnumMap(Class<K> kType)
EnumMap(Map<K, ? extends V> m)
EnumMap(EnumMap<K, ? extends V> em)
```

Первый конструктор создает пустой EnumMap типа *kType*. Второй создает EnumMap, инициированный значениями из *em*. EnumMap не определяет собственных методов.

Компараторы

И 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 порядком", что, как правило, представляет собой тот порядок, которого вы можете ожидать (A перед B, 1 перед 2 и так далее). Если вы хотите упорядочить элементы иным образом, то указывайте объект Comparator при конструировании набора или карты. Это дает вам возможность тонко управлять тем, как элементы будут сохраняться в сортированных коллекциях или картах.

Comparator — обобщенный интерфейс со следующим объявлением:

```
interface Comparator<T>
```

Здесь T указывает тип сравниваемых объектов.

Интерфейс Comparator определяет два метода: compare() и equals(). Метод compare(), представленный ниже, сравнивает два элемента по порядку:

```
int compare(T obj1, T obj2)
```

Здесь *obj1* и *obj2* — это объекты, которые нужно сравнить. Этот метод возвращает ноль, если объекты эквивалентны. Он возвращает положительное значение, если *obj1* больше *obj2*, в противном случае возвращается отрицательное значение. Этот метод может возбуждать исключение ClassCastException, если типы сравниваемых объек-

тов не совместимы. Переопределяя `compare()`, вы можете изменить порядок объектов. Например, чтобы отсортировать в обратном порядке, можете создать компаратор, который возвращает обратные значения при сравнении.

Метод `equals()`, показанный ниже, проверяет объект на эквивалентность вызывающему компаратору:

```
boolean equals(Object obj)
```

Здесь *obj* — это объект, который нужно проверить на эквивалентность. Метод возвращает `true`, если *obj* и вызывающий объект представляют собой объекты типа `Comparator` и используют одинаковый способ упорядочения. В противном случае он возвращает `false`. Переопределение `equals()` не требуется, и большинство простых компараторов в этом не нуждаются.

Использование компараторов

Ниже представлен пример, демонстрирующий мощность настраиваемых компараторов. Он реализует метод `compare()` для строк, работающий в порядке, обратном нормальному. То есть он позволяет сохранить элементы набора-дерева в обратном порядке.

```
// Использование настраиваемого компаратора.
import java.util.*;

// Обратный компаратор для строк.
class MyComp implements Comparator<String> {
public int compare(String a, String b) {
  String aStr, bStr;
  aStr = a;
  bStr = b;
  // Обратное сравнение.
  return bStr.compareTo(aStr);
}
// Нет необходимости переопределять equals().
}

class CompDemo {
public static void main(String args[]) {
  // Создать TreeSet.
  TreeSet<String> ts = new TreeSet<String>(new MyComp());
  // Add elements to the tree set.
  ts.add("C");
  ts.add("A");
  ts.add("B");
  ts.add("E");
  ts.add("F");
  ts.add("D");

  // Отобразить элементы.
  for(String element : ts)
     System.out.print(element + " ");
  System.out.println();
}
}
```

Как показывает следующий вывод, дерево теперь отсортировано в обратном порядке:

```
F E D C B A
```

Взгляните внимательней на класс MyClass, который реализует Comparator и переопределяет compare(). (Как уже говорилось, переопределение equals() не является ни необходимым, ни желательным.) Внутри compare() метод compareTo() объекта String сравнивает две строки. Однако метод compareTo() вызывает bStr, а не aStr. Это приводит к тому, что результат сравнения получается обратным.

В качестве более полезного примера, следующая программа — усовершенствованная версия программы с TreeMap, сохраняющей балансовые счета. В предыдущей версии счета были отсортированы по владельцу счета, но порядок определялся именем. В следующей программе осуществляется сортировка счетов по фамилиям владельцев. Чтобы сделать это, она использует компаратор, сравнивающий фамилии каждого владельца счета. В результате получается карта, отсортированная по фамилиям.

```
// Использование компаратора для сортировки по фамилиям.
import java.util.*;

// Сравнивает последние два слова в полной строке.
class TComp implements Comparator<String> {
public int compare(String a, String b) {
  int i, j, k;
  String aStr, bStr;
  aStr = a;
  bStr = b;

  // Найти индекс символа в строке, с которого начинается фамилия.
  i = aStr.lastIndexOf(' ');
  j = bStr.lastIndexOf(' ');
  k = aStr.substring(i).compareTo(bStr.substring(j));
  if(k==0) // Фамилии совпадают, проверить полное имя
     return aStr.compareTo(bStr);
  else
     return k;
}
// Нет необходимости переопределять equals().
}

class TreeMapDemo2 {
public static void main(String args[]) {
  // Создать карту-дерево.
  TreeMap<String, Double> tm = new TreeMap<String, Double>(new TComp());

  // Поместить элементы в карту.
  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 набор элементов.
  Set<Map.Entry<String, Double>> set = tm.entrySet();

  // Отобразить элементы
  for(Map.Entry<String, Double> me : set) {
     System.out.print(me.getKey() + ": ");
     System.out.println(me.getValue());
  }

  System.out.println();
```

```
    // Добавить 1000 на счет Джона Доу.
    double balance = tm.get("Джон Доу ");
    tm.put("Джон Доу ", balance + 1000);
    System.out.println("Новый баланс Джона Доу: " + tm.get("John Doe"));
  }
  }
```

Вот вывод программы. Обратите внимание, что счета теперь сортированы по фамилиям:

```
Джейн Бейкер: 1378.0
Джон Доу: 3434.34
Ральф Смит: -19.08
Том Смит: 123.22
Тод Халл: 99.22

Новый баланс Джона Доу: 4434.34
```

Класс компаратора `TComp` сравнивает две строки, которые содержат имя и фамилию. Сначала он сравнивает фамилии. Чтобы сделать это, осуществляется поиск позиции последнего пробела в каждой строке и затем сравниваются подстроки, начинающиеся с этой позиции. В случае, когда фамилии эквивалентны, затем сравниваются имена. Это порождает карту-дерево, сортированную по фамилии, а в пределах одинаковых фамилий — по именам. Вы можете убедиться в этом, поскольку Ральф Смит в выводе программы находится перед Томом Смитом.

Алгоритмы коллекций

Каркас коллекций определяет несколько алгоритмов, которые могут быть применимы к коллекциям и картам. Эти алгоритмы определены как статические методы в классе `Collections`. Они собраны в таблице 17.18. Как объяснялось ранее, все алгоритмы были перепроектированы как обобщенные. Хотя обобщенный синтаксис может поначалу показаться несколько запутанным, алгоритмы настолько же просты в применении, как были ранее до его появления. К тому же теперь они стали безопасными в отношении типов.

Таблица 17.18. Алгоритмы, определенные в `Collections`

Метод	Описание
`static <T> boolean addAll(Collection <? super T> c, T ... elements)`	Вставляет элементы, переданные в *elements*, в коллекцию, указанную в *c*. Возвращает `true`, если элементы были добавлены, и `false` — в противном случае.
`static <T> Queue<T> asLifoQueue(Deque<T> c)`	Возвращает представление коллекции “последний вошел — первый вышел” (Добавлен в Java SE 6.)
`static <T> int binarySearch(List<? extends T> list, T value, Comparator<? super T> c)`	Ищет в *list* значение *value* в соответствии с *c*. Возвращает позицию *value* в *list*, или отрицательное значение, если *value* не найдено.
`static <T> int binarySearch(List<? Extends Comparable<? super T>> list, T value)`	Ищет в *list* значение *value*. Список должен быть сортирован. Возвращает позицию *value* в *list* или отрицательное значение, если *value* не найдено.
`static <E> Collection<E> checkedCollection( Collection<E> c, Class<E> t)`	Возвращает безопасное в отношении типов представление коллекции времени выполнения. Попытка вставить несовместимый элемент вызовет исключение `ClassCastException`.

Продолжение табл. 17.18

Метод	Описание
`static <E> List<E> checkedList(List<E> c, Class<E> t)`	Возвращает безопасное в отношении типов представление `List` времени выполнения. Попытка вставить несовместимый элемент вызовет исключение `ClassCastException`.
`static <K, V> Map<K, V> checkedMap(Map<K, V> c, Class<K> keyT, Class<V> valueT)`	Возвращает безопасное в отношении типов представление `Map` времени выполнения. Попытка вставить несовместимый элемент вызовет исключение `ClassCastException`.
`static <E> List<E> checkedSet(Set<E> c, Class<E> t)`	Возвращает безопасное в отношении типов представление `Set` времени выполнения. Попытка вставить несовместимый элемент вызовет исключение `ClassCastException`.
`static <K, V> SortedMap<K, V> checkedSortedMap(SortedMap<K, V> c, Class<K> keyT, Class<V> valueT)`	Возвращает безопасное в отношении типов представление `SortedMap` времени выполнения. Попытка вставить несовместимый элемент вызовет исключение `ClassCastException`.
`static <E> SortedSet<E> checkedSortedSet( SortedSet<E> c, Class<E> t)`	Возвращает безопасное в отношении типов представление `SortedSet` времени выполнения. Попытка вставить несовместимый элемент вызовет исключение `ClassCastException`.
`static <T> void copy(List<? super T> list1, List<? Extends T> list2)`	Копирует элементы *list2* в *list1*.
`static boolean disjoint(Collection<?> a, Collection<?> b)`	Сравнивает элементы в *a* с элементами в *b*. Возвращает `true`, если две коллекции не содержат общих элементов (т.е. непересекающиеся множества элементов). В противном случае возвращает `false`.
`static <T> List<T> emptyList()`	Возвращает неизменяемый, пустой объект `List` заданного типа.
`static <K, V> Map<K, V> emptyMap()`	Возвращает неизменяемый, пустой объект `Map` заданного типа.
`static <T> Set<T> emptySet()`	Возвращает неизменяемый, пустой объект `Set` заданного типа.
`static <T> Enumeration<T> enumeration(Collection<T> c)`	Возвращает перечисление на основе *c*. (См. раздел “Интерфейс `Enumeration`” далее в настоящей главе.)
`static <T> void fill(List<? super T> list, T obj)`	Присваивает *obj* каждому элементу *list*.
`static int frequency(Collection<?> c, Object obj)`	Подсчитывает количество вхождений *obj* в *c* и возвращает результат.
`static int indexOfSubList(List<?> list, List<?> subList)`	Ищет в *list* первое вхождение *subList*. Возвращает индекс первого совпадения или `-1`, если вхождение не найдено.

Продолжение табл. 17.18

Метод	Описание
`static int lastIndexOfSubList(List<?> list, List<?> subList)`	Ищет в `list` последнее вхождение `subList`. Возвращает индекс первого совпадения или `-1`, если вхождение не найдено.
`static <T> ArrayList<T> list(Enumeration<T> enum)`	Возвращает `ArrayList`, содержащий элементы `enum`.
`static <T> T max(Collection<? extends T> c, Comparator<? super T> comp)`	Возвращает максимальный элемент из `c`, определенный посредством `comp`.
`static <T extends Object &Comparable<? super T>> T max(Collection<? extends T> c)`	Возвращает максимальный элемент из `c`, определенный естественным порядком. Коллекция должна быть сортированной.
`static <T> T min(Collection<? extends T> c, Comparator<? super T> comp)`	Возвращает минимальный элемент из `c`, определенный посредством `comp`. Коллекция может быть несортированной.
`static <T extends Object & Comparable<? superT>> T min(Collection<? extends T> c)`	Возвращает минимальный элемент из `c`, определенный естественным порядком.
`static <T> List<T> nCopies(int num, T obj)`	Возвращает `num` копий `obj`, содержащихся в неизменяемом списке. `num` должно быть больше или равно нулю.
`static <E> Set<E> newSetFromMap(Map<E, Boolean> m)`	Создает и возвращает набор на основе `m`, который должен быть пустым на момент вызова метода. (Добавлен в Java SE 6.)
`static <T> boolean replaceAll(List<T> list, T old, T new)`	Заменяет все вхождения `old` на `new` в `list`. Возвращает `true`, если выполнена хотя бы одна замена. В противном случае возвращает `false`.
`static void reverse(List<T> list)`	Изменяет последовательность элементов в `list` на обратную.
`static <T> Comparator<T> reverseOrder(Comparator<T> comp)`	Возвращает компаратор, обратный переданному в `comp`. То есть, возвращенный компаратор порождает последовательность обратную той, что делает `comp`.
`static <T> Comparator<T> reverseOrder()`	Возвращает обратный компаратор, который обращает результат сравнения двух элементов.
`static void rotate(List<T> list, int n)`	Смещает `list` на `n` позиций вправо. Для смещения влево используйте отрицательное значение `n`.
`static void shuffle(List<T> list, Random r)`	Перемешивает (случайным образом) элементы в `list`, используя `r` в качестве источника случайных чисел.
`static void shuffle(List<T> list)`	Перемешивает (случайным образом) элементы в `list`.
`static <T> Set<T> singleton(T obj)`	Возвращает `obj`, как неизменяемый набор. Это простейший способ преобразовать отдельный объект в набор.

Окончание табл. 17.18

Метод	Описание
`static <T> List<T> singletonList(T obj)`	Возвращает *obj* как неизменяемый список. Это простейший способ преобразовать отдельный объект в список.
`static <K, V> Map<K, V> singletonMap(K k, V v)`	Возвращает пару "ключ-значение" *k/v* как неизменяемую карту. Это простейший способ преобразовать пару "ключ-значение" в карту.
`static <T> void sort(List<T> list, Comparator<? super T> comp)`	Сортирует элементы *list*, в соответствии с *comp*.
`static <T extends Comparable<? super T>> void sort(List<T> list)`	Сортирует элементы *list*, в соответствии с естественным порядком.
`static void swap(List<T> list, int idx1, int idx2)`	Меняет местами элементы *list*, находящиеся в позициях *idx1* и *idx2*.
`static <T> Collection<T> synchronizedCollection(Collection<T> c)`	Возвращает безопасную в отношении потоков коллекцию, наполненную элементами *c*.
`static <T> List<T> synchronizedList(List<T> list)`	Возвращает безопасный в отношении потоков список, наполненный элементами *list*.
`static <K, V> Map<K, V> synchronizedMap(Map<K, V> m)`	Возвращает безопасную в отношении потоков карту, наполненную элементами *m*.
`static <T> Set<T> synchronizedSet(Set<T> s)`	Возвращает безопасный в отношении потоков набор, наполненный элементами *s*.
`static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> sm)`	Возвращает безопасную в отношении потоков сортированную карту, наполненную элементами *sm*.
`static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> ss)`	Возвращает безопасный в отношении потоков сортированный набор, наполненный элементами *ss*.
`static <T> Collection<T> unmodifiable Collection(Collection<? extends T> c)`	Возвращает не модифицируемую коллекцию, наполненную элементами *c*.
`static <T> List<T> unmodifiableList(List<? extends T> list)`	Возвращает не модифицируемый список, наполненный элементами *list*.
`static <K, V> Map<K, V> unmodifiableMap(Map<? extends K, ? extends V> m)`	Возвращает не модифицируемую карту, наполненную элементами *m*.
`static <T> Set<T> unmodifiableSet(Set<? extends T> s)`	Возвращает не модифицируемый набор, наполненный элементами *s*.
`static <K, V> SortedMap<K, V> unmodifiableSortedMap(SortedMap<K, ? extends V> sm)`	Возвращает не модифицируемую сортированную карту, наполненную элементами *sm*.
`static <T> SortedSet<T> unmodifiableSortedSet(SortedSet<T> ss)`	Возвращает не модифицируемый сортированный набор, наполненный элементами *ss*.

Некоторые из этих методов могут возбуждать исключение ClassCastException, что происходит, когда предпринимается попытка сравнения несовместимых типов, либо UnsupportedOperationException, что происходит при попытке модифицировать не модифицируемые коллекции. В зависимости от метода возможны и другие исключения.

Особое внимание следует уделить набору перегруженных методов checked, таких как checkedCollection(), который возвращает то, что в документации по API именуется "динамическим представлением, безопасным в отношении типов" коллекций. Это представление представляет собой ссылку на коллекцию, которая во время выполнения отслеживает вставку объектов в коллекцию на предмет совместимости типов. Попытка вставить несовместимый элемент вызовет исключение ClassCastException. Использование этого представления полезно при отладке, так как гарантирует, что коллекция всегда содержит корректные элементы. Связанные с этим методы включают checkedSet(), checkedList(), checkedMap() и так далее. Они позволяют получить безопасное в отношении типов представление указанной коллекции.

Отметим, что несколько методов, такие как synchronizedList() и synchronizedSet(), служат для получения синхронизированных (безопасных в отношении потоков) копий различных коллекций. Как уже объяснялось, ни одна из стандартных реализаций коллекций не является синхронизированной. Для обеспечения синхронизации вы должны применять синхронизирующий алгоритм. И еще один момент: итераторы для синхронизированных коллекций должны использоваться в пределах блоков synchronized.

Набор методов, начинающихся с unmodifiable, возвращает представления различных коллекций, которые не могут быть модифицированы. Это может оказаться удобным, если вы хотите гарантировать доступ к коллекциям только по чтению, без права записи.

В Collection определены три статических переменных: EMPTY_SET, EMPTY_LIST и EMPTY_MAP. Все они являются константами.

В следующей программе демонстрируются некоторые алгоритмы. Программа создает и инициирует связный список. Метод reverseOrder() возвращает Comparator, который обращает сравнение объектов Integer. Элементы списка сортируются согласно этому компаратору, а затем отображаются. Далее этот список тасуется вызовом shuffle(), после чего отображаются его минимальное и максимальное значения.

```
// Демонстрация применения различных алгоритмов.
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 обратного порядка.
   Comparator<Integer> r = Collections.reverseOrder();

   // Сортировать список этим компаратором.
   Collections.sort(ll, r);

   System.out.print("Список отсортирован в обратном порядке: ");

   for(int i : ll)
     System.out.print(i+ " ");
```

```
    System.out.println();

    // Тасовать список.
    Collections.shuffle(ll);

    // Отобразить рандомизированный список.
    System.out.print("Список перемешан: ");

    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 вывод этой программы:

```
Список отсортирован в обратном порядке: 20 8 -8 -20
Список перемешан: 20 -20 8 -8
Минимум: -20
Максимум: 20
```

Обратите внимание, что `min()` и `max()` оперируют списком после того, как он был перетасован. Ни один из этих метолов не требует, чтобы список был отсортирован.

Arrays

Класс `Arrays` предоставляет разнообразные удобные методы для работы с массивами. Эти методы помогают заполнить пробел между коллекциями и массивами. Каждый метод, определенный в `Arrays`, рассматривается далее в разделе.

Метод `asList()` возвращает `List`, наполненный элементами указанного массива. Другими словами, и список, и массив ссылаются на одно и то же. Он имеет следующую сигнатуру:

```
static <T> List asList(T ... array)
```

Здесь *array* — массив, содержащий данные.

Метод `binarySearch()` использует бинарный поиск для нахождения заданного значения. Этот метод должен применяться к отсортированным массивам. Он имеет следующие формы:

```
static int binarySearch(byte array[ ], byte value)
static int binarySearch(char array[ ], char value)
static int binarySearch(double array[ ], double value)
static int binarySearch(float array[ ], float value)
static int binarySearch(int array[ ], int value)
static int binarySearch(long array[ ], long value)
static int binarySearch(short array[ ], short value)
static int binarySearch(Object array[ ], Object value)
static <T> int binarySearch(T[ ] array, T value, Comparator<? super T> c)
```

Здесь *array* — это массив, в котором осуществляется поиск, а *value* — значение, которое нужно найти. Последние две формы возбуждают исключение `ClassCastException`, если *array* содержит элементы, которые невозможно сравнивать (например, `Double` и `StringBuffer`), либо *value* несовместимо с типами *array*.

В последней форме `Comparator` *c* используется для определения порядка элементов в *array*. Во всех классах, если *value* содержится в *array*, возвращается индекс элемента. В противном случае возвращается отрицательное значение.

Метод `copyOf()` появился в Java SE 6. Он возвращает копию массива и имеет следующие формы:

```
static boolean[ ] copyOf(boolean[ ] source, int len)
static byte[ ] copyOf(byte[ ] source, int len)
static char[ ] copyOf(char[ ] source, int len)
static double[ ] copyOf(double[ ] source, int len)
static float[ ] copyOf(float[ ] source, int len)
static int[ ] copyOf(int[ ] source, int len)
static long[ ] copyOf(long[ ] source, int len)
static short[ ] copyOf(short[ ] source, int len)
static <T> T[ ] copyOf(T[ ] source, int len)
static <T,U> T[ ] copyOf(U[ ] source, int len, Class<? extends T[ ]> resultT)
```

Исходный массив специфицирован в *source*, а длина копии — в *len*. Если копия длиннее, чем *source*, она дополняется нулями (для числовых массивов), значениями `null` (для массивов объектов) или `false` (для булевских массивов). Если копия короче, чем *source*, она усекается. В последней форме тип *resultT* становится типом возвращаемого массива. Если *len* отрицательно, возбуждается исключение `NegativeArraySizeException`. Если *source* равно `null`, генерируется исключение `NullPointerException`. Если тип *resultT* не совместим с типом *source*, возникает исключение `ArrayStoreException`.

Метод `copyOfRange()` также появился в Java SE 6. Он возвращает копию диапазона внутри массива и имеет следующие формы:

```
static boolean[ ] copyOfRange(boolean[ ] source, int start, int end)
static byte[ ] copyOfRange(byte[ ] source, int start, int end)
static char[ ] copyOfRange(char[ ] source, int start, int end)
static double[ ] copyOfRange(double[ ] source, int start, int end)
static float[ ] copyOfRange(float[ ] source, int start, int end)
static int[ ] copyOfRange(int[ ] source, int start, int end)
static long[ ] copyOfRange(long[ ] source, int start, int end)
static short[ ] copyOfRange(short[ ] source, int start, int end)
static <T> T[ ] copyOfRange(T[ ] source, int start, int end)
static <T,U> T[ ] copyOfRange(U[ ] source, int start, int end,
                              Class<? extends T[ ]> resultT)
```

Исходный массив специфицирован в *source*. Диапазон для копирования специфицирован индексами, передаваемыми в *start* и *end*. Диапазон распространяется от *start* до *end*-1. Если диапазон длиннее, чем *source*, копия дополняется нулями (для числовых массивов), значениями `null` (для массивов объектов) или `false` (для булевских массивов). В последней форме тип *resultT* становится типом возвращаемого массива.

Если *start* отрицательно или больше длины *source*, возбуждается исключение `ArrayIndexOutOfBoundsException`.

Если *start* больше *end*, генерируется `IllegalArgumentException`. Если *source* равно `null`, генерируется `NullPointerException`. Если *resultT* не совместимо с типом *source*, возбуждается исключение `ArrayStoreException`.

Метод `equals()` возвращает `true`, если два массива эквивалентны. В противном случае он возвращает `false`.

Различные формы метода `equals()` показаны ниже.

```
static boolean equals(boolean array1[], boolean array2[])
static boolean equals(byte array1[], byte array2[])
static boolean equals(char array1[], char array2[])
static boolean equals(double array1[], double array2[])
static boolean equals(float array1[], float array2[])
static boolean equals(int array1[], int array2[])
static boolean equals(long array1[], long array2[])
static boolean equals(short array1[], short array2[])
static boolean equals(Object array1[], Object array2[])
```

Здесь *array1* и *array2* — массивы, сравниваемые на предмет эквивалентности.

Метод `deepEquals()` может быть использован для определения того, являются ли два массива, которые могут содержать вложенные массивы, эквивалентными. Он имеет следующее объявление:

```
static boolean deepEquals(Object[] a, Object[] b)
```

Метод возвращает `true`, если переданные ему массивы *a* и *b* эквивалентны. Если массивы *a* и *b* содержат вложенные массивы, они также сравниваются. Если массивы *a* и *b* либо их вложенные массивы отличаются, метод возвращает `false`.

Метод `fill()` присваивает значение всем элементам массива. Другими словами, он заполняет массив указанным значением. Метод `fill()` имеет две версии. Первая версия, формы которой представлен ниже, заполняет весь массив.

```
static void fill(boolean array[], boolean value)
static void fill(byte array[], byte value)
static void fill(char array[], char value)
static void fill(double array[], double value)
static void fill(float array[], float value)
static void fill(int array[], int value)
static void fill(long array[], long value)
static void fill(short array[], short value)
static void fill(Object array[], Object value)
```

Здесь *value* присваивается всем элементам *array*.

Вторая версия метода `fill()` присваивает значение подмножеству массива. Ее формы перечислены ниже.

```
static void fill(boolean array[], int start, int end, boolean value)
static void fill(byte array[], int start, int end, byte value)
static void fill(char array[], int start, int end, char value)
static void fill(double array[], int start, int end, double value)
static void fill(float array[], int start, int end, float value)
static void fill(int array[], int start, int end, int value)
static void fill(long array[], int start, int end, long value)
static void fill(short array[], int start, int end, short value)
static void fill(Object array[], int start, int end, Object value)
```

Здесь *value* присваивается элементам *array* от позиции *start* до *end*-1. Все эти методы могут возбуждать исключение `IllegalArgumentException`, если *start* больше *end*, либо `ArrayIndexOutOfBoundsException`, если *start* или *end* выходят за пределы массива.

Метод `sort()` сортирует массив таким образом, что он упорядочивается в возрастающем порядке. Метод `sort()` имеет две версии. Первая версия, показанная ниже, сортирует весь массив.

```
static void sort(byte array[])
static void sort(char array[])
static void sort(double array[])
static void sort(float array[])
static void sort(int array[])
static void sort(long array[])
static void sort(short array[])
static void sort(Object array[])
static <T> void sort(T array[], Comparator<? super T> c)
```

Здесь *array* — это массив, подлежащий сортировке. В последней форме *c* — это Comparator, который используется для упорядочения элементов *array*. Последние две формы могут инициировать исключение ClassCastException, если элементы сортируемого массива несовместимы.

Вторая версия sort() позволяет указать диапазон массива, который вы хотите сортировать. Ее формы представлены ниже.

```
static void sort(byte array[], int start, int end)
static void sort(char array[], int start, int end)
static void sort(double array[], int start, int end)
static void sort(float array[], int start, int end)
static void sort(int array[], int start, int end)
static void sort(long array[], int start, int end)
static void sort(short array[], int start, int end)
static void sort(Object array[], int start, int end)
static <T> void sort(T array[], int start, int end, Comparator<? super T> c)
```

Здесь будет отсортирован диапазон элементов массива, начинающийся со *start* и заканчивающийся *end*-1. В последней форме *c* — это Comparator, который используется для определения порядка элементов массива.

Все эти методы могут инициировать исключение IllegalArgumentException, если *start* больше *end*, либо ArrayIndexOutOfBoundsException, если *start* или *end* выходят за пределы массива. Последние две формы также могут возбуждать исключение ClassCastException, если сортируемые элементы массива не совместимы.

В Arrays также переопределены методы toString() и hashCode() для различных типов массивов. Вдобавок в нем предусмотрены методы deepToString() и deepHashCode(), которые эффективно работают с массивами, имеющими вложенные массивы.

В следующей программе иллюстрируется применение некоторых методов класса Arrays.

```
// Демонстрация применения Arrays.
import java.util.*;

class ArraysDemo {
public static void main(String args[]) {
  // Распределить и инициировать массива.
  int array[] = new int[10];
  for(int i = 0; i < 10; i++)
     array[i] = -3 * i;

  // Отобразить, отсортировать и вновь отобразить массив.
  System.out.print("Исходное содержимое: ");
  display(array);
  Arrays.sort(array);
  System.out.print("Отсортированный массив: ");
  display(array);
```

```
  // Наполнение и отображение массива.
  Arrays.fill(array, 2, 6, -1);
  System.out.print("После fill(): ");
  display(array);

  // Сортировать и отобразить массив.
  Arrays.sort(array);
  System.out.print("После повторной сортировки: ");
  display(array);

  // Бинарный поиск значения -9.
  System.out.print("Значение -9 находится в 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 показан вывод этой программы.

```
Исходное содержимое: 0 -3 -6 -9 -12 -15 -18 -21 -24 -27
Отсортированный массив: -27 -24 -21 -18 -15 -12 -9 -6 -3 0
После fill(): -27 -24 -1 -1 -1 -1 -9 -6 -3 0
После повторной сортировки: -27 -24 -9 -6 -3 -1 -1 -1 -1 0
Значение -9 находится в позиции 2
```

Зачем нужны обобщенные коллекции?

Как уже упоминалось в начале этой главы, в JDK 5 каркас коллекций был перепроектирован для учета обобщений. Более того, каркас коллекций — возможно, единственное наиболее важное применение обобщений в Java API. Причина этого в том, что обобщения обеспечивают безопасность типов каркаса коллекций. Прежде чем двигаться дальше, стоит потратить некоторое время на детальное рассмотрение сущности этого усовершенствования.

Давайте начнем с примера, который использует старый код, до введения обобщений. В следующей программе список строк сохраняется в `ArrayList`, а затем отображается содержимое списка.

```
// Пример использования коллекций до введения обощений.
import java.util.*;
class OldStyle {
public static void main(String args[]) {
  ArrayList list = new ArrayList();
  // lines хранит строки, но могут быть сохранены объекты любого типа. В коде
  // старого стиля нет возможности защитить тип сохраняемых в коллекции объектов.
  list.add("один");
  list.add("два");
  list.add("три");
  list.add("четыре");
  Iterator itr = list.iterator();
```

```
    while(itr.hasNext()) {
      // Чтобы извлечь элемент, требуется явное приведение типов,
      // потому что коллекции хранят только Object.
      String str = (String) itr.next(); // здесь необходимо явное приведение.
      System.out.println(str + " имеет длину " + str.length() + " символов.");
    }
  }
  }
```

До введения обобщенного синтаксиса все коллекции хранили ссылки на объекты типа `Object`. Это позволяло сохранять в них ссылки на объекты любого типа. Приведенная выше программа использует это свойство для хранения в `list` объектов типа `String`, но также может сохранять ссылки любого другого типа.

К сожалению, тот факт, что коллекции старого вида сохраняют только объекты типа `Object`, может легко приводить к ошибкам. Во-первых, это требует, чтобы вы, а не компилятор, обеспечивали сохранение в конкретной коллекции только объектов правильных типов. Например, в предыдущем примере `list` однозначно предназначен для сохранения объектов `String`, но нет ничего, что действительно предотвратило бы добавление в коллекцию ссылок на объекты других типов. Так, например, компилятор не обнаружит ничего недопустимого в следующей строке кода:

```
list.add(new Integer(100));
```

Поскольку `list` сохраняет ссылки на `Object`, он может сохранить ссылку на `Integer` — точно так же, как и ссылку на `String`. Однако если вы предполагаете хранить в `list` только строки, то предыдущий оператор повредит вашу коллекцию. Опять-таки компилятор не имеет никакой возможности узнать, что этот оператор неверен.

Вторая проблема со старыми (до введения обобщений) коллекциями в том, что когда вы получаете ссылку из коллекции, то должны вручную явно приводить эту ссылку к правильному типу. Вот почему предыдущая программа приводит ссылку, полученную от `next()`, к типу `String`. До введения обобщений коллекции просто сохраняли ссылки на `Object`. Поэтому при извлечении объектов из коллекций необходимо было приведение типа.

Помимо неудобств, связанных с необходимостью приведения типов ссылок, этот недостаток информации о типе объектов приводит к довольно серьезным, но неожиданно легко порождаемым ошибкам. Поскольку `Object` может быть приведен к любому объектному типу, всегда существовала возможность привести полученную из коллекции ссылку к *неверному типу*. Например, если к предыдущему примеру добавить следующий оператор, пример будет по-прежнему компилироваться без ошибок, но генерировать исключение времени выполнения во время запуска программы:

```
Integer i = (Integer) itr.next();
```

Вспомните, что в предыдущем примере в `list` сохранялись только ссылки на экземпляры `String`. Поэтому при попытке приведения `String` к `Integer` возникнет исключение неверного приведения типов. Так как это случится во время выполнения программы, подобная ошибка весьма серьезна.

Добавление обобщений фундаментально усовершенствует удобство использования и безопасность коллекций, и причины указаны ниже.

- Гарантирует, что в коллекции будут сохранены только ссылки на объекты правильного типа. То есть коллекции всегда будут содержать ссылки известного типа.

- Исключает необходимость приведения типов ссылок, извлеченных из коллекции. Вместо этого ссылка, извлеченная из коллекции, будет автоматически приведена к правильному типу. Это предотвращает ошибки времени выполнения по причине неверного приведения и позволяет избежать целой категории подобных ошибок.

Эти два усовершенствования стали возможны благодаря тому, что каждому классу коллекций может быть передан параметр, специфицирующий тип коллекции. Например, ArrayList теперь объявляется следующим образом:

```
class ArrayList<E>
```

Здесь E — тип элемента, сохраняемого в коллекции. Таким образом, следующая строка объявляет ArrayList для объектов String:

```
ArrayList<String> list = new ArrayList<String>();
```

Теперь в list могут быть добавлены только ссылки на объекты типа String.

Интерфейсы Iterator и ListIterator теперь также стали обобщенными. Это означает, что параметр-тип должен быть согласован с типом коллекции, для которой получен итератор. Более того, эта совместимость типов принудительно обеспечивается во время компиляции.

Следующая программа показывает современную, обобщенную форму программы предыдущей:

```
// Современная, обобщенная версия.
import java.util.*;

class NewStyle {
public static void main(String args[]) {
  // Теперь list содержит ссылки типа String.
  ArrayList<String> list = new ArrayList<String>();
  list.add("один");
  list.add("два");
  list.add("три");
  list.add("четыре");

  // Отметим, что Iterator также обобщенный.
  Iterator<String> itr = list.iterator();

  // Следующий оператор теперь вызовет ошибку времени компиляции.
  // Iterator<Integer> itr = list.iterator(); // Ошибка!
  while(itr.hasNext()) {
     String str = itr.next(); // приведение не требуется
     // Теперь следующая строка породит ошибку компиляции, а не времени
     выполнения
     // Integer i = itr.next(); // Это не откомпилируется
     System.out.println(str + " имеет длину " + str.length() + " символов.");
  }
}
}
```

Теперь list может содержать только ссылки на объекты типа String. Более того, как видно из следующей строки, нет необходимости приводить тип объекта, возвращаемого от next() к типу String:

```
String str = itr.next(); //приведение не нужно
```

Приведение выполняется автоматически.

Так как существует поддержка raw-типов, нет необходимости немедленно обновлять ваш старый код, работающий с коллекциями. Однако весь новый код должен быть обобщенным, а старый вы можете обновлять постепенно, при наличии свободного времени. Добавление обобщений в каркас коллекций — фундаментальное усовершенствование, которое следует использовать везде, где это возможно.

Унаследованные классы и интерфейсы

Как уже объяснялось в начале настоящей главы, ранние версии `java.util` не включали в себя каркас коллекций. Взамен там были определены несколько классов и интерфейсов, обеспечивающих специальные методы хранения объектов. Когда были добавлены коллекции (начиная с J2SE 1.2), некоторые из исходных классов были перепроектированы для поддержки интерфейсов коллекций. То есть они полностью совместимы с каркасом (коллекций). Поскольку никакие классы не были объявлены нежелательными (deprecated), они просто могут рассматриваться как устаревшие. Конечно, там, где коллекции дублируют функциональность унаследованных классов, вы обычно будете в новом коде применять коллекции. Вообще говоря, унаследованные классы поддерживаются потому, что существует код, использующий их. Конечно, вы можете легко синхронизировать коллекции, используя один из алгоритмов, представленных классом `Collection`. Унаследованные классы, определенные в `java.util`, показаны ниже.

- `Dictionary`
- `Hashtable`
- `Properties`
- `Stack`
- `Vector`

Есть также один унаследованный интерфейс, называемый `Enumeration`. Следующие разделы рассматривают `Enumeration` и каждый из унаследованных классов по очереди.

Интерфейс `Enumeration`

Интерфейс `Enumeration` определяет методы, которыми вы можете перечислить (получая по одному за раз) элементы в коллекции объектов. Этот унаследованный интерфейс был замещен `Iterator`. Хотя `Enumeration` и не является не рекомендованным, но считается устаревшим для нового кода. Однако он используется несколькими методами унаследованных классов (таких как `Vector` или `Properties`), также некоторыми другими классами API, и широко используется существующим кодом приложений. Поскольку он все еще задействован, он был перепроектирован в обобщенном виде в JDK 5. Он имеет следующее объявление:

```
interface Enumeration<E>
```

Здесь `E` специфицирует тип элементов, которые будут перечисляться.

В `Enumeration` определены следующие два метода:

```
boolean hasMoreElements()
E nextElement()
```

При реализации hasMoreElements() должен возвращать true до тех пор, пока остаются элементы, подлежащие извлечению, и false — когда все элементы уже перечислены. nextElement() возвращает следующий объект перечисления. То есть каждый вызов nextElement() получает следующий объект перечисления. По завершении прохода по перечислению этот метод генерирует исключение NoSuchElementException.

Vector

Класс Vector реализует динамический массив. Он подобен ArrayList, но с двумя отличиями: Vector синхронизирован и включает много унаследованных методов, не являющихся частью каркаса коллекций. С появлением коллекций Vector был перепроектирован как расширение AbstractList, и в него была добавлена реализация интерфейса List. В версии JDK 5 он был перепроектирован под применение обобщенного синтаксиса, и в нем появилась реализация интерфейса Iterable. Это означает, что Vector стал полностью совместимым с коллекциями и может выдавать свое содержимое в усовершенствованном цикле for.

Vector объявлен следующим образом:

```
class Vector<E>
```

Здесь E специфицирует тип элементов, которые будут храниться.

Ниже перечислены конструкторы Vector.

```
Vector()
Vector(int size)
Vector(int size, int incr)
Vector(Collection<? extends E> c)
```

Первая форма создает вектор по умолчанию, имеющий начальный размер 10. Вторая форма создает вектор, чья начальная емкость равна *size*. Третья форма создает вектор с начальной емкостью *size*, а инкремент указан в *incr*. Инкремент задает количество элементов, которые будут размещаться каждый раз, когда размер вектора будет увеличиваться. Четвертая форма создает вектор, содержащий элементы коллекции *c*.

Все векторы начинаются с некоторой начальной емкости. Когда эта емкость заполнена, при следующей попытке сохранения объекта в векторе он автоматически увеличивает количество выделенного пространства. Это увеличение важно, поскольку распределение памяти — дорогостоящая по времени операция. Общий объем дополнительного пространства памяти, выделенного при каждом перераспределении, задается величиной инкремента, указанного при создании вектора. Если вы не указываете размер инкремента, размер вектора удваивается при каждом цикле распределения памяти.

В Vector определены следующие защищенные данные-члены:

```
int capacityIncrement;
int elementCount;
Object[] elementData;
```

Значение инкремента сохраняется в capacityIncrement. Количество элементов, находящихся в данный момент в векторе, хранится в elementCount. Массив, хранящий сам вектор, содержится в elementData.

В дополнение к методам коллекций, определенных в List, Vector определяет несколько унаследованных методов, которые перечислены в табл. 17.19.

Таблица 17.19. Унаследованные методы, определенные в `Vector`

Метод	Описание
`void addElement(E element)`	Объект, указанный в *element*, добавляется к вектору.
`int capacity()`	Возвращает емкость вектора.
`Object clone()`	Возвращает дубликат вызывающего объекта.
`boolean contains(Object element)`	Возвращает `true`, если *element* содержится в `vector`, и `false` — в противном случае.
`void copyInto(Object array[])`	Элементы, содержащиеся в вызывающем векторе, копируются в массив *array*.
`E elementAt(int index)`	Возвращает элемент, расположенный в позиции *index*.
`Enumeration<E> elements()`	Возвращает перечисление элементов вектора.
`void ensureCapacity(int size)`	Устанавливает минимальную емкость вектора равной *size*.
`E firstElement()`	Возвращает первый элемент вектора.
`int indexOf(Object element)`	Возвращает индекс первого вхождения *element*. Если объект не найден в векторе, возвращает `-1`.
`int indexOf(Object element, int start)`	Возвращает индекс первого вхождения *element* после *start*. Если объект не найден в векторе, возвращает `-1`.
`void insertElementAt(E element, int index)`	Добавляет *element* к вектору в позицию *index*.
`boolean isEmpty()`	Возвращает `true`, если вектор пуст и `false` — в противном случае.
`E lastElement()`	Возвращает последний элемент вектора.
`int lastIndexOf(Object element)`	Возвращает индекс последнего вхождения *element*. Если объект не найден в векторе, возвращает `-1`.
`int lastIndexOf(Object element, int start)`	Возвращает индекс последнего вхождения *element* перед *start*. Если объект не найден в векторе, возвращает `-1`.
`void removeAllElements()`	Очищает вектор. После выполнения этого метода размер вектора равен нулю.
`boolean removeElement(Object element)`	Удаляет *element* из вектора. Если в векторе содержится более одного экземпляра данного элемента, то удаляется только первый из них. Возвращает `true`, если элемент удален, и `false` — если объект не найден.
`void removeElementAt(int index)`	Удаляет из вектора *element*, расположенный в позиции *index*.
`void setElementAt(E element, int index)`	Устанавливается значение элемента в позиции *index*.
`void setSize(int size)`	Устанавливается количество элементов вектора в *size*. Если новый размер меньше старого, элементы теряются. Если же новый размер больше старого, добавляются `null`-элементы.
`int size()`	Возвращает количество элементов, содержащихся в векторе.
`String toString()`	Возвращает строковый эквивалент значения вектора.
`void trimToSize()`	Устанавливает емкость вектора равной количеству элементов, содержащихся в нем на данный момент.

Поскольку Vector реализует List, вы можете использовать его так же, как вы применяете экземпляр ArrayList. Вы можете также манипулировать им, используя один из унаследованных методов. Например, после создания экземпляра Vector вы можете добавлять элементы к нему с помощью addElement(). Чтобы получить элемент, расположенный в определенной позиции, вызывайте elementAt(). Чтобы получить первый элемент вектора, применяйте firstElement(), а чтобы последний — lastElement(). Индекс определенного элемента можно получить методом indexOf() или lastIndexOf(). Чтобы удалить элемент вектора, вызывайте removeElement() или removeElementAt().

В следующей программе вектор используется для сохранения разных типов числовых объектов. В ней демонстрируется несколько унаследованных методов, определенных в Vector. Кроме того, в программе показана работа с интерфейсом Enumeration.

```
// Демонстрация различных операций с Vector.
import java.util.*;
class VectorDemo {
public static void main(String args[]) {
  // Начальный размер 3, инкремент 2
  Vector<Integer> v = new Vector<Integer>(3, 2);
  System.out.println("Начальный размер: " + v.size());
  System.out.println("Начальная емкость: " + v.capacity());
  v.addElement(1);
  v.addElement(2);
  v.addElement(3);
  v.addElement(4);
  System.out.println("Емкость после четырех добавлений: " + v.capacity());
  v.addElement(5);
  System.out.println("Текущая емкость: " + v.capacity());
  v.addElement(6);
  v.addElement(7);
  System.out.println("Текущая емкость: " + v.capacity());
  v.addElement(9);
  v.addElement(10);
  System.out.println("Текущая емкость: " + v.capacity());
  v.addElement(11);
  v.addElement(12);
  System.out.println("Первый элемент: " + v.firstElement());
  System.out.println("Последний элемент: " + v.lastElement());
  if(v.contains(3))
     System.out.println("Вектор содержит 3.");
  // Перечисление элементов вектора.
  Enumeration vEnum = v.elements();
  System.out.println("\nЭлементы вектора:");
  while(vEnum.hasMoreElements())
     System.out.print(vEnum.nextElement() + " ");
  System.out.println();
}
}
```

Вывод этой программы приведен ниже:

```
Начальный размер: 0
Начальная емкость: 3
Емкость после четырех добавлений: 5
Текущая емкость: 5
Текущая емкость: 7
```

```
Текущая емкость: 9
Первый элемент: 1
Последний элемент: 12
Вектор содержит 3.
Элементы вектора:
1 2 3 4 5 6 7 9 10 11 12
```

Вместо того чтобы полагаться на перечисление объектов в цикле (как это делает приведенная выше программа), вы можете использовать итератор. Например, в программу можно поместить следующий итеративный код:

```
// Использовать итератор для отображения содержимого
Iterator<Integer> vItr = v.iterator();
System.out.println("\nЭлементы вектора:");
while(vItr.hasNext())
   System.out.print(vItr.next() + " ");
System.out.println();
```

Вы можете пройти по вектору циклом “for-each”, как показано в следующей версии предыдущего кода:

```
// Использовать расширение цикла for для отображения элементов.
System.out.println("\nЭлементы вектора:");
for(int i : v)
   System.out.print(i + " ");
System.out.println();
```

Поскольку интерфейс `Enumeration` не рекомендуется применять в новом коде, обычно вы будете использовать итераторы и циклы “for-each” для прохода по всем элементам вектора. Конечно, существует большой объем кода, использующего `Enumeration`. К счастью, перечисления и итераторы работают почти одинаково.

Stack

`Stack` — это подкласс `Vector`, который реализует стандартный стек “последний вошел — первый вышел”. `Stack` определяет только конструктор по умолчанию, создающий пустой стек. С появлением версии JDK 5 подкласс `Stack` был перепроектирован под обобщенный синтаксис, и теперь он объявлен следующим образом:

```
class Stack<E>
```

Здесь `E` указывает тип элементов, сохраняемых в стеке.

`Stack` включает все методы, определенные в `Vector`, и добавляет некоторые свои, которые перечислены в таблице 17.20.

Таблица 17.20. Методы, определенные в `Stack`

Метод	Описание
`boolean empty()`	Возвращает `true`, если стек пустой и `false`, если он содержит элементы.
`E peek()`	Возвращает элемент с вершины стека, но не удаляет его.
`E pop()`	Возвращает элемент с вершины стека, удаляя его.
`E push(E element)`	Вталкивает *`element`* в стек. *`element`* также возвращается.
`int search(Object element)`	Ищет *`element`* в стеке. Если *`element`* найден, возвращает смещение от вершины стека до этого элемента. В противном случае возвращает `-1`.

Чтобы поместить объект в верхушку стека, вызывайте push(). Чтобы удалить и вернуть верхний элемент, вызывайте pop(). Исключение EmptyStackException генерируется, если применить pop() к пустому стеку. Вы можете использовать peek() для возврата верхнего объекта без его удаления. Метод empty() возвращает true, если стек пуст. Метод search() определяет, содержится ли объект в стеке, и возвращает количество операций pop(), необходимых для перемещения его в вершину стека.

Ниже приведен пример, который создает стек, вставляет в него несколько объектов Integer, а затем извлекает их обратно.

```
// Демонстрация применения 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 представлен вывод программы: обратите внимание на вызов обработчика исключений для EmptyStackException, позволяющего успешно обработать ситуацию с пустым стеком.

```
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
```

Отметим еще один момент: хотя `Stack` в версии Java SE 6 устаревшим не считается, лучшим выбором будет `ArraDeaue`.

Dictionary

`Dictionary` — это абстрактный класс, представляющий репозиторий для хранения пар "ключ-значение" и работающий в основном подобно `Map`. Передав ключ и значение, вы можете сохранить значение в объекте `Dictionary`. Однажды сохраненное значение можно извлечь по его ключу. То есть, подобно карте, `Dictionary` (словарь) можно считать списком пар "ключ-значение". Хотя пока `Dictionary` не объявлен нежелательным, его можно рассматривать как устаревший, поскольку его полностью заменяет `Map`. Однако `Dictionary` все еще применяется, поэтому мы опишем его здесь.

С появлением JDK 5 класс `Dictionary` был также сделан обобщенным. Он объявлен следующим образом:

```
class Dictionary<K, V>
```

Здесь `K` указывает тип ключей, а `V` — тип значений. Абстрактные методы, определенные в `Dictionary`, перечислены в таблице 17.21.

Таблица 17.21. Методы, определенные в `Dictionary`

Метод	Описание
`Enumeration<V> elements()`	Возвращает перечисление значений, хранимых в словаре.
`V get(Object key)`	Возвращает объект, содержащий значение, ассоциированное с *key*. Если *key* не содержится в словаре, возвращается `null`.
`boolean isEmpty()`	Возвращает `true`, если словарь пуст, и `false`, если он содержит хотя бы одно значение.
`Enumeration<K> keys()`	Возвращает перечисление ключей, хранимых в словаре.
`V put(K key, V value)`	Вставляет ключ и значение в словарь. Возвращает `null`, если *key* еще не содержится в словаре, в противном случае возвращает предыдущее значение, ассоциированное с *key*.
`V remove(Object key)`	Удаляет *key* и его значение из словаря. Возвращает значение, ассоциированное с *key*. Если *key* не содержится в словаре, возвращает `null`.
`int size()`	Возвращает количество элементов в словаре.

Чтобы добавить ключ и его значение в словарь, используйте метод `put()`. Вызывайте `get()` для извлечения значения по заданному ключу. Ключи и значения могут быть возвращены как `Enumertaion` методами `keys()` и `elements()` соответственно. Метод `size()` возвращает количество пар "ключ-значение", сохраненных в словаре, а `isEmpty()` возвращает `true`, если словарь пуст. Для удаления любой пары "ключ-значение" можно применять метод `remove()`.

Помните! *Класс* `Dictionary` *устарел. Вы должны реализовывать интерфейс* `Map` *для получения функциональности хранения пар "ключ-значение".*

Hashtable

`Hashtable` — это часть исходного пакета `java.util` и конкретная реализация `Dictionary`. Однако с появлением коллекций класс `Hashtable` был перепроектирован с тем, чтобы также реализовывать интерфейс `Map`. То есть `Hashtable` теперь интегрирован в каркас коллекций. Он подобен `HashMap`, но синхронизирован.

Подобно `HashMap`, `Hashtable` сохраняет пары "ключ-значение" в хеш-таблице. Однако ни ключи, ни значения не могут быть равны `null`. При использовании `Hashtable` вы указываете объект, который служит ключом, и значение, которое вы хотите связать с этим ключом. Ключ затем хешируется, а результирующий хеш-код используется в качестве индекса, по которому значение сохраняется в таблице.

`Hashtable` был сделан обобщенным в JDK 5. Он объявлен следующим образом:

```
class Hashtable<K, V>
```

Здесь `K` указывает тип ключей, а `V` — тип значений.

Хеш-таблица может только сохранять объекты, которые переопределяют методы `hashCode()` и `equals()`, определенные в `Object`. Метод `hashCode()` должен вычислять и возвращать хеш-код объекта. Конечно, `equals()` должен сравнивать два объекта. К счастью, многие из встроенных классов Java уже реализуют метод `hashCode()`. Так, например, наиболее общий тип `Hashtable` использует в качестве ключа объект `String`. `String` реализует и `hashCode()`, и `equals()`.

Конструкторы `Hashtable` показаны ниже:

```
Hashtable()
Hashtable(int size)
Hashtable(int size, float fillRatio)
Hashtable(Map<? extends K, ? extends V> m)
```

Первая версия — конструктор по умолчанию. Вторая версия создает хеш-таблицу, имеющую начальный размер, указанный в *`size`*. (Размер по умолчанию — 11.) Третья версия создает хеш-таблицу с начальным размером *`size`* и коэффициентом наполнения, заданным в *`fillRatio`*. Этот коэффициент лежит в пределах от 0,0 до 1,0 и определяет, насколько полной должна быть таблица, прежде чем она будет расширена. Точнее, когда число элементов превышает емкость, умноженную на коэффициент наполнения, хеш-таблица расширяется. Если вы не указываете коэффициент наполнения, используется значение по умолчанию 0,75. И, наконец, четвертая версия создает хеш-таблицу, инициализированную элементами из коллекции *`m`*. Емкость хеш-таблицы устанавливается вдвое больше, чем размер *`m`*. Используется фактор загрузки по умолчанию, равный 0,75.

В дополнение к методам, определенным в интерфейсе `Map`, который теперь реализует `Hashtable`, `Hashtable`, также определяет унаследованные методы, перечисленные в табл. 17.22. Некоторые методы при попытке использования значения `null` в качестве ключа возбуждают исключение `NullPointerException`.

Таблица 17.22. Унаследованные методы, определенные в `Hashtable`

Метод	Описание
`void clear()`	Сбрасывает и очищает хеш-таблицу.
`Object clone()`	Возвращает дубликат вызывающего объекта.
`boolean contains(Object value)`	Возвращает `true`, если некоторое значение, эквивалентное *value* существует в хеш-таблице. Возвращает `false`, если значение не найдено.
`boolean containsKey(Object key)`	Возвращает `true`, если некоторый ключ, эквивалентный *key* существует в хеш-таблице. Возвращает `false`, если ключ не найден.
`boolean containsValue(Object value)`	Возвращает `true`, если некоторое значение, эквивалентное *value* существует в хеш-таблице. Возвращает `false`, если значение не найдено.
`Enumeration<V> elements()`	Возвращает перечисление значений, содержащихся в хеш-таблице.
`V get(Object key)`	Возвращает объект, который содержит значение, ассоциированное с *key*. Если *key* не найден в хеш-таблице, возвращается `null`.
`boolean isEmpty()`	Возвращает `true`, если хеш-таблица пуста. Возвращает `false`, если она содержит хоть одно значение.
`Enumeration<K> keys()`	Возвращает перечисление ключей, содержащихся в хеш-таблице.
`V put(K key, V value)`	Вставляет в хеш-таблицу ключ и значение. Возвращает `null`, если *key* не был найден в таблице на момент вставки. В противном случае возвращает предыдущее значение, ассоциированное с этим ключом.
`void rehash()`	Увеличивает размер хеш-таблицы и повторно хеширует все ее ключи.
`V remove(Object key)`	Удаляет *key* из хеш-таблицы. Возвращает значение, ассоциированное с *key*. Если *key* не найден в хеш-таблице, возвращается `null`.
`int size()`	Возвращает количество элементов в хеш-таблице.
`String toString()`	Возвращает строковый эквивалент хеш-таблицы.

Следующий пример представляет переработанную версию программы управления банковскими счетами, показанную ранее, с применением `Hashtable` для хранения имен депозиторов и их текущих балансов:

```
// Демонстрация применения 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();

  // Добавить 1,000 на счет Джона Доу.
  bal = balance.get("Джон Доу");
  balance.put("Джон Доу", bal+1000);
  System.out.println("Новый баланс Джона Доу: " + balance.get("Джон Доу"));
}
}
```

Вывод этой программы показан ниже:

```
Тод Холл: 99.22
Ральф Смит: -19.08
Джон Доу: 3434.34
Джейн Бейкер: 1378.0
Том Смит: 123.22

Новый баланс Джона Доу: 4434.34
```

Одно существенное замечание: подобно классу `Map`, `Hashtable` не поддерживает напрямую итераторы. То есть предыдущая программа использует перечисление для отображения содержимого `balance`. Однако вы можете получить представления хеш-таблицы в виде наборов (`Set`), которые допускают использование итераторов. Чтобы сделать это, просто воспользуйтесь одним из методов представления коллекций, определенный в `Map`, таким как `entrySet()` или `keySet()`. Например, вы можете получить представление в виде набора всех ключей и пройти циклом по нему с применением либо итератора, либо усовершенствованного цикла `for`. Ниже представлена переработанная версия программы, которая демонстрирует эту технику.

```
// Применение итераторов с 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);

  // Отобразить все счета в хеш-таблице.
  // Для начала, получить ключи в виде набора.
  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();

    // Добавить 1 000 на счет Джона Доу.
    bal = balance.get("Джон Доу");
    balance.put("Джон Доу", bal+1000);
    System.out.println("Новый баланс Джона Доу: " + balance.get("Джон Доу"));
  }
  }
```

Properties

`Properties` (свойства) — подкласс `Hashtable`. Он служит для поддержки списков значений, в которых ключами являются объекты `String` и значения — также объекты `String`. Класс `Properties` используется многими другими классами Java. Так, например, это тип объекта, возвращаемого `System.getProperties()`, когда извлекаются переменные окружения. Хотя сам класс `Properties` не является обобщенным, некоторые из его методов используют обобщенный синтаксис.

`Properties` определяет следующую переменную-экземпляр:

```
Properties defaults;
```

Эта переменная содержит список свойств по умолчанию, ассоциированных с объектом `Properties`.

В `Properties` определены следующие конструкторы:

```
Properties()
Properties(Properties propDefault)
```

Первая версия создает объект `Properties`, не имеющий значений по умолчанию. Второй создает объект, используя *`propDefault`* в качестве значений по умолчанию. В обоих случаях список свойств пуст.

В дополнение к методам, унаследованным `Properties` от `Hashtable`, этот класс определяет методы, перечисленные в табл. 17.23. `Properties` также имеет один нежелательный метод: `save()`. Он был заменен `store()`, поскольку `save()` корректно не обрабатывал ошибки.

Одно удобное свойство класса `Properties` — это то, что вы можете указать значения по умолчанию, которые будут возвращены, если никакое значение не ассоциировано с определенным ключом. Например, значение по умолчанию может быть указано вместе с ключом в методе `getProperty()` — как, например, в `getProperty("имя", "значение_по_умолчанию")`. Если значение `"имя"` не найдено, возвращается `"значение_по_умолчанию"`.

При конструировании объекта `Properties` вы можете передать ему другой экземпляр `Properties` в качестве списка свойств по умолчанию для нового экземпляра. В этом случае, если вы вызываете `getProperty("foo")` для данного объекта `Properties`, и `"foo"` не существует, Java ищет его в объекте `Properties` по умолчанию. Это позволяет иметь произвольное количество уровней вложения свойств по умолчанию.

Таблица 17.23. Методы, определенные в Properties

Метод	Описание
String getProperty(String *key*)	Возвращает значение, ассоциированное с *key*. Если *key* нет ни в самом списке свойств, ни в списке свойств по умолчанию, то возвращается null.
String getProperty(String *key*, String *defaultProperty*)	Возвращает значение, ассоциированное с *key*. Если *key* нет ни в самом списке свойств, ни в списке свойств по умолчанию, то возвращается *defaultProperty*.
void list(PrintStream *streamOut*)	Посылает список свойств в выходной поток, связанный с *streamOut*.
void list(PrintWriter *streamOut*)	Посылает список свойств в выходной поток, связанный с *streamOut*.
void load(InputStream *streamIn*) throws IOException	Загружает список свойств из входного потока, связанного с *streamIn*.
void load(Reader *streamIn*) throws IOException	Загружает список свойств из входного потока, связанного с *streamIn*. (Добавлен в Java SE 6.)
void loadFromXML(InputStream *streamIn*) throws IOException, InvalidPropertiesFormatException	Загружает список свойств из XML-документа, связанного с *streamIn*.
Enumeration<?> propertyNames()	Возвращает перечисление ключей. Это включает также ключи, найденные в списке свойств по умолчанию.
Object setProperty(String *key*, String *value*)	Ассоциирует *value* с *key*. Возвращает предыдущее значение, ассоциированное с ключом *key*, либо null, если такой ассоциации не найдено.
void store(OutputStream *streamOut*, String *description*) throws IOException	После записи строки, указанной в *description*, список свойств записывается в поток, связанный с *streamOut*.
void store(Writer *streamOut*, String *description*) throws IOException	После записи строки, указанной в *description*, список свойств записывается в поток, связанный с *streamOut*. (Добавлен в Java SE 6.)
void storeToXML(OutputStream *streamOut*, String *description*) throws IOException	После записи строки, указанной в *description*, список свойств записывается в XML-документ, связанный с *streamOut*.
void storeToXML(OutputStream *streamOut*, String *description*, String *enc*)	Список свойств и строка, указанная в *description*, записываются в XML-документ, связанный с *streamOut* с применением указанной кодировки символов.
Set<String> stringPropertyNames()	Возвращает набор ключей. (Добавлен в Java SE 6.)

В следующем примере демонстрируется применение `Properties`. В нем создается список свойств, в котором ключами являются названия штатов, а значениями — названия из столиц. Обратите внимание, что попытка найти столицу Флориды включает значение по умолчанию.

```
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 набор ключей.
  Set states = capitals.keySet();

  // Показать все штаты и столицы.
  for(Object name : states)
     System.out.println("Столица штата " + name + " - " +
                        capitals.getProperty((String)name) + ".");
  System.out.println();

  // Поиск штата, не содержащегося в списке — с указанием умолчания.
  String str = capitals.getProperty("Флорида", "не найдена");
  System.out.println("Столица Флориды " + str + ".");
}
}
```

Вывод этой программы показан ниже:

```
Столица штата Миссури — Джефферсон-Сити.
Столица штата Иллинойс - Спрингфилд.
Столица штата Индиана - Индианаполис.
Столица штата Калифорния - Сакраменто.
Столица штата Вашингтон - Олимпия.

Столица Флориды не найдена.
```

Поскольку Флорида не содержится в списке, используется значение по умолчанию.

Хотя это исключительно правильно — использовать значения по умолчанию при вызове `getProperty()`, как показано в предыдущем примере, существует лучший способ управления значениями по умолчанию для большинства приложений, имеющих дело со списками свойств. Для большей гибкости задавайте список свойств по умолчанию при конструировании объекта `Properties`. Если нужный ключ в главном списке на найден, поиск производится в списке по умолчанию. Например, ниже представлена слегка измененная версия предыдущей программы с применением списка штатов по умолчанию. Теперь, когда ищется столица Флориды, она будет найдена в списке по умолчанию.

```
// Использование списка 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 набор ключей.
  Set states = capitals.keySet();

  // Показать все штаты и столицы.
  for(Object name : states)
     System.out.println("Столица штата " + name + " - " +
                        capitals.getProperty((String)name) + ".");
  System.out.println();

  // Теперь Флорида будет найдена в списке по умолчанию.
  String str = capitals.getProperty("Флорида");
  System.out.println("Столица Флориды - " + str + ".");
}
}
```

Использование `store()` и `load()`

Один из наиболее удобных аспектов класса `Properties` в том, что информация, содержащаяся в объекте `Properties`, может быть легко сохранен и загружен с диска методами `store()` и `load()`. В любой момент вы можете записать объект `Properties` в поток либо прочесть его обратно. Это делает списки свойств особенно удобными для реализации простых баз данных. Например, следующая программа использует список свойств для создания простого телефонного справочника, хранящего имена и номера телефонов. Чтобы найти номер лица, вы вводите его или ее имя. Программа использует методы `store()` и `load()` для сохранения и чтения списка. Когда эта программа выполняется, вначале она пытается загрузить список из файла по имени `phonebook.dat`. Если этот файл существует, он загружается. Затем вы можете добавлять новые значения в список. Если вы это делаете, список сохраняется при завершении программы. Обратите внимание, насколько компактный код требуется для реализации маленькой, но функциональной компьютеризованной телефонной книги.

```
/* Простая база данных телефонных номеров, использующая списки 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;

   // Попытка 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 вносить новые имена и номера телефонов.
    do {
       System.out.println("Введите имя" + " ('выход' для останова): ");
       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"));
    // Если телефонная книга изменилась, сохранить ее.
    if(changed) {
       FileOutputStream fout = new FileOutputStream("phonebook.dat");
       ht.store(fout, "Телефонная книга");
       fout.close();
    }
    // Искать номер по имени.
    do {
       System.out.println("Введите имя для поиска" +
                          " ('выход' для останова): ");
       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 предлагает вам, как программисту, мощный набор тщательно спроектированных решений для некоторых наиболее часто встречающихся программистских задач. Теперь, когда каркас коллекций стал обобщенным, он может применяться с поддержкой полной безопасности типов, что способствует его дальнейшему развитию. Рассмотрите возможность применения коллекций в следующий раз, когда вам понадобится сохранять и извлекать информацию. Помните, что коллекции не предназначены только для “крупных задач” вроде корпоративных баз данных, списков почтовых рассылок либо систем инвентаризации. Они также эффективны для решения небольших задач. Например, `TreeMap` — отличная коллекция для хранения структуры каталогов или наборов файлов. `TreeSet` может оказаться довольно удобным для хранения информации по управлению проектом. В общем случае, типы проблем, при решении которых средствами коллекций можно получить существенный выигрыш, ограничиваются только вашим воображением.

18 ГЛАВА

java.util: прочие служебные классы

В этой главе продолжается обсуждение пакета `java.util` и рассматриваются классы и интерфейсы, которые не являются частью системы коллекций. Это включает классы, разбивающие строки на лексемы, работающие с датами, генерирующие случайные числа, связывающие ресурсы и наблюдающие за событиями. Также описываются новые классы `Formatter` и `Scanner`, которые облегчают чтение и запись форматированных данных. И, наконец, вкратце упоминаются вложенные пакеты `java.util`.

StringTokenizer

Обработка текста часто предполагает разбор форматированной входной строки. *Разбор* (parsing) — это разделение текста на набор дискретных составных частей, или *лексем* (token), представляющих определенные последовательности, которые могут иметь некоторое семантическое значение. Класс `StringTokenizer`, представляющий первый шаг в процессе разбора, часто называют лексическим анализатором или сканером. `StringTokenizer` реализует интерфейс `Enumeration`. Таким образом, получая входную строку, вы можете перечислить содержащиеся в ней индивидуальные лексемы с помощью `StringTokenizer`.

Чтобы использовать `StringTokenizer`, вы указываете входную строку и строку, содержащую разделители. *Разделители* (delimiters) — это символы, разделяющие лексемы. Каждый символ в строке-разделителе рассматривается как допустимый разделитель — например, строка `",;:"` устанавливает в качестве разделителей запятую, точку с запятой и двоеточие. Набор разделителей по умолчанию состоит из пробельных символов: пробела, знака табуляции, перевода строки и возврата каретки.

Конструкторы `StringTokenizer` показаны ниже:

```
StringTokenizer(String str)
StringTokenizer(String str, String delimiters)
StringTokenizer(String str, String delimiters, boolean delimAsToken)
```

Во всех трех версиях *`str`* — это строка, которая будет разбита на части. В первой версии используются разделители по умолчанию. Во второй и третьей версиях *`delimiters`* —

это строка, специфицирующая разделители. В третьей версии, если *delimAsToken* равно true, то сами разделители возвращаются в качестве отдельных лексем при разборе строки. В противном случае разделители не возвращаются. Разделители также не возвращаются в первых двух формах.

Однажды создав объект StringTokenizer, можно использовать метод nextToken() для извлечения последовательных лексем. Метод hasMoreTokens() возвращает true до тех пор, пока существуют лексемы для извлечения. Поскольку StringTokenizer реализует Enumeration, методы hasMoreElements() и nextElement() также реализованы, и они работают точно так же, как, соответственно, hasMoreTokens() и nextToken(). Методы StringTokenizer перечислены в табл. 18.1.

Ниже приведен пример, создающий StringTokenizer для разбора пар "ключ=значение". Последовательность наборов "ключ=значение" разделена точками с запятой.

```
// Демонстрация применения StringTokenizer.
import java.util.StringTokenizer;
class STDemo {
 static String in = "title=Java: The Complete Reference;" +
   "author=Schildt;" +
   "publisher=Osborne/McGraw-Hill;" +
   "copyright=2007";
  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);
   }
  }
}
```

Вывод этой программы выглядит так:

```
title  Java: The Complete Reference
author Schildt
publisher  Osborne/McGraw-Hill
copyright  2007
```

Таблица 18.1. Методы, определенные в StringTokenizer

Метод	Описание
int countTokens()	Используя текущий набор разделителей, метод определяет количество лексем, которые осталось разобрать и вернуть в результате.
boolean hasMoreElements()	Возвращает true, если один или более лексем осталось в строке, в противном случае возвращает false.
boolean hasMoreTokens()	Возвращает true, если один или более лексем осталось в строке, в противном случае возвращает false.
Object nextElement()	Возвращает следующую лексему как Object.
String nextToken()	Возвращает следующую лексему как String.
String nextToken(String *delimiters*)	Возвращает следующую лексему как Object и устанавливает строку разделителей, как указано в *delimiters*.

BitSet

Класс BitSet создает специальный тип массива, содержащий битовые значения. Этот массив при необходимости может увеличиваться в размере. Это делает его похожим на битовый вектор. Конструкторы BitSet показаны ниже:

```
BitSet()
BitSet(int size)
```

Первая версия создает объект по умолчанию. Вторая версия позволяет указать начальный размер (то есть количество бит, которые можно сохранить). Все биты инициируются нулями.

BitSet определяет методы, представленные в табл. 18.2.

Таблица 18.2. Методы, определенные в BitSet

Метод	Описание
void and(BitSet *bitSet*)	Выполняет операцию логического "И" (AND) между вызывающим объектом BitSet и указанным в параметре *bitSet*. Результат помещается в вызывающий объект.
void andNot(BitSet *bitSet*)	Для каждого бита в bitSet, равного 1, очищается соответствующий бит в вызывающем BitSet.
int cardinality()	Возвращает количество установленных бит в вызывающем объекте.
void clear()	Устанавливает все биты в ноль.
void clear(int *index*)	Устанавливает в ноль бит в позиции *index*.
void clear(int *startIndex*, int *endIndex*)	Устанавливает в ноль биты от *startIndex* до *endIndex*-1.
Object clone()	Дублирует вызывающий объект.
boolean equals(Object *bitSet*)	Возвращает true, если вызывающий набор бит эквивалентен переданному в параметре *bitSet*. В противном случае возвращает false.
void flip(int *index*)	Обращает бит, находящийся в позиции *index*.
void flip(int *startIndex*, int *endIndex*)	Обращает биты в диапазоне от *startIndex* до *endIndex*-1.
boolean get(int *index*)	Возвращает текущее значение бита в позиции *index*.
BitSet get(int *startIndex*, int *endIndex*)	Возвращает BitSet, состоящий из бит от *startIndex* до *endIndex*-1.
int hashCode()	Возвращает хеш-код вызывающего объекта.
boolean intersects(BitSet *bitSet*)	Возвращает true, если хотя бы одна пара соответствующих бит в вызывающем объекте и *bitSet* равна 1.
boolean isEmpty()	Возвращает true, если все биты вызывающего объекта установлены в 0.
int length()	Возвращает количество бит, необходимых для того, чтобы уместить содержимое вызывающего BitSet. Это значение определяется по положению последнего бита, равного 1.

Окончание табл. 18.2

Метод	Описание
`int nextClearBit(int startIndex)`	Возвращает позицию следующего очищенного бита (то есть следующего бита, равного 0), начиная с индекса, указанного в `startIndex`.
`int nextSetBit(int startIndex)`	Возвращает позицию следующего установленного бита (то есть следующего бита, равного 1), начиная с индекса, указанного в `startIndex`. Если ни один бит не установлен, возвращается `-1`.
`void or(BitSet bitSet)`	Выполняет операцию логического "ИЛИ" (OR) между вызывающим объектом `BitSet` и указанным в параметре `bitSet`. Результат помещается в вызывающий объект.
`void set(int index)`	Устанавливает бит в позиции `index` равным 1.
`void set(int index, boolean v)`	Устанавливает бит в позиции `index` равным значению, переданному в `v`. `true` устанавливает бит в 1, `false` — в 0.
`void set(int startIndex, int endIndex)`	Устанавливает биты в позициях от `startIndex` до `endIndex-1` равными 1.
`void set(int startIndex, int endIndex, boolean v)`	Устанавливает биты в позициях от `startIndex` до `endIndex-1` равными значению, переданному в `v`. `true` устанавливает бит в 1, `false` — в 0.
`int size()`	Возвращает количество бит в вызывающем объекте `BitSet`.
`String toString()`	Возвращает строковый эквивалент вызывающего объекта `BitSet`.
`void xor(BitSet bitSet)`	Выполняет операцию исключающего логического "ИЛИ" (XOR) над содержимым вызывающего объекта `BitSet` и переданного в `bitSet`. Результат помещается в вызывающий объект.

Ниже приведен пример, демонстрирующий использование `BitSet`.

```
// Демонстрация применения 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("Начальный шаблон в bits1: ");
  System.out.println(bits1);
  System.out.println("\nНачальный шаблон в bits2: ");
  System.out.println(bits2);

  // Логическое И над битами
  bits2.and(bits1);
  System.out.println("\nbits2 AND bits1: ");
  System.out.println(bits2);
```

```
    // Логическое ИЛИ над битами
    bits2.or(bits1);
    System.out.println("\nbits2 OR bits1: ");
    System.out.println(bits2);

    // Логическое исключающее ИЛИ над битами
    bits2.xor(bits1);
    System.out.println("\nbits2 XOR bits1: ");
    System.out.println(bits2);
  }
  }
```

Вывод этой программы показан ниже. Когда `toString()` преобразует объект `BitSet` в его строковый эквивалент, каждый установленный бит представляется его позицией. Очищенные биты не показаны.

```
Начальный шаблон в bits1:
{0, 2, 4, 6, 8, 10, 12, 14}

Начальный шаблон в 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:
{}
```

Date

Класс `Date` инкапсулирует текущую дату и время. Прежде чем мы начнем изучать `Date`, важно отметить, что этот класс ощутимо изменился по сравнению с его оригинальной версией, определенной в Java 1.0. Когда вышла версия Java 1.1, многие из функций исходного класса `Date` были перемещены в классы `Calendar` и `DateFormat` и, как результат, многие из исходных методов `Date` объявлены нежелательными (deprecated). Поскольку эти методы из Java 1.0 не должны использоваться в новом коде, здесь они не описываются.

`Date` поддерживает следующие конструкторы:

```
Date()
Date(long millisec)
```

Первый конструктор инициализирует объект текущей датой и временем. Второй конструктор принимает один аргумент, который представляет собой количество миллисекунд, прошедших с полуночи 1 января 1970 г. Методы `Date`, не относящиеся к нерекомендованным, перечислены в табл. 18.3. `Date` также реализует интерфейс `Comparable`.

Как вы можете видеть в табл. 18.3, средства класса `Date` не позволяют получать индивидуальные компоненты даты и времени. Как демонстрируется в следующей программе, вы можете только получить дату и время в терминах миллисекунд либо в строковом представлении по умолчанию, возвращаемом `toString()`.

Таблица 18.3. Методы, определенные в `Date`

Метод	Описание
`boolean after(Date date)`	Возвращает `true`, если вызывающий объект `Date` содержит более позднюю дату, чем указанная в *`date`*. В противном случае возвращает `false`.
`boolean before(Date date)`	Возвращает `true`, если вызывающий объект `Date` содержит более раннюю дату, чем указанная в *`date`*. В противном случае возвращает `false`.
`Object clone()`	Дублирует вызывающий объект `Date`.
`int compareTo(Date date)`	Сравнивает значения вызывающего объекта и *`date`*. Возвращает 0, если эти значения эквивалентны. Возвращает отрицательное значение, если вызывающий объект содержит более раннюю дату, чем *`date`*. Возвращает положительное значение, если вызывающий объект содержит более позднюю дату.
`boolean equals(Object date)`	Возвращает `true`, если вызывающий объект `Date` содержит то же самое время и дату, что и указанные в *`date`*. В противном случае возвращает `false`.
`long getTime()`	Возвращает количество миллисекунд, прошедших с полуночи 1 января 1970 г.
`int hashCode()`	Возвращает хеш-код вызывающего объекта.
`void setTime(long time)`	Устанавливает время и дату в значение, переданное в *`time`*, которое представляет число миллисекунд, прошедших с полночи 1 января 1970 г.
`String toString()`	Преобразует вызывающий объект `Date` в строку и возвращает результат.

Чтобы получить более детальную информацию о времени и дате, используйте класс `Calendar`.

```
// Показывает время и дату, используя только методы класса Date.
import java.util.Date;
class DateDemo {
public static void main(String args[]) {

  // Создать объект Date
  Date date = new Date();

  // Отобразить дату и время с помошью toString()
  System.out.println(date);

  // Отобразить количество миллисекунд, прошедших с 1 января 1970 г. по GMT
  long msec = date.getTime();
  System.out.println("Миллисекунд, прошедших с 1 января 1970 г. по GMT = " + msec);
}
}
```

Пример вывода этой программы:

```
Mon Jan 01 16:28:16 CST 2007
Миллисекунд, прошедших с 1 января 1970 г. по GMT = 1167690496023
```

Calendar

Абстрактный класс Calendar представляет набор методов, позволяющих преобразовывать время в миллисекундах во множество удобных компонентов. Некоторые примеры типов информации, которые могут быть представлены — это год, месяц, день, часы, минуты и секунды. Его предназначение в том, чтобы подклассы Calendar представляли специфическую функциональность по интерпретации информации о времени в соответствии со своими собственными правилами. Это один аспект библиотеки классов Java, который позволяет писать программы, работающие в различных интернациональных окружениях. Примером такого подкласса может служить GregorianCalendar.

Calendar не имеет общедоступных конструкторов.

Calendar определяет несколько защищенных переменных экземпляра. areFieldsSet — переменная типа boolean, которая указывает, были ли установлены компоненты времени. fields — массив целочисленных значений, содержащих компоненты времени. isSet — массив переменных типа boolean, указывающих, что специфический компонент времени был установлен. time — переменная типа long, содержащая текущее время для объекта. isTimeSet — переменная boolean, указывающая, что текущее время было установлено.

Некоторые часто используемые методы Calendar показаны в табл. 18.4.

Таблица 18.4. Часто используемые методы Calendar

Метод	Описание
`abstract void add(int which, int val)`	Добавляет *val* к компоненту времени или даты, указанному в *which*. Чтобы отнять, добавляйте отрицательное значение. Параметр *which* может быть одним из полей, определенных в Calendar, таких как Calendar.HOUR.
`boolean after(Object calendarObj)`	Возвращает true, если вызывающий объект Calendar содержит более позднюю дату, чем *calendarObj*. В противном случае возвращает false.
`boolean before(Object calendarObj)`	Возвращает true, если вызывающий объект Calendar содержит более раннюю дату, чем *calendarObj*. В противном случае возвращает false.
`final void clear()`	Обнуляет все компоненты времени в вызывающем объекте.
`final void clear(int which)`	Обнуляет компонент времени вызывающего объекта, указанный в *which*.
`Object clone()`	Возвращает дубликат вызывающего объекта.
`boolean equals(Object calendarObj)`	Возвращает true, если вызывающий объект Calendar содержит дату, эквивалентную *calendarObj*. В противном случае возвращает false.
`int get(int calendarField)`	Возвращает значение одного компонента вызывающего объекта. Компонент указывается в *calendarField*. Примеры компонентов, которые можно запросить: Calendar.YEAR, Calendar.MONTH, Calendar.MINUTE и т.п.

Окончание табл. 18.4

Метод	Описание
`static Locale[] getAvailableLocales()`	Возвращает массив объектов `Locale`, который содержит локали, для которых в системе доступны календари/
`static Calendar getInstance()`	Возвращает объект `Calendar` для локали и часового пояса по умолчанию.
`static Calendar getInstance(TimeZone tz)`	Возвращает объект `Calendar` для локали по умолчанию и часового пояса *tz*.
`static Calendar getInstance(Locale locale)`	Возвращает объект `Calendar` для локали *locale* и часового пояса по умолчанию.
`static Calendar getInstance(TimeZone tz, Locale locale)`	Возвращает объект `Calendar` для локали *locale* и часового пояса *tz*.
`final Date getTime()`	Возвращает объект `Date`, содержащий время, эквивалентное вызывающему объекту.
`TimeZone getTimeZone()`	Возвращает часовой пояс вызывающего объекта.
`final boolean isSet(int which)`	Возвращает `true`, если указанный компонент времени установлен. В противном случае возвращает `false`.
`void set(int which, int val)`	Устанавливает компонент даты или времени вызывающего объекта, указанный в *which*, равным значению *val*. *which* должен иметь значение одного из полей `Calendar`, такое как `Calendar.HOUR`.
`final void set(int year, int month, int dayOfMonth)`	Устанавливает в вызывающем объекте различные компоненты даты и времени.
`final void set(int year, int month, int dayOfMonth, int hours, int minutes)`	Устанавливает в вызывающем объекте различные компоненты даты и времени.
`final void set(int year, int month, int dayOfMonth, int hours, int minutes, int seconds)`	Устанавливает в вызывающем объекте различные компоненты даты и времени.
`final void setTime(Date d)`	Устанавливает в вызывающем объекте различные компоненты даты и времени. Информация передается в объекте *d* типа `Date`.
`void setTimeZone(TimeZone tz)`	Устанавливает часовой пояс для вызывающего объекта равным *tz*.

В `Calendar` определены следующие целые константы, которые используются, когда вы получаете или устанавливаете компоненты календаря:

```
ALL_STYLES                FRIDAY                PM
AM                        HOUR                  SATURDAY
AM_PM                     HOUR_OF_DAY           SECOND
APRIL                     JANUARY               SEPTEMBER
AUGUST                    JULY                  SHORT
DATE                      JUNE                  SUNDAY
DAY_OF_MONTH              LONG                  THURSDAY
```

```
DAY_OF_WEEK                 MARCH                   TUESDAY
DAY_OF_WEEK_IN_MONTH        MAY                     UNDECIMBER
DAY_OF_YEAR                 MILLISECOND             WEDNESDAY
DECEMBER                    MINUTE                  WEEK_OF_MONTH
DST_OFFSET                  MONDAY                  WEEK_OF_YEAR
ERA                         MONTH                   YEAR
FEBRUARY                    NOVEMBER                ZONE_OFFSET
FIELD_COUNT                 OCTOBER
```

В следующей программе демонстрируется использование нескольких методов Calendar.

```
// Демонстрация применения 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, инициализированный
  // текущей датой и временем с локалью
  // и часовым поясом по умолчанию.
  Calendar calendar = Calendar.getInstance();

  // Отобразить текущее время и дату.
  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 информацию даты и времени и отобразить ее.
  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));
}
}
```

Пример вывода этой программы:

```
Дата: Jan 1 2007
Время: 11:24:25
Измененное время: 10:29:22
```

GregorianCalendar

GregorianCalendar — конкретная реализация Calendar, которая представляет нормальный Григорианский календарь, с которым вы хорошо знакомы. Метод getInstance() класса Calendar обычно возвращает GregorianCalendar, инициированный текущей датой и временем в локали и часовом поясе по умолчанию.

GregorianCalendar определяет два поля: AD и BC. Они представляют две эры, определенные в Григорианском календаре.

Кроме того, имеется также несколько конструкторов для объектов GregorianCalendar. Конструктор по умолчанию, GregorianCalendar(), инициирует объект текущим временем и датой в локали и часовом поясе по умолчанию. Есть и другие конструкторы, представленные ниже по мере возрастания специализации:

```
GregorianCalendar(int year, int month, int dayOfMonth)
GregorianCalendar(int year, int month, int dayOfMonth, int hours, int minutes)
GregorianCalendar(int year, int month, int dayOfMonth, int hours,
                  int minutes, int seconds)
```

Все три версии устанавливают день, месяц и год. Здесь *year* указывает год. Месяц задается в *month*, причем ноль означает январь. День месяца — это *dayOfMonth*. Первая версия устанавливает время в полночь. Вторая версия устанавливает также часы и минуты. Третья версия добавляет секунды.

Вы также можете сконструировать объект GregorianCalendar, указав локаль и/или часовой пояс. Следующие конструкторы создают объекты, инициированные текущим временем и датой, используя заданный часовой пояс и/или локаль:

```
GregorianCalendar(Locale locale)
GregorianCalendar(TimeZone timeZone)
GregorianCalendar(TimeZone timeZone, Locale locale)
```

GregorianCalendar представляет реализацию абстрактных методов Calendar. Кроме того, в нем определены некоторые дополнительные методы. Возможно, наиболее интересный из них — isLeapYear(), который проверяет високосный год. Его форма такова:

```
boolean isLeapYear(int year)
```

Метод возвращает true, если *year* — високосный год, и false — в противоположном случае.

В следующей программе демонстрируется использование GregorianCalendar.

```
// Демонстрация применения GregorianCalendar
import java.util.*;

class GregorianCalendarDemo {
public static void main(String args[]) {
  String months[] = {
      "Янв", "Фев", "Мар", "Апр",
      "Май", "Июн", "Июл", "Авг",
      "Сен", "Окт", "Ноя", "Дек"};
  int year;

  // Создать Григорианский календарь, инициализированный
  // текущей датой и временем с локалью
  // и часовым поясом по умолчанию
  GregorianCalendar gcalendar = new GregorianCalendar();
```

```
    // Отобразить текущее 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 — високосный ли текущий год
    if(gcalendar.isLeapYear(year)) {
      System.out.println("Текущий год високосный");
    }
    else {
      System.out.println("Текущий год не високосный");
    }
  }
  }
```

Пример вывода этой программы:

```
Дата: Jan 1 2007
Время: 11:24:27
Текущий год не високосный
```

TimeZone

Еще один класс, имеющий отношение ко времени — это `TimeZone`. Класс `TimeZone` позволяет работать с часовыми поясами, смещенными относительно Гринвича (GMT), также известного как универсальное глобальное время (Coordinated Universal Time — UCT). Он также учитывает летнее время. `TimeZone` поддерживает только конструктор по умолчанию. Примеры методов, определенных в `TimeZone`, перечислены в табл. 18.5.

Таблица 18.5. Некоторые из методов `TimeZone`

Метод	Описание
`Object clone()`	Возвращает специфичную для `TimeZone` версию `clone()`.
`static String[] getAvailableIDs()`	Возвращает массив объектов `String`, представляющих имена всех часовых поясов.
`static String[] getAvailableIDs(int timeDelta)`	Возвращает массив объектов `String`, представляющих имена всех часовых поясов, отстоящих на смещение *timeDelta* относительно GMT.
`static TimeZone getDefault()`	Возвращает объект `TimeZone`, который представляет часовой пояс по умолчанию, принятый на данном компьютере.
`String getID()`	Возвращает имя вызывающего объекта `TimeZone`.
`abstract int getOffset( int era, int year, int month, int dayOfMonth, int dayOfWeek, int millisec)`	Возвращает смещение, которое должно быть добавлено к GMT, чтобы вычислить локальное время. Это значение корректируется с учетом летнего времени. Параметры метода представляют компоненты даты и времени.

Окончание табл. 18.5

Метод	Описание
`abstract int getRawOffset()`	Возвращает смещение, которое должно быть добавлено к GMT, чтобы вычислить локальное время. Значение не корректируется с учетом летнего времени.
`static TimeZone getTimeZone(String tzName)`	Возвращает объект `TimeZone` по часовому поясу, имеющему название *tzName*.
`abstract boolean inDaylightTime(Date d)`	Возвращает `true`, если дата, представленная в *d*, относится к летнему времени в вызывающем объекте. В противном случае возвращает `false`.
`static void setDefault(TimeZone tz)`	Устанавливает часовой пояс по умолчанию для данного хоста. *tz* — ссылка на объект `TimeZone`, который нужно использовать.
`void setID(String tzName)`	Устанавливает имя часового пояса (то есть его идентификатор) равным *tzName*.
`abstract void setRawOffset(int millis)`	Устанавливает смещение относительно GMT в миллисекундах.
`abstract boolean useDaylightTime()`	Возвращает `true`, если вызывающий объект использует летнее время. В противном случае возвращает `false`.

SimpleTimeZone

Класс `SimpleTimeZone` — это удобный подкласс `TimeZone`. Он реализует абстрактные методы `TimeZone` и позволяет работать с часовыми поясами в Григорианском календаре. Он также учитывает летнее время.

`SimpleTimeZone` определяет четыре конструктора. Первый из них:

```
SimpleTimeZone(int timeDelta, String tzName)
```

Этот конструктор создает объект `SimpleTimeZone`. Здесь *timeDelta* — это смещение относительно Гринвича, а *tzName* — название часового пояса.

Второй конструктор:

```
SimpleTimeZone(int timeDelta, String tzId, int dstMonth0,
               int dstDayInMonth0, int dstDay0, int time0,
               int dstMonth1, int dstDayInMonth1, int dstDay1,
               int time1)
```

Здесь смещение относительно GMT задается в *timeDelta*. Имя часового пояса передается в *tzId*. Начало действия летнего времени определяется параметром *dstMonth0*, *dstDayInMonth0*, *dstDay0* и *time0*. Окончание действия летнего времени задается параметрами *dstMonth1*, *dstDayInMonth1*, *dstDay1* и *time1*.

Третий конструктор `SimpleTimeZone`:

```
SimpleTimeZone(int timeDelta, String tzId, int dstMonth0,
               int dstDayInMonth0, int dstDay0, int time0,
               int dstMonth1, int dstDayInMonth1, int dstDay1,
               int time1, int dstDelta)
```

Здесь *dstDelta* — количество миллисекунд, сохраненных переходом на летнее время.

Четвертый конструктор SimpleTimeZone:

```
SimpleTimeZone(int timeDelta, String tzId, int dstMonth0,
               int dstDayInMonth0, int dstDay0, int time0,
               int time0mode, int dstMonth1, int dstDayInMonth1,
               int dstDay1, int time1, int time1mode, int dstDelta)
```

Здесь *time0mode* указывает режим начального времени, а *time1mode* — режим конечного времени. Ниже перечислены допустимые значения этих режимов.

- STANDARD_TIME
- WALL_TIME
- UTC_TIME

Режим времени определяет, как интерпретируются значения времени. Значение режима по умолчанию, используемое другими конструкторами — WALL_TIME.

Locale

Класс Locale предназначен для создания объектов, каждый из которых описывает географический или культурный регион. Это один из нескольких классов, которые обеспечивают возможность строить программы, предназначенные для выполнения в различных национальных окружениях. Например, форматы, применяемые для отображения дат, времен и чисел, в разных регионах отличаются.

Интернационализация — это крупная тема, выходящая за пределы контекста настоящей книги. Однако большинство программ нуждаются только в том, чтобы иметь дело с ее основами, что включает установки только текущей локали.

Класс Locale определяет следующие константы, которые удобны для обращения с наиболее часто используемыми локалями:

CANADA	GERMAN	KOREAN
CANADA_FRENCH	GERMANY	PRC
CHINA	ITALIAN	SIMPLIFIED_CHINESE
CHINESE	ITALY	TAIWAN
ENGLISH	JAPAN	TRADITIONAL_CHINESE
FRANCE	JAPANESE	UK
FRENCH	KOREA	US

Например, выражение Locale.CANADA представляет объект Locale для Канады.

Вот как выглядят конструкторы Locale:

```
Locale(String language)
Locale(String language, String country)
Locale(String language, String country, String data)
```

Эти конструкторы строят объект Locale для представления специфического *языка*, а в случае последних двух конструкторов — также и *страны*. Эти значения должны содержать коды стран и языков в стандарте ISO. Вспомогательная информация, специфичная для конкретного браузера или поставщика, может быть передана в *data*.

Locale определяет несколько методов. Один из наиболее важных — setDefault() — показан ниже.

```
static void setDefault(Locale localeObj)
```

Он устанавливает локаль по умолчанию в *localeObj*.

Вот еще несколько других интересных методов:

```
final String getDisplayCountry()
final String getDisplayLanguage()
final String getDisplayName()
```

Они возвращают читабельные для человека строки, которые могут быть использованы для отображения наименования страны, наименования языка и полного описания локали.

Локаль по умолчанию можно получить методом `getDefault()`, показанным ниже:

```
static Locale getDefault()
```

`Calendar` и `GregorianCalendar` — это примеры классов, которые работают в чувствительной к локали манере. `DateFormat` и `SimpleDateFormat` также зависят от локали.

Random

Класс `Random` представляет собой генератор псевдослучайных чисел. Они называются псевдослучайными, поскольку представляют собой просто сложные распределенные последовательности. `Random` определяет следующие конструкторы:

```
Random()
Random(long seed)
```

Первая версия создает генератор чисел, использующий текущее время в качестве начального значения. Вторая форма позволяет вам указать это значение вручную.

Если вы инициируете объект `Random` начальным числом, то этим определяете начальную точку случайной последовательности. Если вы используете одно и то же начальное число для инициализации разных объектов `Random`, то получите от каждого из них одинаковые случайные последовательности. Если вы хотите получить разные последовательности, инициируйте объекты разными числами. Простейший способ сделать это — использовать текущее время в качестве инициирующего значения для объекта `Random`. Такой подход уменьшит вероятность получения повторяющихся последовательностей.

Общедоступные методы `Random` перечислены в табл. 18.6.

Таблица 18.6. Методы, определенные в `Random`

Метод	Описание
`boolean nextBoolean()`	Возвращает следующее случайное значение типа `boolean`.
`void nextBytes(byte vals[])`	Заполняет *vals* случайно сгенерированными значениями.
`double nextDouble()`	Возвращает следующее случайное значение типа `double`.
`float nextFloat()`	Возвращает следующее случайное значение типа `float`.
`double nextGaussian()`	Возвращает следующее значение гауссова случайного числа.
`int nextInt()`	Возвращает следующее случайное значение типа `int`.
`int nextInt(int n)`	Возвращает следующее случайное значение типа `int` в диапазоне от 0 до *n*.
`long nextLong()`	Возвращает следующее случайное значение типа `long`.
`void setSeed(long newSeed)`	Устанавливает начальное значение (то есть начальную точку для генератора случайных чисел) равным *newSeed*.

Как видите, есть семь типов случайных чисел, которые вы можете извлечь из объекта Random. Булевские случайные значения доступны через nextBoolean(). Случайные байты можно получить с помощью nextBytes(). Целые случайные числа можно получать методом nextInt(). Равномерно распределенные по диапазону допустимых случайные длинные целые выдает nextLong(). Методы nextFloat() и nextDouble() возвращают случайные значения float и double, равномерно распределенные между 0,0 и 1,0. И, наконец, nextGaussian() возвращает значение double, центрированное по 0,0 со стандартным отклонением в 1,0. Это то, что известно под названием *кривая нормального распределения*.

Ниже приведен пример, демонстрирующий последовательность, генерируемую nextGaussian(). Он получает 100 случайных гауссовых значений и усредняет их. Программа также считает количество значений, попадающих в два стандартных отклонения — плюс или минус, используя инкремент 0,5 для каждой категории. Результат отображается графически на экране.

```
// Демонстрирует случайные гауссовы значения.
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));
  // отобразить кривую распределения
  for(int i=0; i<10; i++) {
     for(int x=bell[i]; x>0; x--)
       System.out.print("*");
     System.out.println();
  }
}
}
```

Ниже можно видеть пример запуска этой программы. Как видите, получается колоколоподобное распределение чисел.

```
Среднее всех значений: 0.0702235271133344
**
*******
******
***************
******************
*****************
*************
**********
********
***
```

Observable

Класс Observable служит для создания подклассов, за которыми могут наблюдать остальные части вашей программы. Когда с объектом такого подкласса происходят изменения, наблюдающие классы извещаются об этом. Наблюдающие классы должны реализовывать интерфейс Observer, в котором определен метод update(). Метод update() вызывается, когда наблюдатель получает извещение об изменении наблюдаемого объекта.

Observable определяет методы, показанные в табл. 18.7. Объект, который подлежит наблюдению, должен следовать двум простым правилам. Во-первых, если он изменяется, то должен вызывать setChanged(). Во-вторых, когда он готов известить наблюдателей об изменении, то должен вызвать notifyObservers(). Это заставляет наблюдающий объект (или объекты) вызывать метод update(). Будьте осторожны: если объект обращается к notifyObservers(), не вызвав предварительно setChanged(), то никакого действия не последует. Наблюдаемый объект должен вызывать и setChanged(), и notifyObservers(), прежде чем будет вызван update().

Отметим, что notifyObservers() имеет две формы: одна с аргументом, другая без. Если вызвать notifyObservers() с аргументом, этот объект передается методу update() наблюдателя в качестве второго параметра. В противном случае update() передается null. Вы можете использовать второй параметр для передачи объекта любого типа, подходящего вашему приложению.

Таблица 18.7. Методы, определенные в Observable

Метод	Описание
void addObserver(Observer *obj*)	Добавляет *obj* к списку объектов, наблюдающих за вызывающим объектом.
protected void clearChanged()	Вызов этого метода помечает вызывающий объект как “не изменявшийся”.
int countObservers()	Возвращает количество объектов, наблюдающих текущий объект.
void deleteObserver(Observer *obj*)	Удаляет *obj* из списка объектов, наблюдающих вызывающий объект.
void deleteObservers()	Удаляет все наблюдатели вызывающего объекта.
boolean hasChanged()	Возвращает true, если вызывающий объект модифицировался, и false, если нет.
void notifyObservers()	Извещает наблюдателей о том, что вызывающий объект изменялся вызовом update(). Вторым параметром update() передается null.
void notifyObservers(Object *obj*)	Извещает наблюдателей о том, что вызывающий объект изменялся вызовом update(). Вторым параметром update() передается *obj*.
protected void setChanged()	Вызывается, когда вызывающий объект изменялся.

Интерфейс Observer

Чтобы организовать наблюдение за наблюдаемым объектом, вы должны реализовать интерфейс Observer. Этот интерфейс определяет только один метод, показанный ниже:

```
void update(Observable observOb, Object arg)
```

Здесь *observOb* — это объект, подлежащий наблюдению, а *arg* — значение, переданное notifyObservers(). Метод update() вызывается, когда происходит изменение наблюдаемого объекта.

Пример использования Observer

Ниже приведен пример, демонстрирующий работу с наблюдаемым объектом. Он создает класс наблюдателя по имени Watcher, который реализует интерфейс Observer. Класс, подлежащий мониторингу, называется BeingWatched. Он расширяет Observable. Внутри BeingWatched имеется метод counter(), который просто выполняет обратный отсчет от указанного значения. Он использует sleep() для ожидания 10 секунд между отсчетами. Каждый раз, когда счетчик изменяется, вызывается notifyObservers(), которому передается в качестве аргумента текущее значение счетчика. Это заставляет вызываться метод update() внутри Watcher, который отображает текущее значение счетчика. Внутри main() создаются наблюдающий и наблюдаемый объекты Watcher и BeingWatched под именами, соответственно, observing и observed. Затем observing добавляется к списку наблюдателей для observed. Это означает, что observing.update() будет вызываться каждый раз, когда counter() вызывает notifyObservers().

```
/* Демонстрация применения класса Observable
   и интерфейса Observer.
*/
import java.util.*;

// Класс-наблюдатель.
class Watcher implements Observer {
public void update(Observable obj, Object arg) {
  System.out.println("update() вызван, count равен " +
            ((Integer)arg).intValue());
}
}

// 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();
   /* Добавить наблюдателя в список наблюдателей наблюдаемого объекта */
   observed.addObserver(observing);
   observed.counter(10);
}
}
```

Вывод этой программы показан ниже:

```
update() вызван, count равен 10
update() вызван, count равен 9
update() вызван, count равен 8
update() вызван, count равен 7
update() вызван, count равен 6
update() вызван, count равен 5
update() вызван, count равен 4
update() вызван, count равен 3
update() вызван, count равен 2
update() вызван, count равен 1
update() вызван, count равен 0
```

Наблюдателями могут быть более одного объекта. Например, в следующей программе реализуются два класса наблюдателя, и объекты каждого класса добавляются к списку наблюдателей `BeingWatched`. Второй наблюдатель ожидает, пока счетчик достигнет нулевого значения, после чего подает звуковой сигнал.

```
/* Объект может наблюдаться одним
   или более наблюдателями.
*/
import java.util.*;

// Первый класс-наблюдатель.
class Watcher1 implements Observer {
public void update(Observable obj, Object arg) {
  System.out.println("update() вызван, count равен " +
            ((Integer)arg).intValue());
}
}

// Второй класс-наблюдатель.
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" + '\7');
}
}

// 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();

    // Добавить оба наблюдателя
    observed.addObserver(observing1);
    observed.addObserver(observing2);
    observed.counter(10);
  }
  }
```

Класс `Observable` и интерфейс `Observer` позволяют реализовывать изощренные программные архитектуры, основанные на методологии "документ-представление". Они также удобны в многопоточных программах.

Timer и TimerTask

Интересным и полезным средством, предоставляемым пакетом `java.util` является возможность планировать запуск задания на определенное время в будущем. Классы, которые поддерживают это — `Timer` и `TimerTask`. Используя эти классы, вы можете создать поток, выполняющийся в фоновом режиме и ожидающий заданное время. Когда время истечет, задача, связанная с этим потоком, будет запущена. Различные опции позволяют запланировать задачу на повторяющийся запуск либо на запуск по определенной дате. Хотя всегда существует возможность вручную создать задачу, которая будет запущена в определенное время с помощью класса `Thread`, все же классы `Timer` и `TimerTask` значительно упрощают этот процесс.

`Timer` и `TimerTask` работают вместе. `Timer` — это класс, используемый для планирования выполнения задачи. Запланированная к выполнению задача должна быть экземпляром `TimerTask`. То есть, чтобы запланировать задачу, вы сначала создаете объект `TimerTask`, а затем планируете его запуск с помощью экземпляра `Timer`.

`TimerTask` реализует интерфейс `Runnable`. Это значит, что он может быть использован для создания потока выполнения. Его конструктор показан ниже:

```
TimerTask()
```

В `TimerTask` определены методы, перечисленные в табл. 18.8. Обратите внимание, что метод `run()` является абстрактным, а это означает, что он должен быть переопределен. Метод `run()`, определенный в интерфейсе `Runnable`, содержит исполняемый код. То есть простейший способ создать задачу для таймера — это расширить `TimerTask` и переопределить `run()`.

Как только задача создана, она планируется для выполнения объектом типа `Timer`. Вот как выглядят конструкторы класса `Timer`:

```
Timer()
Timer(boolean DThread)
Timer(String tName)
Timer(String tName, boolean DThread)
```

Первая версия создает объект `Timer` и затем запускает его как обычный поток. Вторая использует поток-демон, если параметр *`Dthread`* равен `true`. Поток-демон будет выполняться только до тех пор, пока выполняется остальная часть программы. Третий и четвертый конструкторы позволяют указывать имя объекта `Timer`. Методы класса `Timer` перечислены в табл. 18.9.

Таблица 18.8. Методы, определенные в классе `TimerTask`

Метод	Описание
`boolean cancel()`	Прерывает задание. Возвращает `true`, если выполнение задания прервано. В противном случае возвращает `false`.
`abstract void run()`	Содержит код задания таймера.
`long scheduledExecutionTime()`	Возвращает время, на которое последний раз планировался запуск задания.

Таблица 18.9. Методы, определенные в классе `Timer`

Метод	Описание
`void cancel()`	Прерывает поток таймера.
`int purge()`	Удаляет прерванные задания из очереди таймера.
`void schedule(TimerTask TTask, long wait)`	*`TTask`* планируется к выполнению через период в миллисекундах, переданный в параметре *`wait`*.
`void schedule(TimerTask TTask, long wait, long repeat)`	*`TTask`* планируется к выполнению через период в миллисекундах, переданный в параметре *`wait`*. Задание затем выполняется повторно периодически — каждые *`repeat`* миллисекунд.
`void schedule(TimerTask TTask, Date targetTime)`	*`TTask`* планируется к выполнению на время, указанное в *`targetTime`*.
`void schedule(TimerTask TTask, Date targetTime, long repeat)`	*`TTask`* планируется к выполнению на время, указанное в *`targetTime`*. Задание затем выполняется повторно периодически — каждые *`repeat`* миллисекунд.
`void scheduleAtFixedRate(TimerTask TTask, long wait, long repeat)`	*`TTask`* планируется к выполнению через период в миллисекундах, переданный в параметре *`wait`*. Задание затем выполняется повторно периодически — каждые *`repeat`* миллисекунд. Время каждого повтора задается относительно первого запуска, а не предыдущего. То есть общее время выполнения остается фиксированным.
`void scheduleAtFixedRate(TimerTask TTask, Date targetTime, long repeat)`	*`TTask`* планируется к выполнению на время, указанное в *`targetTime`*. Задание затем выполняется повторно периодически — каждые *`repeat`* миллисекунд. Время каждого повтора задается относительно первого запуска, а не предыдущего. То есть общее время выполнения остается фиксированным.

Как только объект Timer создан, запуск планируется вызовом его метода schedule(). Как показано в табл. 18.9, существует несколько форм метода schedule(), позволяющих запланировать задание разными способами.

Если вы создаете задание, не являющееся демоном, то, возможно, захотите вызвать cancel() для его прерывания при завершении программы. Если вы не сделаете этого, то ваша программа может "зависнуть" на некоторое время.

В следующей программе демонстрируется работа с Timer и TimerTask. В ней определяется задание таймера, чей метод run() отображает сообщение "Задание таймера выполняется". Это задание планируется на запуск каждые полсекунды после начальной паузы в одну секунду.

```
// Демонстрация применения Timer и TimerTask.
import java.util.*;
class MyTimerTask extends TimerTask {
public void run() {
  System.out.println("Задание таймера выполняется.");
}
}

class TTest {
public static void main(String args[]) {
  MyTimerTask myTask = new MyTimerTask();
  Timer myTimer = new Timer();
  /* Устанавливает начальную паузу в 1 секунду,
     затем повторяется каждые полсекунды.
  */
  myTimer.schedule(myTask, 1000, 500);
  try {
     Thread.sleep(5000);
  } catch (InterruptedException exc) {}
  myTimer.cancel();
}
}
```

Currency

Класс Currency инкапсулирует информацию о валюте. Он не определяет конструкторов. Методы класса Currency перечислены в табл. 18.10. В следующей программе демонстрируется использование Currency.

```
// Демонстрация применения 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 дробных разрядов по умолчанию: " +
      c.getDefaultFractionDigits());
}
}
```

Вывод этой программы:

```
Символ: $
Количество дробных разрядов по умолчанию: 2
```

Таблица 18.10. Методы, определенные в классе `Currency`

Метод	Описание
`String getCurrencyCode()`	Возвращает код валюты в стандарте ISO 4217.
`int getDefaultFractionDigits()`	Возвращает количество десятичных знаков после точки, которые обычно используются с данной валютой. Например, для доллара это будет 2 знака.
`static Currency getInstance(Locale` *`localeObj`*`)`	Возвращает объект `Currency` для локали, указанной в *`localeObj`*.
`static Currency getInstance(String` *`code`*`)`	Возвращает объект `Currency`, ассоциированный с кодом валюты, переданным в *`code`*.
`String getSymbol()`	Возвращает символ валюты (такой как $) для вызывающего объекта.
`String getSymbol(Locale` *`localeObj`*`)`	Возвращает символ валюты (такой как $) для локали, указанной в *`localeObj`*.
`String toString()`	Возвращает код валюты вызывающего объекта.

Formatter

С появлением JDK 5 в Java добавилось средство, в котором давно нуждались программисты: возможность просто создавать форматированный вывод. С самого начала Java предоставляла богатый и разнообразный программный интерфейс, однако все же не всегда она предлагала простой способ создать форматированный текстовый вывод, особенно для числовых значений. Такие классы, как `NumberFormat`, `DateFormat` и `MessageFormat`, которые предлагались в более ранних версиях Java, имели богатые возможности форматирования, но они были не слишком удобны в применении. Более того, в отличие от C и C++, которые поддерживают широко известное и используемое семейство функций `printf()`, предлагающее простой способ форматирования вывода, Java до сих пор не имела таких средств. Одна из причин этого заключалась в том, что форматирование в стиле `printf()` требует использования списков аргументов переменной длины (varargs), которые не поддерживались в Java вплоть до выхода JDK 5. Как только переменные списки аргументов стали доступными, стало возможным добавить форматировщик общего назначения.

В центре системы поддержки форматированного вывода находится класс `Formatter`. Он предлагает *преобразования формата*, позволяющие отображать числа, строки, время и даты практически в любом виде по вашему желанию. Он работает в манере, подобной функции C/C++ `printf()`, из чего следует, что если вы знакомы с C/C++, то изучение класса `Formatter` для вас будет очень простым. Это также упрощает преобразование кода C/C++ в Java. Если вы не знакомы с C/C++, все равно будет достаточно просто научиться форматировать данные.

На заметку! *Несмотря на то что класс `Formatter` работает очень похоже на функцию C/C++ `printf()`, существуют некоторые отличия и усовершенствования. Таким образом, даже если вы имеете опыт использования C/C++, все равно рекомендуется внимательно прочитать этот раздел.*

Конструкторы Formatter

Прежде чем можно будет использовать Formatter для форматирования вывода, вы должны создать объект Formatter. В общем случае, Formatter работает, преобразуя бинарную форму данных, используемых программой, в форматированный текст. Он явно сохраняет форматированный текст в буфере, содержимое которого может быть доступно вашей программе в любой момент, когда понадобится. Можно позволить Formatter создавать этот буфер автоматически, или же вы можете указать этот буфер явно при создании объекта Formatter. Можно также заставить Formatter направлять свой буфер в файл.

Класс Formatter определяет много конструкторов, позволяющих создавать объекты Formatter разными способами. Вот примеры:

```
Formatter()
Formatter(Appendable buf)
Formatter(Appendable buf, Locale loc)
Formatter(String filename) throws FileNotFoundException
Formatter(String filename, String charset)
  throws FileNotFoundException, UnsupportedEncodingException
Formatter(File outF) throws FileNotFoundException
Formatter(OutputStream outStrm)
```

Здесь *buf* указывает буфер для форматированной строки. Если *buf* равен *null*, то Formatter автоматически распределяет объект StringBuffer для хранения отформатированной строки. Параметр *loc* указывает локаль. Если локаль не задана, используется локаль по умолчанию. Параметр *filename* специфицирует имя файла, который будет принимать форматированный вывод. Параметр *charset* указывает набор символов. Если не указано никакого набора символов, то используется набор символов по умолчанию. Параметр *outF* представляет собой ссылку на открытый файл, который должен принимать вывод. Параметр *outStrm* задает ссылку на выходной поток, куда будет направлен вывод. Когда используется файл, вывод также пишется в файл.

Возможно, наиболее широко применяется первый конструктор, который не имеет параметров. Он автоматически использует локаль по умолчанию и распределяет StringBuffer для хранения отформатированного вывода.

Методы Formatter

Formatter определяет методы, перечисленные в табл. 18.11.

Таблица 18.11. Методы, определенные в классе Formatter

Метод	Описание
void close()	Закрывает вызываемый Formatter. Это приводит к тому, что все ресурсы, используемые объектом, освобождаются. После закрытия Formatter больше использовать нельзя. Попытка использования закрытого Formatter приводит к возбуждению исключения FormatterClosedException.
void flush()	Сбрасывает буфер формата. Это приводит к тому, что весь вывод, находящийся в буфере, записывается в адресате. В основном используется к Formatter, примененному к файлу.

Окончание табл. 18.11

Метод	Описание
`Formatter format(String fmtString, Object ... args)`	Форматирует аргументы, переданные в *args*, в соответствие со спецификаторами формата, содержащимися в *fmtString*. Возвращает вызываемый объект.
`Formatter format(Locale loc, String fmtString, Object ... args)`	Форматирует аргументы, переданные в *args*, в соответствие со спецификаторами формата, содержащимися в *fmtString*. При форматировании используется локаль, определенная в *loc*. Возвращает вызываемый объект.
`IOException ioException()`	Если лежащий в основе объект, указанный в качестве адресата, генерирует `IOException`, возвращается это исключение. В противном случае возвращается `null`.
`Locale locale()`	Возвращает локаль вызывающего объекта.
`Appendable out()`	Возвращает ссылку на лежащий в основе объект, который назначен в качестве адресата для вывода.
`String toString()`	Возвращает объект `String`, содержащий форматированный вывод.

Основы форматирования

После того, как вы создали `Formatter`, его можно применять для создания форматированных строк. Чтобы сделать это, используйте метод `format()`. Его наиболее часто используемая версия показана ниже:

```
Formatter format(String fmtString, Object ... args)
```

Строка *fmtString* состоит из элементов двух типов. Первый тип состоит из символов, которые просто копируются в выходной буфер. Второй тип содержит *спецификаторы формата*, определяющие способ, в соответствие с которым должны отображаться последующие аргументы.

В простейшей форме спецификатор формата начинается со знака процента с последующим *спецификатором преобразования*. Все спецификаторы преобразования формата состоят из единственного символа. Например, спецификатор формата для числа с плавающей точкой выглядит как `%f`. В общем случае, должно быть столько аргументов, сколько есть спецификаторов формата, и соответствие аргументов устанавливается слева направо. Например, рассмотрим следующий фрагмент:

```
Formatter fmt = new Formatter();
fmt.format("Форматировать %s очень просто: %d %f", "с помощью Java", 10, 98.6);
```

Приведенная кодовая последовательность создает объект `Formatter`, содержащий следующую строку:

```
Форматировать с помощью Java очень просто: 10 98.600000
```

В этом примере спецификаторы формата `%s`, `%d` и `%f` замещаются аргументами, следующими за строкой формата. То есть `%s` заменяется на `"с помощью Java"`, `%d` — на `10`, а `%f` — на `98.6`. Все остальные символы используются, как есть. Как вы можете ожидать, спецификатор параметра `%s` указывает строку, а `%d` — целое число. Как уже упоминалось, `%f`, означает число с плавающей точкой.

Таблица 18.12. Спецификаторы формата

Спецификатор формата	Применяемое преобразование
%a %A	Шестнадцатеричное с плавающей точкой
%b %B	Булевское
%c	Символ
%d	Десятичное целое
%h %H	Хеш-код аргумента
%e %E	Научная нотация
%f	Десятичное с плавающей точкой
%g %G	Использует либо %e, либо %f, смотря что короче
%o	Восьмеричное целое
%n	Вставляет символ перевода строки
%s %S	Строка
%t %T	Время и дата
%x %X	Шестнадцатеричное целое
%%	Вставляет символ %

Метод format() принимает широкое разнообразие спецификаторов формата, которые показаны в табл. 18.12. Обратите внимание, что многие спецификаторы имеют и заглавную, и прописную формы. Когда используется заглавная форма, буквы показываются в верхнем регистре. Во всем остальном формы эквивалентны. Важно понимать, что Java проверяет каждый спецификатор формата на соответствие типу аргумента. Если аргумент не соответствует, возбуждается исключение IllegalFormatException.

После того, как вы отформатируете строку, вы можете получить ее методом toString(). Например, в продолжение предыдущего примера: следующий оператор получает форматированную строку, содержащуюся в fmt:

```
String str = fmt.toString();
```

Конечно, если вы просто хотите отобразить форматированную строку, нет причин вначале присваивать ее объекту String. Например, когда объект Formatter передается println(), то автоматически вызывается его метод toString().

Ниже приведена короткая программа, которая собирает все вместе, демонстрируя, как создавать и отображать отформатированную строку.

```
// Очень простой пример применения Formatter.
import java.util.*;
class FormatDemo {
```

```
public static void main(String args[]) {
  Formatter fmt = new Formatter();
  fmt.format("Форматирование %s просто %d %f", " с Java", 10, 98.6);
  System.out.println(fmt);
}
}
```

Еще один момент: вы можете получить ссылку на выходной буфер, вызывая `out()`. Он возвращает ссылку на объект `Appendable`.

Теперь, когда вы знакомы с общим механизмом создания форматированных строк, остальная часть этого раздела посвящена подробному описанию каждого преобразования. Она также описывает различные опции, такие как выравнивание, минимальная ширина поля и точность.

Форматирование строк и символов

Для форматирования индивидуального символа используйте `%c`. Это заставит соответствующий символьный аргумент выводиться без каких-либо модификаций. Чтобы отформатировать строку, применяйте `%s`.

Форматирование чисел

Чтобы форматировать целые числа в десятичном формате, используйте `%d`. Чтобы форматировать значения с плавающей точкой в десятичном формате, применяйте `%f`. Для форматирования значений с плавающей точкой в научной нотации указывайте `%e`. Числа, представленные в научной нотации, принимают следующую общую форму:

x.ddddddе+/–yy

Спецификатор формата `%g` заставляет `Formatter` использовать либо `%f`, либо `%e`, в зависимости от того, что получается короче. В следующей программе демонстрируется эффект спецификатора формата `%g`.

```
// Демонстрирует применение спецификатора формата %g.
import java.util.*;
class FormatDemo2 {
public static void main(String args[]) {
  Formatter fmt = new Formatter();
  for(double i=1000; i < 1.0e+10; i *= 100) {
     fmt.format("%g ", i);
     System.out.println(fmt);
  }
}
}
```

Эта программа дает такой вывод:

```
1000.000000
1000.000000 100000.000000
1000.000000 100000.000000 1.000000e+07
1000.000000 100000.000000 1.000000e+07 1.000000e+09
```

Вы можете отображать целые числа в восьмеричном или шестнадцатеричном формате, используя соответственно `%o` или `%x`. Например, следующий фрагмент:

```
fmt.format("Шестнадцатеричное: %x, восьмеричное: %o", 196, 196);
```

генерирует такой вывод:

```
Шестнадцатеричное: c4, восьмеричное: 304
```

Вы можете отображать числа с плавающей точкой в шестнадцатеричном формате, используя %a. Формат, генерируемый %a, на первый взгляд может показаться странным. Дело в том, что его представление использует форму, подобную научной нотации, состоящей из мантиссы и экспоненты, обе — в шестнадцатеричном формате. Вот как выглядит общий формат:

```
0x1.sigpexp
```

Здесь *sig* содержит дробную часть мантиссы, а *exp* — экспоненту. p указывает начало экспоненты. Например, следующий вызов:

```
fmt.format("%a", 123.123);
```

генерирует такой вывод:

```
0x1.ec7df3b645a1dp6
```

Форматирование времени и даты

Одно из наиболее мощных преобразований задается с помощью спецификатора формата %t. Он позволяет форматировать информацию о времени и дате. Спецификатор %t работает несколько иначе, чем другие, поскольку требует применения суффиксов для описания части и точности требуемого времени и даты. Суффиксы перечислены в табл. 18.13. Например, чтобы отобразить минуты, вы должны использовать %tM, где M означает минуты в двухсимвольном поле. Аргументы, относящиеся к спецификатору %t, должны иметь тип Calendar, Date, Long или long.

Ниже приведена программа, демонстрирующая применение некоторых форматов.

```
// Форматирования времени и даты.
import java.util.*;
class TimeDateFormat {
public static void main(String args[]) {
  Formatter fmt = new Formatter();
  Calendar cal = Calendar.getInstance();

  // Отобразить стандартный 12-часовой формат.
  fmt.format("%tr", cal);
  System.out.println(fmt);

  // Отобразить полную информацию о дате и времени.
  fmt = new Formatter();
  fmt.format("%tc", cal);
  System.out.println(fmt);

  // Отобразить только часы и минуты.
  fmt = new Formatter();
  fmt.format("%tl:%tM", cal, cal);
  System.out.println(fmt);

  // Отобразить название и номер месяца.
  fmt = new Formatter();
  fmt.format("%tB %tb %tm", cal, cal, cal);
  System.out.println(fmt);
}
}
```

Пример вывода:

```
09:17:15 AM
Mon Jan 01 09:17:15 CST 2007
9:17
January Jan 01
```

Таблица 18.13. 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, отформатированная как *день месяц дата чч::мм:сс пояс год*
C	Первые два знака года
d	День месяца, как десятичное число (01–31)
D	месяц/день/год
e	День месяца, как десятичное число (1–31)
F	год-месяц-день
h	Сокращенное название месяца
H	Часы (от 00 до 23)
I	Часы (от 01 до 12)
j	День года, как десятичное число (от 001 до 366)
k	Часы (от 0 до 23)
l	Часы (от 1 до 12)
L	Миллисекунды (от 000 до 999)
m	Месяц, как десятичное число (от 01 до 13)
M	Минуты, как десятичное число (от 00 до 59)
N	Наносекунды (от 000000000 до 999999999)
p	Локальный эквивалент AM или PM в нижнем регистре
Q	Миллисекунды, прошедшие с 01/01/1970
r	*чч:мм* (12-часовой формат)
R	*чч:мм* (24-часовой формат)
S	Секунды (от 00 до 60)
s	Секунды, прошедшие с 01/01/1970 UTC
T	*чч:мм:СС* (24-часовой формат)
y	Годы в десятичных числах без века (от 00 до 99)
Y	Годы в десятичных числах, включая век (от 0001 до 9999)
z	Смещение от UTC
Z	Наименование часового пояса

Спецификаторы %n и %%

Спецификаторы формата %n и %% отличаются от других в том, что они не соответствуют аргументу. Вместо этого они просто представляют собой управляющие последовательности, которые вставляют символ в выходную последовательность. %n вставляет перевод строки, а %% — знак процента. Ни один из этих символов не может быть введен непосредственно в форматирующую строку. Конечно, вы можете также использовать стандартную управляющую последовательность \n, чтобы вставить знак перевода строки.

Ниже приведен пример, демонстрирующий использование спецификаторов формата %n и %%.

```
// Демонстрация применения спецификаторов формата %n и %%.
import java.util.*;
class FormatDemo3 {
public static void main(String args[]) {
  Formatter fmt = new Formatter();
  fmt.format("Копирование файла%nПеремещение на %d%% завершено", 88);
  System.out.println(fmt);
}
}
```

Программа отображает следующий вывод:

```
Копирование файла
Перемещение на 88% завершено
```

Указание минимальной ширины поля

Целое число, помещенное между символом % и кодом преобразования формата, выступает в качестве *спецификатора минимальной ширины*. Он дополняет вывод пробелами, чтобы обеспечивать заданную минимальную длину. Если строка или число получаются длиннее, чем этот заданный минимум, они будут напечатаны полностью. По умолчанию дополнение осуществляется пробелами. Если вы хотите дополнять нулями, поместите 0 перед спецификатором ширины поля. Например, %05d дополнит число, состоящее из менее чем 5 разрядов, нулями — таким образом, чтобы его общая ширина была равна пяти. Спецификатор ширины поля может применяться вместе со всеми спецификаторами формата, кроме %n.

В следующей программе демонстрируется использование спецификатора минимальной ширины поля путем добавления его к спецификатору преобразования %f.

```
// Демонстрация применения 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);
}
}
```

Эта программа генерирует следующий вывод:

```
|10.123450|
|   10.123450|
|00010.123450|
```

Первая строка отображает число 10,12345 с шириной по умолчанию. Вторая отображает это значение в 12-символьном поле. Третья строка отображает значение в 12-символьном поле, дополняя его ведущими нулями.

Минимальный модификатор ширины поля часто используется для генерации таблиц, состоящих из строк и столбцов. Например, следующая программа выдает таблицу квадратов и кубов чисел от 1 до 10.

```
// Создает таблицу квадратов и кубов.
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);
   }
}
}
```

Ее вывод показан ниже:

```
 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 может быть применим к спецификаторам формата `%f`, `%e`, `%g` и `%s`. Он следует за спецификатором минимальной ширины поля (если таковой имеется) и состоит из точки с последующим целым числом. Его конкретное значение зависит от типа данных, к которому он применяется.

Когда вы применяете спецификатор точности к данным с плавающей точкой с применением спецификаторов преобразования `%f` или `%e`, то он определяет количество отображаемых десятичных разрядов. Например, `%10.4f` отображает число, по меньшей мере, в 10 символов шириной с четырьмя разрядами после запятой. При использовании `%g` точность определяет количество значащих десятичных разрядов. Точность по умолчанию составляет 6 знаков после запятой.

В применении к строкам спецификатор точности задает максимальную ширину поля. Например, `%5.7s` отображает строку длиной минимум в пять символов и не превышающей семь символов. Если строка длиннее максимальной ширины, конечные символы будут усечены.

В следующей программе демонстрируется работа с модификатором точности.

```
// Демонстрация применения модификатора точности.
import java.util.*;
class PrecisionDemo {
```

```
public static void main(String args[]) {
  Formatter fmt = new Formatter();

  // Формат с 4 десятичными разрядами.
  fmt.format("%.4f", 123.1234567);
  System.out.println(fmt);

  // Формат с 2 десятичными разрядами в 16-символьном поле.
  fmt = new Formatter();
  fmt.format("%16.2e", 123.1234567);
  System.out.println(fmt);

  // Отобразить максимум 15 символов строки.
  fmt = new Formatter();
  fmt.format("%.15s", "Форматировать в Java теперь очень просто.");
  System.out.println(fmt);
}
}
```

Она выдает такой вывод:

```
123.1235
    1.23e+02
Форматировать в
```

Использование флагов формата

`Formatter` распознает набор *флагов* формата, которые позволяют вам управлять различными аспектами преобразования. Все флаги формата — одиночные символы, и флаг формата следует за знаком `%` в спецификаторе формата. Флаги перечислены в табл. 18.14.

Таблица 18.14. Флаги формата

Флаг	Эффект
`-`	Выравнивание влево
`#`	Альтернативный формат преобразования
`0`	Вывод дополняется нулями вместо пробелов
пробел	Положительным числам предшествует пробел
`+`	Положительным числам предшествует знак +
`,`	Числовые значения, включающие групповые разделители
`(`	Отрицательные числовые значения заключены в скобки

Не все флаги применимы ко всем спецификаторам формата. В следующих разделах они объясняются более подробно.

Выравнивание вывода

По умолчанию весь вывод выравнивается вправо. То есть если ширина поля больше, чем напечатанные данные, то данные будут размещены в правой части поля. Вы можете выравнивать значения влево, поместив знак минуса сразу после `%`. Например, `%-10.2f` выравнивает влево число с плавающей точкой с двумя разрядами после десятичной точки в пределах 10-символьного поля.

Например, рассмотрим следующую программу:

```
// Демонстрация выравнивания влево.
import java.util.*;
class LeftJustify {
public static void main(String args[]) {
     Formatter fmt = new Formatter();

     // По умолчанию выравнивается вправо
     fmt.format("|%10.2f|", 123.123);
     System.out.println(fmt);

     // А теперь влево.
     fmt = new Formatter();
     fmt.format("|%-10.2f|", 123.123);
     System.out.println(fmt);
}
}
```

Она выдает такой результат:

```
|    123.12|
|123.12     |
```

Как видите, вторая строка выровнена влево в пределах 10-символьного поля.

Флаги пробела, +, 0 и (

Чтобы заставить отображать знак + перед положительными числовыми значениями, добавьте флаг +, например:

```
fmt.format("%+d", 100);
```

порождает такую строку:

```
+100
```

Когда создаются столбцы чисел, иногда бывает удобно выводить пробел перед положительными числами, чтобы положительные и отрицательные значения выводились в столбик. Чтобы достичь этого, добавьте флаг пробела. Например:

```
// Демонстрация применения 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 = new Formatter();
     fmt.format("% d", 100);
     System.out.println(fmt);
     fmt = new Formatter();
     fmt.format("% d", -200);
     System.out.println(fmt);
     fmt = new Formatter();
     fmt.format("% d", 200);
     System.out.println(fmt);
}
}
```

Результат показан ниже:

```
-100
 100
-200
 200
```

Обратите внимание, что положительные значения имеют ведущий пробел, что обеспечивает ровное расположение разрядов в столбце.

Чтобы отобразить отрицательные числовые значения в скобках вместо добавления ведущего минуса, используйте флаг (. Например:

```
fmt.format("%(d", -100);
```

дает следующий результат:

```
(100)
```

Флаг 0 заставляет дополнять вывод нулями вместо пробелов.

Флаг “запятая”

При отображении больших чисел часто бывает удобно добавлять разделители групп, которыми в английском окружении являются запятые. Например, значение 1234567 легче читается, когда оно отформатировано в виде 1,234,567. Для добавления спецификаторов группирования служит флаг “запятая” (,). Например:

```
fmt.format("%,.2f", 4356783497.34);
```

выдает такую строку:

```
4,356,783,497.34
```

Флаг

Флаг # может применяться к %o, %x, %e и %f. Для %e и %f флаг # обеспечивает наличие десятичной точки даже в случае, если нет дробных разрядов. Если вы перед спецификатором формата %x поставите #, то шестнадцатеричное число будет напечатано с префиксом 0x. Если предварить флагом # спецификатор %x, число будет печататься с ведущим нулем.

Опция верхнего регистра

Как упоминалось ранее, некоторые из спецификаторов формата имеют версии в верхнем регистре, которые заставляют при преобразовании применять буквы верхнего регистра, где это возможно. В табл. 18.15 описывается упомянутый эффект.

Например, следующий вызов

```
fmt.format("%X", 250);
```

порождает строку

```
PA
```

Следующий вызов:

```
fmt.format("%E", 123.1234);
```

дает в результате строку

```
1.231234E+02
```

Таблица 18.15. Опции верхнего регистра

Спецификатор	Эффект
%A	Заставляет шестнадцатеричные цифры от *a* до *f* отображаться в верхнем регистре, т.е. от *A* до *F*. Кроме того, префикс 0x отображается как 0X, а p — как P.
%B	Переводит в верхний регистр значения true и false.
%E	Заставляет символ экспоненты *e* отображаться в верхнем регистре.
%G	Заставляет символ экспоненты *e* отображаться в верхнем регистре.
%H	Заставляет шестнадцатеричные цифры от *a* до *f* отображаться в верхнем регистре, от *A* до *F*.
%S	Переводит соответствующую спецификатору строку в верхний регистр.
%T	Заставляет алфавитный вывод отображаться в верхнем регистре.
%X	Заставляет шестнадцатеричные цифры от *a* до *f* отображаться в верхнем регистре, от *A* до *F*. Кроме того, префикс 0x отображается как 0X, если таковой присутствует.

Использование индекса аргументов

Formatter включает очень удобное средство, позволяющее указать аргумент, к которому должен применяться конкретный спецификатор формата. Обычно порядок аргументов и спецификаторов формата совпадает — слева направо. То есть первый спецификатор формата относится к первому аргументу, второй — ко второму аргументу, и т.д. Однако, используя индекс аргумента, вы можете явно управлять тем, к какому из аргументов относится спецификатор формата.

Индекс аргумента следует сразу за % в спецификаторе формата. Он имеет следующий вид:

```
n$
```

Здесь *n* — индекс нужного аргумента, начиная с 1. Рассмотрим следующий пример.

```
fmt.format("%3$d %1$d %2$d", 10, 20, 30);
```

Этот код порождает строку:

```
30 10 20
```

В этом примере первый спецификатор формата соответствует 30, второй — 10, а третий — 20. То есть аргументы используются в порядке, отличном от простого порядка "слева направо".

Одной из выгод индексирования аргументов является то, что оно позволяет повторно использовать аргумент, не указывая его дважды. Например, рассмотрим следующую строку:

```
fmt.format("%d в шестнадцатеричном формате равно %1$x", 255);
```

она порождает следующий результат:

```
255 в шестнадцатеричном формате равно ff
```

Как видите, аргумент 255 используется с обоими спецификаторами формата.

Существует удобное сокращение, называемое *относительным индексом*, которое позволяет повторно использовать аргументы, соответствующие предшествующему спецификатору формата. Просто укажите < вместо индекса аргумента. Например, следующий вызов format() порождает тот же результат, что и предыдущий пример.

```
fmt.format("%d в шестнадцатеричном формате равно %<x", 255);
```

Относительные индексы особенно удобны при создании пользовательских форматов времени и даты.

Рассмотрим следующий пример:

```
// Использование относительных индексов для упрощения
// создания 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);
}
}
```

Вот как выглядит вывод:

```
Today is day 1 of Jan, 2007
```

Благодаря относительной индексации, аргумент cal может быть передан только один раз вместо трех.

Подключение Java-функции printf()

Хотя нет ничего технически неправильного в непосредственном применении Formatter (как это делалось в предыдущих примерах) при генерации вывода для отображения на консоли, существует более удобная альтернатива: метод printf(). Метод printf() автоматически использует Formatter для создания форматированной строки. Затем она отправляет эту строку в System.out, который по умолчанию представляет собой консоль. Метод printf() определен и в PrintStream и в PrintWriter. Метод printf() рассматривается в главе 19.

Scanner

Класс Scanner — дополнение к Formatter. Добавленный в JDK 5, Scanner читает форматированный ввод и преобразует его в бинарную форму. Хотя всегда есть возможность прочитать форматированный ввод, это требует больших усилий, чем большинство программистов готовы приложить. Благодаря добавлению Scanner, упростилось чтение всех типов числовых значений, строк и данных других типов, независимо от того, получены они из дискового файла, с клавиатуры либо из другого источника.

Scanner может применяться для чтения ввода с консоли, из файла, из строки или с другого источника, реализующего интерфейс Readable либо ReadableByteChannel. Например, вы можете использовать Scanner для чтения числа с клавиатуры и присвоения его значения переменной. Как вы увидите, несмотря на свою мощь, Scanner неожиданно прост в применении.

Конструкторы Scanner

Scanner определяет конструкторы, перечисленные в табл. 18.16. В общем случае, Scanner может быть создан для String, InputStream, File или любого другого объекта, реализующего интерфейсы Readable либо ReadableByteChannel. Ниже приведены некоторые примеры.

Следующая последовательность создает Scanner, который читает Test.txt:

```
FileReader fin = new FileReader("Test.txt");
Scanner src = new Scanner(fin);
```

Это работает, потому что FileReader реализует интерфейс Readable. То есть вызов конструктора интерпретируется как Scanner(Readable).

Таблица 18.16. Конструкторы Scanner

Конструктор	Описание
Scanner(File *from*) throws FileNotFoundException	Создает Scanner, который использует указанный файл *from* в качестве входного источника.
Scanner(File *from*, String *charset*) throws FileNotFoundException	Создает Scanner, который использует указанный файл *from* с кодировкой, заданной в *charset*, в качестве входного источника.
Scanner(InputStream *from*)	Создает Scanner, который использует указанный поток *from* в качестве входного источника.
Scanner(InputStream *from*, String *charset*)	Создает Scanner, который использует указанный поток *from* с кодировкой, заданной в *charset*, в качестве входного источника.
Scanner(Readable *from*)	Создает Scanner, который использует указанный объект Readable *from* в качестве входного источника.
Scanner (ReadableByteChannel *from*)	Создает Scanner, который использует указанный объект ReadableByteChannel *from* в качестве входного источника.
Scanner(ReadableByteChannel *from*, String *charset*)	Создает Scanner, который использует указанный объект ReadableByteChannel *from* с кодировкой, заданной в *charset*, в качестве входного источника.
Scanner(String *from*)	Создает Scanner, который использует указанную строку в качестве входного источника.

Следующая строка создает Scanner, читающий из стандартного ввода, которым по умолчанию является клавиатура:

```
Scanner conin = new Scanner(System.in);
```

Это работает, потому что System.in — это объект типа InputStream. То есть вызов конструктора отображается как Scanner(InputStream).

Следующая последовательность создает Scanner, который читает строку:

```
String instr = "10 99.88 сканирование очень просто.";
Scanner conin = new Scanner(instr);
```

Основы сканирования

После того, как вы создаете Scanner, его очень просто использовать для чтения форматированного ввода. В общем случае, Scanner читает *лексемы* из некоторого лежащего в основе источника, который вы указываете при создании объекта Scanner. С точки зрения Scanner *лексема* — это порция ввода, отделенная набором разделителей, которыми по умолчанию являются пробелы. Лексема читается в соответствии с определенным *регулярным выражением*, задающим формат данных. Хотя Scanner позволяет определить специфический тип выражения, которому будет соответствовать следующая операция ввода, он включает множество предопределенных шаблонов, соответствующих примитивным типам, таким как int и double, а также строкам. То есть часто вы не будете нуждаться в указании шаблонов.

В общем случае для использования Scanner следуйте описанной ниже процедуре.

1. Определите, доступен ли специфический тип ввода вызовом одного из методов Scanner hasNext*X*, где *X* — нужный тип данных.
2. Если ввод доступен, читайте его одним из методов Scanner next*X*.
3. Повторяйте процесс до завершения ввода.

Как следует из вышесказанного, Scanner определяет два набора методов, которые позволяют читать ввод. Первый — это методы hasNext*X*, перечисленные в табл. 18.17. Эти методы определяют, доступен ли указанный тип ввода. Например, вызов hasNextInt() возвращает true только в том случае, если следующая лексема, подлежащая чтению, является целым числом. Если требуемые данные доступны, вы читаете их одним из методов Scanner next*X*, описанных в таблице 18.18. Например, чтобы прочесть следующее целое число, вызывайте nextInt(). Приведенная ниже последовательность показывает, как читать список целых чисел, вводимых с клавиатуры.

```
Scanner conin = new Scanner(System.in);
int i;
// Читать список целых.
while(conin.hasNextInt()) {
   i = conin.nextInt();
   // ...
}
```

В приведенном выше примере цикл while остановится, как только следующая лексема не будет целым числом. То есть цикл прекращает читать целые, как только во входном потоке обнаруживается значение, отличное от типа целого числа.

Если метод next не может найти тип данных, который он ожидает, он возбуждает исключение NoSuchElementException. По этой причине сначала лучше проверить, что данные требуемого типа доступны, с помощью метода hasNext, прежде чем вызывать соответствующий ему метод next.

Таблица 18.17. Методы hasNext класса Scanner

Метод	Описание
boolean hasNext()	Возвращает true, если доступна для чтения следующая лексема любого типа. В противном случае возвращает false.
boolean hasNext(Pattern *pattern*)	Возвращает true, если доступна для чтения лексема, соответствующая шаблону *pattern*. В противном случае возвращает false.

Окончание табл. 18.17

Метод	Описание
`boolean hasNext(String pattern)`	Возвращает `true`, если доступна для чтения лексема, соответствующая шаблону *pattern*. В противном случае возвращает `false`.
`boolean hasNextBigDecimal()`	Возвращает `true`, если доступно для чтения значение, которое можно поместить в объект `BigDecimal`. В противном случае возвращает `false`.
`boolean hasNextBigInteger()`	Возвращает `true`, если доступно для чтения значение, которое можно поместить в объект `BigInteger`. В противном случае возвращает `false`. Используется основание по умолчанию. (Если только его не изменить, основание по умолчанию равно 10.)
`boolean hasNextBigInteger(int radix)`	Возвращает `true`, если доступно для чтения значение по основанию *radix*, которое можно поместить в объект `BigInteger`. В противном случае возвращает `false`.
`boolean hasNextBoolean()`	Возвращает `true`, если доступно для чтения значение типа `boolean`. В противном случае возвращает `false`.
`boolean hasNextByte()`	Возвращает `true`, если доступно для чтения значение типа `byte`. В противном случае возвращает `false`.
`boolean hasNextByte(int radix)`	Возвращает `true`, если доступно для чтения значение типа `byte` с указанным основанием *radix*. В противном случае возвращает `false`.
`boolean hasNextDouble()`	Возвращает `true`, если доступно для чтения значение типа `double`. В противном случае возвращает `false`.
`boolean hasNextFloat()`	Возвращает `true`, если доступно для чтения значение типа `float`. В противном случае возвращает `false`.
`boolean hasNextInt()`	Возвращает `true`, если доступно для чтения значение, которое можно поместить в объект `int`. В противном случае возвращает `false`. Используется основание по умолчанию. (Если только его не изменить, основание по умолчанию равно 10.)
`boolean hasNextInt(int radix)`	Возвращает `true`, если доступно для чтения значение типа `int` с указанным основанием *radix*. В противном случае возвращает `false`.
`boolean hasNextLine()`	Возвращает `true`, если доступна строка ввода.
`boolean hasNextLong()`	Возвращает `true`, если доступно для чтения значение типа `long`. В противном случае возвращает `false`. Используется основание по умолчанию. (Если только его не изменить, основание по умолчанию равно 10.)
`boolean hasNextLong(int radix)`	Возвращает `true`, если доступно для чтения значение `long` по основанию *radix*. В противном случае возвращает `false`.
`boolean hasNextShort()`	Возвращает `true`, если доступно для чтения значение типа `short`. В противном случае возвращает `false`. Используется основание по умолчанию. (Если только его не изменить, основание по умолчанию равно 10.)
`boolean hasNextShort(int radix)`	Возвращает `true`, если доступно для чтения значение `short` по основанию *radix*. В противном случае возвращает `false`.

Таблица 18.18. Методы next класса Scanner

Метод	Описание
`String next()`	Возвращает следующую лексему любого типа из входного источника.
`String next(Pattern pattern)`	Возвращает следующую лексему, соответствующую шаблону, переданному в *pattern*, из входного источника.
`String next(String pattern)`	Возвращает следующую лексему, соответствующую шаблону, переданному в *pattern*, из входного источника.
`BigDecimal nextBigDecimal()`	Возвращает следующую лексему как объект `BigDecimal`.
`BigInteger nextBigInteger()`	Возвращает следующую лексему как объект `BigInteger`. Используется основание по умолчанию (если не изменено, равное 10).
`BigInteger nextBigInteger(int radix)`	Возвращает следующую лексему как объект `BigInteger`. Используется основание *radix*.
`boolean nextBoolean()`	Возвращает следующую лексему как значение `boolean`.
`byte nextByte()`	Возвращает следующую лексему как значение `byte`. Используется основание по умолчанию (если не изменено, равное 10).
`byte nextByte(int radix)`	Возвращает следующую лексему как значение `byte` по основанию *radix*.
`double nextDouble()`	Возвращает следующую лексему как значение `double`.
`float nextFloat()`	Возвращает следующую лексему как значение `float`.
`int nextInt()`	Возвращает следующую лексему как значение `int`. Используется основание по умолчанию (если не изменено, равное 10).
`int nextInt(int radix)`	Возвращает следующую лексему как значение `int` по основанию *radix*.
`String nextLine()`	Возвращает следующую строку ввода.
`long nextLong()`	Возвращает следующую лексему как значение `long`. Используется основание по умолчанию (если не изменено, равное 10).
`long nextLong(int radix)`	Возвращает следующую лексему как значение `long` по основанию *radix*.
`short nextShort()`	Возвращает следующую лексему как значение `short`. Используется основание по умолчанию (если не изменено, равное 10).
`short nextShort(int radix)`	Возвращает следующую лексему как значение `short` по основанию *radix*.

Некоторые примеры применения `Scanner`

Добавление в Java класса `Scanner` сделало ранее утомительную задачу совершенно простой. Чтобы понять почему, давайте рассмотрим несколько примеров. Следующая программа подсчитывает среднее из списка чисел, введенных с клавиатуры.

```
// Применение Scanner для вычисления среднего из списка 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 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;
       }
     }
  }
  System.out.println("Среднее равно " + sum / count);
}
}
```

Эта программа читает числа с клавиатуры, суммирует их в процессе, до тех пор, пока пользователь не введет строку “готово”. Затем она прекращает ввод и отображает среднее значение введенных чисел.

Ниже показан пример ее выполнения:

```
Введите числа для подсчета среднего.
1.2
2
3.4
4
готово
Среднее равно 2.65
```

Программа читает числа до тех пор, пока не получит лексему, которую нельзя интерпретировать как корректное значение типа `double`. Когда подобное происходит, она проверяет, что лексема равна строке “готово”. Если это так, программа завершается нормально. В противном случае она отображает сообщение об ошибке.

Обратите внимание, что числа читаются с помощью `nextDouble()`. Этот метод читает любые числа, которые могут быть преобразованы в `double`, включая целые значения вроде `2` и значения с плавающей точкой, такие как `3.4`. То есть число, прочитанное

nextDouble(), не требует наличия десятичной точки. Тот же общий принцип применим к любому методу next. Они найдут соответствие и прочитают данные в любом формате, который может представлять данные запрашиваемого типа.

Еще один симпатичный момент, касающийся Scanner — это то, что одна и та же техника, используемая для чтения одного источника, может быть применена для другого. Например, ниже представлена измененная версия предшествующей программы, предназначенная для чтения списка чисел из файла.

```
// Использование Scanner для вычисления среднего значения числе 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;

   // Записать вывод 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 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;
       }
     }
   }
   fin.close();

   System.out.println("Среднее равно " + sum / count);
}
}
```

Вот вывод:

```
Среднее равно 6.2
```

Вы можете использовать Scanner для чтения ввода, который содержит некоторые различные типы данных, даже если порядок их следования заранее не известен. Вы просто должны проверять, какого типа доступны данные, прежде чем читать их. Например, рассмотрим следующую программу:

```
// Применение Scanner для чтения данных разного типа 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;

  // Писать вывод 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 до конца.
  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);
     }
  }

  fin.close();
}
}
```

Ниже показан результат.

```
String: Тестирование
String: Scanner
int: 10
double: 12.2
String: один
boolean: true
String: два
boolean: false
```

При чтении данных смешанных типов, как это делает предыдущая программа, вы должны быть немного более внимательными относительно порядка, в котором вызываются методы `next`. Например, если в цикле поменять порядок вызовов `nextInt()` и `nextDouble()`, то оба числовых значения будут прочитаны как `double`, поскольку `nextDouble()` соответствует любой строке, содержащей число, которое может быть интерпретировано как `double`.

Установка разделителей

Scanner определяет, где начинаются и заканчиваются лексемы, на основе набора *разделителей*. По умолчанию разделителями являются пробельные символы, и именно этот набор разделителей используется в предыдущем примере. Однако можно изменить разделители, вызвав метод useDelimiters(), показанный ниже:

```
Scanner useDelimiter(String pattern)
Scanner useDelimiter(Pattern pattern)
```

Здесь *pattern* — регулярное выражение, которое определяет набор разделителей.

Ниже показана измененная версия приведенной ранее программы, которая читает список чисел, разделенных запятыми и любым количеством пробелов.

```
// Применение Scanner для вычисления среднего в списке
// разделенных запятыми значений.
import java.util.*;
import java.io.*;
class SetDelimiters {
public static void main(String args[]) throws IOException {
  int count = 0;
  double sum = 0.0;

  // Писать вывод в файл.
  FileWriter fout = new FileWriter("test.txt");

  // Теперь сохранить значения в списке, разделенном 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);

  // Установитьв качестве разделителей запятые и пробелы.
  src.useDelimiter(", *");

  // Читать и суммировать 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;
        }
     }
  }
  fin.close();
  System.out.println("Среднее равно " + sum / count);
}
}
```

В этой версии числа, записанные в test.txt, разделены запятыми и пробелами. Применение шаблона разделителей ",*" сообщает объекту Scanner, чтобы он воспринимал запятую и ноль или более пробелов в качестве разделителей. Вывод программы будет тем же, что и ранее.

Вы можете получить текущий шаблон разделителей вызовом метода `delimiter()`, показанного ниже:

```
Pattern delimiter()
```

Прочие возможности `Scanner`

`Scanner` определяет несколько других методов в дополнение к уже упомянутым. В частности, одним из наиболее часто используемых в некоторых случаях является `findInLine()`. Его общие формы представлены ниже:

```
String findInLine(Pattern pattern)
String findInLine(String pattern)
```

Этот метод ищет указанный шаблон внутри следующей строки текста. Если шаблон найден, соответствующая ему лексема принимается и возвращается. В противном случае возвращается `null`. Метод работает независимо от набора разделителей. Этот метод удобен, если требуется специфический шаблон. Например, следующая программа ищет поле возраста во входной строке и затем отображает его.

```
// Демонстрация применения 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 и отобразить возраст.
  conin.findInLine("Возраст:"); // найти "Возраст"
  if(conin.hasNext())
     System.out.println(conin.next());
  else
     System.out.println("Ошибка!");
}
}
```

Результатом будет `28`. В программе `findInLine()` применяется для поиска вхождения шаблона `"Возраст:"`. Когда он найден, читается следующая лексема, которая представляет собой значение возраста.

С `findInLine()` связан метод `findWithinHorixon()`. Он показан ниже:

```
String findWithinHorizon(Pattern pattern, int count)
String findWithinHorizon(String pattern, int count)
```

Этот метод пытается найти вхождение указанного шаблона в следующие *count* символов. В случае успеха метод возвращает совпадающий шаблон и `null` — в противном случае. Если *count* равен нулю, то поиск выполняется во всем вводе до тех пор, пока либо будет обнаружено совпадение, либо будет достигнут конец входной информации.

Вы можете пропустить шаблон, используя метод `skip()`, показанный ниже:

```
Scanner skip(Pattern pattern)
Scanner skip(String pattern)
```

Если соответствие *pattern* имеется, `skip()` просто пропускает его и возвращает ссылку на вызывающий объект. Если шаблон не найден, `skip()` возбуждает исключение `NoSuchElementException`.

Другие методы Scanner включают radix(), который возвращает основание системы счисления по умолчанию, используемое Scanner при чтении чисел, useRadix(), который устанавливает основание системы счисления, reset(), который сбрасывает сканер, и close(), закрывающий сканер.

Классы ResourceBundle, ListResourceBundle и PropertyResourceBundle

Пакет java.util включает три класса, предназначенные для интернационализации ваших программ. Первый из них — абстрактный класс ResourceBundle. Он определяет методы, позволяющие управлять коллекцией чувствительных к локали ресурсов, таких как строки, используемые в качестве меток элементов пользовательского интерфейса программ. Вы можете определить два или более набора переведенных строк, поддерживающих различные языки, скажем, английский, немецкий или китайский, причем каждый из переводов будет находиться в собственной связке (bundle). Затем вы можете загружать нужную связку в соответствии с текущими локальными установками и использовать строки для конструирования пользовательского интерфейса программы.

Связки ресурсов идентифицируются *именем семейства* (также называемыми их *базовыми именами*). К имени семейства может быть добавлен двухсимвольный *код языка*, указывающий на конкретный язык. В этом случае, если запрошенная локаль соответствует коду языка, то используется эта версия связки ресурсов. Например, связка ресурсов с именем семейства SampleRB может иметь немецкую версию SampleRB_de и русскую версию под названием SampleRB_ru. (Обратите внимание, что знак подчеркивания связывает имя семейства с кодом языка.) Таким образом, если текущая локаль — Locale.GERMAN, будет применяться связка SampleRB_de.

Также возможно указать специфические варианты языка, которые относятся к определенной стране, специфицируя *код страны* после коля языка. Код страны — это двухсимвольный идентификатор в верхнем регистре, такой как AU для Австрии или IN — для Индии. Коду страны также предшествует знак подчеркивания, когда он связывается с именем связки ресурсов. Связка ресурсов, имеющая только имя семейства, является применяемой по умолчанию. Она используется, когда недоступны специфичные для языка связки.

На заметку! *Коды языков определены стандартом ISO 639, а коды стран — стандартом ISO 3166.*

Методы, определенные в ResourceBundle, перечислены в табл. 18.19. Одно важное замечание: null-ключи не допускаются, а потому несколько методов возбуждают исключение NullPointerException, если получают null в качестве ключа. Следует обратить внимание на вложенный класс ResourceBundle.Control. Он добавлен в Java SE 6 и используется для управления процессом загрузки связок ресурсов.

У ResourceBundle есть два подкласса. Первый из них — PropertyResourceBundle, управляющий ресурсами через файлы свойств. PropertyResourceBundle не добавляет собственных методов.

Таблица 18.19. Методы, определенные в `ResourceBundle`

Метод	Описание
`static final void clearCache()`	Удаляет все связки ресурсов из кэша, которые были загружены загрузчиком классов по умолчанию. (Добавлен в Java SE 6.)
`static final void clearCache(ClassLoader ldr)`	Удаляет все связки ресурсов из кэша, которые были загружены загрузчиком *ldr*. (Добавлен в Java SE 6.)
`boolean containsKey(String k)`	Возвращает `true`, если *k* — ключ внутри вызывающей связки ресурсов (или ее родителя). (Добавлен в Java SE 6.)
`static final ResourceBundle getBundle(String familyName)`	Загружает связку ресурсов с именем семейства *familyName*, используя локаль и загрузчик классов по умолчанию. Возбуждает исключение `MissingResourceException`, если не найдено связки ресурсов, соответствующей имени *familyName*.
`static final ResourceBundle getBundle(String familyName, Locale loc)`	Загружает связку ресурсов с именем семейства *familyName*, используя указанную локаль и загрузчик классов по умолчанию. Возбуждает исключение `MissingResourceException`, если не найдено связки ресурсов, соответствующей имени *familyName*.
`static ResourceBundle getBundle(String familyName, Locale loc, ClassLoader ldr)`	Загружает связку ресурсов с именем семейства *familyName*, используя указанную локаль и указанный загрузчик классов. Возбуждает исключение `MissingResourceException`, если не найдено связки ресурсов, соответствующей имени *familyName*.
`static final ResourceBundle getBundle(String familyName, ResourceBundle.Control cntl)`	Загружает связку ресурсов с именем семейства *familyName*, используя локаль и загрузчик классов по умолчанию. Процесс загрузки находится под управлением *cntl*. Возбуждает исключение `MissingResourceException`, если не найдено связки ресурсов, соответствующей имени *familyName*. (Добавлен в Java SE 6.)
`static final ResourceBundle getBundle(String familyName, Locale loc, ResourceBundle.Control cntl)`	Загружает связку ресурсов с именем семейства *familyName*, используя указанную локаль и загрузчик классов по умолчанию. Процесс загрузки находится под управлением *cntl*. Возбуждает исключение `MissingResourceException`, если не найдено связки ресурсов, соответствующей имени *familyName*. (Добавлен в Java SE 6.)
`static ResourceBundle getBundle(String familyName, Locale loc, ClassLoader ldr, ResourceBundle.Control cntl)`	Загружает связку ресурсов с именем семейства *familyName*, используя указанную локаль и указанный загрузчик классов. Процесс загрузки находится под управлением *cntl*. Возбуждает исключение `MissingResourceException`, если не найдено связки ресурсов, соответствующей имени *familyName*. (Добавлен в Java SE 6.)
`abstract Enumeration<String> getKeys()`	Возвращает ключи связки ресурсов как перечисление строк. Любые родительские ключи также получаются.
`Locale getLocale()`	Возвращает локаль, поддерживаемую связкой ресурсов.
`final Object getObject(String k)`	Возвращает объект, ассоциированный с ключом, переданным в *k*. Возбуждает исключение `MissingResourceException`, если *k* не найден в связке ресурсов.

Окончание табл. 18.19

Метод	Описание
`final String getString(String k)`	Возвращает строку, ассоциированную с ключом, переданным в `k`. Возбуждает исключение `MissingResourceException`, если `k` не найден в связке ресурсов. Возбуждает исключение `ClassCastException`, если объект, ассоциированный с `k`, не является строкой.
`final String[] getStringArray(String k)`	Возвращает массив строк, ассоциированных с ключом, переданным в `k`. Возбуждает исключение `MissingResourceException`, если `k` не найден в связке ресурсов. Возбуждает исключение `ClassCastException`, если объект, ассоциированный с `k`, не является массивом строк.
`protected abstract Object handleGetObject(String k)`	Возвращает объект, ассоциированный с ключом, переданным в `k`. Возвращает `null`, если `k` не найден в связке ресурсов.
`protected Set<String> handleKeySet()`	Возвращает ключи связки ресурсов как набор. Родительские ключи не извлекаются. Также не получаются ключи со значениями `null`. (Добавлен в Java SE 6.)
`Set<String> keySet()`	Возвращает ключи связки ресурсов как набор строк. Родительские ключи также извлекаются. (Добавлен в Java SE 6.)
`protected void setParent(ResourceBundle parent)`	Устанавливает *`parent`* как родительскую связку для данной связки ресурсов. При поиске ключа, если он не найден в вызывающем ресурсном объекте, поиск продолжится в родительском.

Второй — это абстрактный класс `ListResourceBundle`, управляющий ресурсами в массиве пар "ключ-значение". `ListResourceBundle` добавляет метод `getContents()`, который должны реализовать все подклассы. Выглядит он так:

```
protected abstract Object[][] getContents()
```

Он возвращает двумерный массив, содержащий пары "ключ-значение", представляющие ресурсы. Ключи могут быть строками. Значения — обычно строки, но могут быть и объектами других типов.

Рассмотрим пример, демонстрирующий использование связки ресурсов. Связка ресурсов имеет имя семейства `SampleRB`. Два класса связок ресурсов этого семейства создаются расширением `ListResourceBundle`. Первый называется `SampleRB` и представляет собой связку по умолчанию для 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] = "StopText";
  resources[1][1] = "Stop";
  resources[2][0] = "StartText";
  resources[2][1] = "Start";
  return resources;
}
}
```

Вторая связка ресурсов, показанная ниже, называется `SampleRB_de`. Она содержит немецкий перевод.

```
import java.util.*;
// Немецкоязычная версия.
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;
}
}
```

Следующая программа демонстрирует эти две связки ресурсов, отображая строки, ассоциированные с каждым ключом как для версии по умолчанию (английской), так и для немецкой версии.

```
// Демонстрация связки ресурсов.
import java.util.*;

class LRBDemo {
public static void main(String args[]) {
  // Загрузить связку по умолчанию.
  ResourceBundle rd = ResourceBundle.getBundle("SampleRB");
  System.out.println("Англоязычная версия: ");
  System.out.println("Строка для ключа Title: " + rd.getString("title"));
  System.out.println("Строка для ключа StopText: " + rd.getString("StopText"));
  System.out.println("Строка для ключа StartText: " +
           rd.getString("StartText"));
  // Загрузить немецкую связку.
  rd = ResourceBundle.getBundle("SampleRB", Locale.GERMAN);
  System.out.println("\nНемецкоязычная версия: ");
  System.out.println("Строка для ключа Title: " + rd.getString("title"));
  System.out.println("Строка для ключа StopText: " + rd.getString("StopText"));
  System.out.println("Строка для ключа StartText: " +
           rd.getString("StartText"));
}
}
```

Вывод этой программы будет таким:

```
Англоязычная версия:
Строка для ключа Title: My Program
Строка для ключа StopText: Stop
Строка для ключа StartText: Start

Немецкоязычная версия:
Строка для ключа Title: Mein Programm
Строка для ключа StopText: Anschlag
Строка для ключа StartText: Anfang
```

Прочие служебные классы и интерфейсы

В дополнение к уже описанным классам java.util содержит классы, перечисленные в табл. 18.20.

Таблица 18.20. Дополнительные классы java.util

Класс	Описание
EventListenerProxy	Расширяет класс EventListener для принятия дополнительных параметров. См. в главе 22 дискуссию о слушателях событий.
EventObject	Суперкласс для всех классов событий. События обсуждаются в главе 22.
FormattableFlags	Определяет флаги форматирования, используемые в интерфейсе Formattable.
PropertyPermission	Управляет правами доступа к свойствам.
ServiceLoader	Обеспечивает средства нахождения поставщиков служб.
UUID	Инкапсулирует и управляет универсальными уникальными идентификаторами (Universally Unique Identifier — UUID).

Интерфейсы, описанные в табл. 18.21, также входят в состав пакета java.util.

Таблица 18.21. Дополнительные интерфейсы java.util

Интерфейс	Описание
EventListener	Указывает, что класс является слушателем событий. События обсуждаются в главе 22.
Formattable	Описывает класс, обеспечивающий специальное (настраиваемое) форматирование.

Вложенные пакеты java.util

Java определяет следующие вложенные пакеты java.util:

- java.util.concurrent
- java.util.concurrent.atomic
- java.util.concurrent.locks
- java.util.jar
- java.util.logging
- java.util.prefs
- java.util.regex
- java.util.spi
- java.util.zip

Ниже все они кратко описаны.

java.util.concurrent, java.util.concurrent.atomic, java.util.concurrent.locks

Пакет `java.util.concurrent` наряду с его двумя подпакетами, `java.util.concurrent.atomic` и `java.util.concurrent.locks`, предназначен для поддержки параллельного программирования. Эти пакеты предлагают высокопроизводительную альтернативу применению встроенных в Java средств синхронизации, когда требуются безопасные в отношении потоков операции. Эти пакеты подробно рассматриваются в главе 26.

java.util.jar

Пакет `java.util.jar` предлагает возможность чтения и записи архивных файлов Java Archive (JAR).

java.util.logging

Пакет `java.util.logging` обеспечивает поддержку журналов активности программ, которые могут быть использованы для записи действий программ, а также для поиска проблем и отладки.

java.util.prefs

Пакет `java.util.prefs` обеспечивает поддержку пользовательских предпочтений. Обычно применяется для поддержки конфигураций программ.

java.util.regex

Пакет `java.util.regex` обеспечивает поддержку работы с регулярными выражениями. Он подробно описан в главе 27.

java.util.spi

Пакет `java.util.spi` обеспечивает поддержку поставщиков служб. (Добавлен в Java SE 6.)

java.util.zip

Пакет `java.util.zip` обеспечивает возможность чтения и записи файлов архивов в популярных форматах ZIP и GZIP. Доступны также входные и выходные потоки ZIP и GZIP.

19
ГЛАВА

Ввод-вывод: пакет `java.io`

Эта глава посвящена `java.io` — пакету, поддерживающему операции ввода-вывода. В главе 13 был представлен обзор системы ввода-вывода Java. Здесь же мы рассмотрим систему ввода-вывода Java более подробно.

Как известно всем программистам с давних времен, большинство программ не могут выполнять свою работу, не имея доступа к внешним данным. Данные извлекаются из источника *ввода*. Результат программы направляется в *вывод*. На языке Java эти понятия определяются очень широко. Например, источником ввода или местом вывода может служить сетевое соединение, буфер в памяти или дисковый файл — всеми ими можно манипулировать посредством классов ввода-вывода Java. Хотя физически они совершенно различны, все эти устройства описываются единой абстракцией — *потоком*. Поток, как уже объяснялось в главе 13 — это логическая сущность, которая выдает или получает информацию. Поток присоединен к физическому устройству посредством системы ввода-вывода Java. Все потоки ведут себя похоже, даже несмотря на то, что физические устройства, к которым они присоединены, в корне отличаются.

На заметку! *В дополнение к средствам ввода-вывода, описанным здесь, Java предлагает дополнительную поддержку ввода-вывода в пакете* `java.nio`*, о котором пойдет речь в главе 27.*

Классы и интерфейсы ввода-вывода Java

Классы ввода-вывода, определенные в `java.io`, перечислены ниже:

```
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	

Класс `Console` был добавлен в Java SE 6.

Пакет `java.io` также содержит два устаревших (deprecated) класса, которые не показаны в приведенном выше перечне, а именно — `LineNumberInputStream` и `StringBufferInputStream`. Эти классы не должны использоваться в новом коде.

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` присутствует множество классов и интерфейсов. Среди них — байтовые и символьные потоки, сериализация объектов (их сохранение и восстановление). В настоящей главе рассматривается несколько наиболее широко используемых компонентов ввода-вывода. Новый класс `Console` также описан. Начнем обсуждение с одного из наиболее отличающихся классов ввода-вывода — `File`.

File

Хотя большинство классов, определенных в `java.io`, работают с потоками, класс `File` этого не делает. Он имеет дело непосредственно с файлами и файловой системой. То есть класс `File` не указывает, как извлекается и сохраняется информация в файлах; он описывает свойства самих файлов. Объект `File` служит для получения и манипулирования информацией, ассоциированной с дисковым файлом, такой как права доступа, время, дата и путь к каталогу, а также для навигации иерархиями подкаталогов.

`File` — первичный источник и место назначения для данных во многих программах. Хотя существует несколько ограничений в части использования файлов в аплетах (из соображений безопасности), тем не менее, они продолжают оставаться центральным ресурсом для хранения постоянной и разделяемой информации. Каталог в Java трактуется как объект `File` с единственным дополнительным свойством: списком имен файлов, которые могут быть получены методом `list()`.

Для создания объектов типа `File` могут быть использованы следующие конструкторы:

```
File(String directoryPath)
File(String directoryPath, String filename)
File(File dirObj, String filename)
File(URI uriObj)
```

Здесь *`directoryPath`* — имя пути файла, *`filename`* — имя файла или подкаталога, *`dirObj`* — объект `File`, указывающий каталог, а *`uriObj`* — объект `URI`, описывающий файл.

В следующем примере создаются три файла: `f1`, `f2` и `f3`. Первый объект `File` конструируется с путем к каталогу в единственном аргументе. Второй включает два аргумен-

та — путь и имя файла. Третий включает путь, присвоенный f1, и имя файла; f3 ссылается на тот же файл, что и f2.

```
File f1 = new File("/");
File f2 = new File("/","autoexec.bat");
File f3 = new File(f1,"autoexec.bat");
```

На заметку! *Java корректно обращается с разделителями пути, которые отличаются между UNIX и Windows. Если вы используете прямой слэш (/) в Windows-версии Java, то путь будет все равно будет сформирован корректно. Помните, однако, что для использования символа обратного слэша (\) в Windows вы должны применять в строках управляющую последовательность (\\).*

File определяет множество методов, представляющих стандартные свойства объекта File. Например, getName() возвращает имя файла, getParent() — имя родительского каталога, а exists() возвращает true, если файл существует, и false — если нет. Однако класс File не симметричен. То есть в нем есть несколько методов, позволяющих *проверять* свойства простого файлового объекта, у которых нет дополняющих их методов изменения этих атрибутов. В следующем примере демонстрируется применение нескольких методов File.

```
// Демонстрация работы с 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("Время модификации: " + f1.lastModified());
  p("Размер: " + f1.length() + " байт");
}
}
```

Запустив эту программу, вы увидите нечто вроде следующего:

```
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
является абсолютным
```

```
Время модификации: 812465204000
Размер: 695 байт
```

Большинство методов `File` самоочевидно. `isFile()` и `isAbsolute()` — нет. `isFile()` возвращает `true`, если вызывается с файлом, и `false` — если с каталогом. Также `isFile()` возвращает `false` для некоторых специальных файлов, таких как драйверы устройств и именованные каналы, поэтому этот метод может применяться для того, чтобы убедиться, что данный файл действительно ведет себя как файл. Метод `isAbsolute()` возвращает `true`, если файл имеет абсолютный путь, и `false` — если относительный.

Также `File` включает два полезных служебных метода. Первый из них, `renameTo()`, показан ниже:

```
boolean renameTo(File newName)
```

Здесь имя файла, указанное в параметре *newName*, становится новым именем вызывающего объекта `File`. Метод возвращает `true` в случае успеха и `false` — в случае неудачи, если файл не может быть переименован (если вы либо пытаетесь переименовать файл так, например, что в результате он должен переместиться из одного каталога в другой, или же указываете имя существующего файла).

Второй служебный метод — `delete()`, который удаляет дисковый файл, представленный путем вызывающего объекта `File`. Выглядит он так:

```
boolean delete()
```

Также вы можете использовать метод `delete()` для удаления каталога, если он пуст. `delete()` возвращает `true`, если ему удается удалить файл, и `false`, если файл не может быть удален. В табл. 19.1 приведено еще несколько методов класса `File`, которые вы наверняка сочтете полезными.

Таблица 19.1. Полезные методы класса `File`

Метод	Описание
`void deleteOnExit()`	Удаляет файл, ассоциированный с вызывающим объектом по завершении работы виртуальной машины Java.
`long getFreeSpace()`	Возвращает количество свободных байт хранилища, доступных в разделе, ассоциированном с вызывающим объектом. (Добавлен в Java SE 6.)
`long getTotalSpace()`	Возвращает емкость хранилища раздела, ассоциированного с вызывающим объектом. (Добавлен в Java SE 6.)
`long getUsableSpace()`	Возвращает количество доступных годных к употреблению свободных байт, находящихся на разделе, ассоциированном с вызывающим объектом. (Добавлен в Java SE 6.)
`boolean isHidden()`	Возвращает `true`, если вызывающий файл является скрытым, и `false` — в противном случае.
`boolean setLastModified(long millisec)`	Устанавливает временную метку для вызываемого файла в значение `millisec`, которое представляет количество миллисекунд, прошедших с 1 января 1970 г. по UTC.
`boolean setReadOnly()`	Делает вызывающий файл доступным только для чтения.

Также существуют методы для пометки файлов как доступных только для чтения, записи или выполнения. Поскольку File реализует интерфейс Comparable, также поддерживается метод compareTo().

Каталоги

Каталог — это объект File, содержащий список других файлов и каталогов. После создания объекта File, являющегося каталогом, его метод isDirectory() вернет true. В этом случае вы можете вызвать на этом объекте метод list(), чтобы извлечь список других файлов и каталогов, находящихся внутри него. Упомянутый метод имеет две формы. Вот первая из них:

```
String[] list()
```

Список файлов возвращается в виде массива объектов String. Приведенная ниже программа иллюстрирует использование list() для просмотра содержимого каталога.

```
// Использование каталогов.
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 + " is not a directory");
  }
}
}
```

Ниже приведен вывод этой программы (Конечно, вывод, который вы увидите, будет отличаться, — 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
```

Использование FilenameFilter

Часто вам понадобится ограничить количество файлов, возвращенных методом list(), для включения только тех файлов, которые соответствуют определенному шаблону имен, или *фильтру*. Чтобы сделать это, вы должны использовать вторую форму list():

```
String[] list(FilenameFilter FFObj)
```

В этой форме *FFObj* — объект класса, реализующего интерфейс FilenameFilter.

FilenameFilter определяет единственный метод accept(), вызываемый по одному разу с каждым файлом в списке. Его общая форма такова:

```
boolean accept(File directory, String filename)
```

Метод accept() возвращает true для файлов каталога, указанного в directory, которые должны быть включены в список (то есть те, что соответствуют аргументу filename), и возвращает false для файлов, которые следует из списка исключить.

Класс OnlyExt, показанный ниже, реализует FilenameFilter. Он будет использован для модификации предыдущей программы так, чтобы ограничить видимость имен файлов, возвращенных list(), только теми из них, которые оканчиваются расширением, указанным при конструировании этого объекта.

```
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);
   }
}
```

Модифицированная программа просмотра листинга каталога показана ниже. Теперь она выведет только файлы с расширением .html.

```
// Каталог файлов .HTML.
import java.io.*;
class DirListOnly {
public static void main(String args[]) {
  String dirname = "/java";
  File f1 = new File(dirname);
  FilenameFilter only = new OnlyExt("html");
  String s[] = f1.list(only);
  for (int i=0; i < s.length; i++) {
     System.out.println(s[i]);
  }
}
}
```

Альтернатива — listFiles()

Существует вариация метода list(), именуемая listFiles(), которую вы можете счесть удобной. Сигнатуры listFiles() показаны ниже:

```
File[ ] listFiles( )
File[ ] listFiles(FilenameFilter FFObj)
File[ ] listFiles(FileFilter FObj)
```

Эти методы возвращают список файлов в виде массива объектов File вместо строк. Первый метод возвращает все файлы, второй — только те, что удовлетворяют указанному FilenameFilter. Помимо возвращения массива объектов File, эти две версии listFiles() работают точно так же, как и методы list().

Третья версия listFiles() возвращает те файлы, у которых путевые имена отвечают указанному FileFilter. FileFilter определяет единственный метод accept(), который вызывается один раз для каждого файла в списке. Его общая форма такова:

```
boolean accept(File path)
```

Метод accept() возвращает true для файлов, которые должны быть включены в список (то есть те, что соответствуют аргументу *path*), и false — для тех, которые следует исключить.

Создание каталогов

Другими двумя полезными служебными методами File являются mkdir() и mkdirs(). Метод mkdir() создает каталог, возвращая true в случае успеха и false — в случае неудачи. Неудача говорит о том, что путь, указанный в объекте File, уже существует, или что каталог не может быть создан по причине того, что полный путь к нему еще не существует. Чтобы создать каталог, для которого путь еще не создан, используйте метод mkdirs(). Он создаст как сам каталог, так и всех его родителей.

Интерфейсы Closeable и Flushable

В последнее время (с появлением версии JDK 5), к java.io были добавлены еще два интерфейса: Closeable и Flushable. Эти интерфейсы реализуются несколькими классами ввода-вывода. Их включение не добавило новой функциональности потоковым классам. Они просто предоставили унифицированный способ спецификации того, что поток может быть закрыт или сброшен.

Объекты класса, реализующего Closeable, могут быть закрыты. Для этого они определяют метод close() следующего вида:

```
void close() throws IOException
```

Этот метод закрывает вызывающий поток, освобождая все ресурсы, которые мог удерживать. Этот интерфейс реализован всеми классами ввода-вывода, которые открывают поток, впоследствии подлежащий закрытию.

Объекты класса, реализующего интерфейс Flushable, могут заставить буферизованный вывод записываться в поток, к которому присоединен данный объект. Он определяет метод flush(), показанный ниже:

```
void flush() throws IOException
```

Сброс потока обычно вынуждает буферизованный вывод физически записываться на лежащем в основе устройстве. Этот интерфейс реализован всеми классами ввода-вывода, способными выполнять запись в поток.

Класс Stream

Основанный на потоках ввод-вывод Java построен на основе абстрактных классов: InputStream, OutputStream, Reader и Writer. Эти классы уже были кратко упомянуты в главе 13. Они используются для создания некоторых конкретных подклассов потоков.

Хотя ваши программы реализуют свои операции ввода-вывода через конкретные подклассы, классы верхнего уровня определяют базовую функциональность, общую для всех потоковых классов.

`InputStream` и `OutputStream` предназначены для байтовых потоков, а `Reader` и `Writer` — для символьных. Классы байтовых потоков и классы символьных потоков формируют отдельные иерархии. В целом вы должны использовать классы символьных потоков, имея дело с символами строк, а классы байтовых потоков — работая с байтами или другими двоичными объектами.

В оставшейся части этой главы мы будем рассматривать как байт-ориентированные, так и символ-ориентированные потоки.

Байтовые потоки

Классы байтовых потоков предоставляют богатую среду для обработки байт-ориентированного ввода-вывода. Байтовый поток может быть использован с объектами любого типа, включая двоичные данные. Такая многосторонность делает байтовые потоки важными для многих типов программ. Поскольку классы байтовых потоков берут свое начало с `InputStream` и `OutputStream`, с них и начнем обсуждение.

InputStream

`InputStream` — абстрактный класс, определяющий Java-модель байтового потокового ввода. Он реализует интерфейс `Closeable`. Большинство методов этого класса в случае возникновения ошибочных ситуаций возбуждает исключение `IOException`. (Сюда не входят `mark()` и `markSupported()`.) В табл. 19.2 перечислены методы `InputStream`.

Таблица 19.2. Методы, определенные в `InputStream`

Метод	Описание
`int available()`	Возвращает количество байт ввода, которые доступны в данный момент для чтения.
`void close()`	Закрывает источник ввода. Последующие попытки чтения генерируют `IOException`.
`void mark(int numBytes)`	Помещает метку в текущую точку входного потока, которая остается корректной до тех пор, пока не будет прочитано *numBytes* байт.
`boolean markSupported()`	Возвращает `true`, если `mark()`/`reset()` поддерживаются вызывающим потоком.
`int read()`	Возвращает целочисленное представление следующего доступного байта в потоке. При достижении конца файла возвращается `-1`.
`int read(byte buffer[])`	Пытается читать до *numBytes* в *buffer*, возвращая количество успешно прочитанных байт. По достижении конца файла возвращает `-1`.
`int read(byte buffer[], int offset, int numBytes)`	Пытается читать до *numBytes* в *buffer*, начиная с *buffer*`[`*offset*`]`, возвращая количество успешно прочитанных байт. По достижении конца файла возвращает `-1`.
`void reset()`	Сбрасывает входной указатель в ранее установленную метку.
`long skip(long numBytes)`	Игнорирует (то есть пропускает) *numBytes* байт ввода, возвращая количество в действительности проигнорированных байт.

OutputStream

OutputStream — абстрактный класс, определяющий потоковый байтовый вывод. Реализует интерфейсы Closeable и Flushable. Большинство методов этого класса возвращают void и возбуждают IOException в случае ошибок. (Последнее не относится к mark() и markSupported().) В табл. 19.3 перечислены методы OutputStream.

На заметку! *Большинство методов, описанных в таблицах 19.2 и 19.3, реализованы подклассами InputStream и OutputStream. Методы mark() и reset() являются исключениями; имейте это в виду, когда ниже речь пойдет о каждом из подклассов.*

Таблица 19.3. Методы, определенные в OutputStream

Метод	Описание
`int close()`	Закрывает выходной поток. Последующие попытки записи сгенерируют IOException.
`void flush()`	Финализирует выходное состояние, очищая все буферы. То есть очищает буферы вывода.
`void write(int b)`	Записывает единственный байт в выходной поток. Обратите внимание, что параметр имеет тип int, а это позволяет вызывать write() с выражениями без необходимости приведения их обратно к byte.
`void write(byte buffer[])`	Записывает полный массив байт в выходной поток.
`void write(byte buffer[], int numBytes)`	Записывает поддиапазон из numBytes байт из массива *buffer*, начиная с *buffer*[*offset*].

FileInputStream

Класс FileInputStream создает InputStream, который вы можете использовать для чтения байт из файла. Так выглядят два его наиболее часто используемые конструктора:

```
FileInputStream(String filepath)
FileInputStream(File fileObj)
```

Каждый из них может возбудить исключение FileNotFoundException. Здесь *filepath* — полное путевое имя файла, а *fileObj* — объект File, описывающий файл.

В следующем примере создается два FileInputStream, использующих один и тот же дисковый файл и оба эти конструктора:

```
FileInputStream f0 = new FileInputStream("/autoexec.bat")
File f = new File("/autoexec.bat");
FileInputStream f1 = new FileInputStream(f);
```

Хотя первый конструктор, вероятно, используется чаще, второй позволяет внимательно исследовать файл с помощью методов класса File, прежде чем присоединять его к входному потоку. При создании FileInputStream он также открывается для чтения. FileInputStream переопределяет шесть своих методов абстрактного класса InputStream. Методы mark() и reset() не переопределяются, и все попытки использовать reset() с FileInputStream приводят к возникновению IOException.

Следующий пример показывает, как прочесть один байт, массив байтов и поддиапазон массива байт. Также он иллюстрирует, как использовать available() для определе-

ния оставшегося числа байт, а также метод `skip()` — для пропуска нежелательных байт. Программа читает свой собственный исходный файл, который должен присутствовать в текущем каталоге.

```
// Demonstrate FileInputStream.
import java.io.*;
class FileInputStreamDemo {
public static void main(String args[]) throws IOException {
  int size;
  InputStream f =
     new FileInputStream("FileInputStreamDemo.java");
  System.out.println("Total Available Bytes: " + (size = f.available()));
  int n = size/40;
  System.out.println("First " + n + " bytes of the file one read() at a time");
  for (int i=0; i < n; i++) {
     System.out.print((char) f.read());
  }
  System.out.println("\nStill Available: " + f.available());
  System.out.println("Reading the next " + n + " with one read(b[])");
  byte b[] = new byte[n];
  if (f.read(b) != n) {
     System.err.println("couldn't read " + n + " bytes.");
  }
  System.out.println(new String(b, 0, n));
  System.out.println("\nStill Available: " + (size = f.available()));
  System.out.println("Skipping half of remaining bytes with skip()");
  f.skip(size/2);
  System.out.println("Still Available: " + f.available());
  System.out.println("Reading " + n/2 + " into the end of array");
  if (f.read(b, n/2, n/2) != n/2) {
     System.err.println("couldn't read " + n/2 + " bytes.");
  }
  System.out.println(new String(b, 0, b.length));
  System.out.println("\nStill Available: " + f.available());
  f.close();
}
}
```

Так выглядит вывод этой программы:

```
Total Available Bytes: 1433
First 35 bytes of the file one read() at a time
// Demonstrate FileInputStream.
im
Still Available: 1398
Reading the next 35 with one read(b[])
port java.io.*;
class FileInputS
Still Available: 1363
Skipping half of remaining bytes with skip()
Still Available: 682
Reading 17 into the end of array
port java.io.*;
read(b) != n) {
S
Still Available: 665
```

Этот несколько надуманный пример демонстрирует чтение тремя способами, пропуск ввода и проверку количества доступных данных в потоке.

На заметку! *Приведенный пример (как и прочие примеры этой главы) обрабатывает все исключения ввода-вывода, которые могут возникнуть в виде перемещения* `IOException` *за пределы* `main()`*, что означает, что они обрабатываются JVM. Этого достаточно для демонстрационной программы (и небольших служебных программ, которые вы пишете для собственного пользования), но коммерческие приложения обычно требуют обработки исключений ввода-вывода внутри программы.*

FileOutputStream

`FileOutputStream` создает `OutputStream`, который вы можете использовать для записи байт в файл. Вот его наиболее часто используемые конструкторы:

```
FileOutputStream(String filePath)
FileOutputStream(File fileObj)
FileOutputStream(String filePath, boolean append)
FileOutputStream(File fileObj, boolean append)
```

Они могут возбуждать исключение `FileNotFoundException`. Здесь *`filePath`* — полное путевое имя файла, а *`fileObj`* — объект `File`, описывающий файл. Если параметр *`append`* равен `true`, файл открывается в режиме добавления.

Создание `FileOutputStream` не зависит от того, существует ли указанный файл. `FileOutputStream` создает его перед открытием, когда вы создаете объект. В случае попытки открытия файла, доступного только для чтения, будет возбуждено исключение `IOException`.

В следующем примере создается буфер байт, сначала создавая `String`, а затем используя метод `getBytes()` для извлечения ее эквивалента в виде байтового массива. Затем создаются три файла. Первый — `file1.txt`, будет содержать каждый второй байт примера. Второй — `file2.txt`, будет содержать полный набор байт. Третий и последний — `file3.txt`, будет содержать только последнюю четверть.

```
// Демонстрация применения FileOutputStream.
import java.io.*;
class FileOutputStreamDemo {
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.";
  byte buf[] = source.getBytes();
  OutputStream f0 = new FileOutputStream("file1.txt");
  for (int i=0; i < buf.length; i += 2) {
     f0.write(buf[i]);
  }
  f0.close();
  OutputStream f1 = new FileOutputStream("file2.txt");
  f1.write(buf);
  f1.close();
  OutputStream f2 = new FileOutputStream("file3.txt");
  f2.write(buf,buf.length-buf.length/4,buf.length/4);
  f2.close();
}
}
```

Так будет выглядеть содержимое каждого файла после выполнения этой программы. Сначала file1.txt:

```
Nwi h iefralgo e
t oet h i ftercuty n a hi u ae.
```

Затем 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, file3.txt:

```
nd pay their due taxes.
```

ByteArrayInputStream

ByteArrayInputStream — реализация входного потока, использующего байтовый массив в качестве источника данных. Этот класс имеет два конструктора, каждый из которых требует байтового массива в качестве источника данных:

```
ByteArrayInputStream(byte array[ ])
ByteArrayInputStream(byte array[ ], int start, int numBytes)
```

Здесь *array* — источник данных. Второй конструктор создает InputStream из подмножества вашего байтового массива, который начинается с символа в позиции, указанной в *start*, и длиной, равной *numBytes*.

В следующем примере создается пара ByteArrayInputStreams, которая инициализируется байтами, представляющими английский алфавит.

```
// Демонстрация применения 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 input2 — только первые три буквы.

ByteArrayInputStream реализует и mark(), и reset(). Однако если mark() не вызывается, то reset() устанавливает указатель потока в его начало — в данном случае в начало байтового массива, переданного конструктору. Следующий пример показывает, как использовать метод reset() для чтения одного и того же ввода дважды. В этом случае мы читаем и печатаем буквы “abc” сначала в нижнем регистре, а затем в верхнем.

```
import java.io.*;
class ByteArrayInputStreamReset {
public static void main(String args[]) throws IOException {
  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();
  }
}
}
```

Код в этом примере сначала читает каждый символ потока и печатает его, как он есть — в нижнем регистре. Затем он сбрасывает поток и начинает чтение заново, на этот раз перед печатью преобразуя каждый символ в верхний регистр. Вывод получается таким:

```
abc
ABC
```

ByteArrayOutputStream

`ByteArrayOutputStream` — реализация потока вывода, использующего байтовый массив в качестве места назначения. `ByteArrayOutputStream` имеет два конструктора, показанных ниже:

```
ByteArrayOutputStream( )
ByteArrayOutputStream(int numBytes)
```

В первой форме создается буфер в 32 байта размером. Во втором создается буфер указанного в параметре *numBytes* размера. Буфер хранится в защищенном поле `buf` класса `ByteArrayOutputStream`. Размер буфера увеличивается автоматически по мере необходимости. Количество байт, содержащееся в буфере, хранится в защищенном поле `count` класса `ByteArrayOutputStream`.

В следующем примере демонстрируется использование `ByteArrayOutputStream`.

```
// Демонстрация применения ByteArrayOutputStream.
import java.io.*;
class ByteArrayOutputStreamDemo {
public static void main(String args[]) throws IOException {
   ByteArrayOutputStream f = new ByteArrayOutputStream();
   String s = "This should end up in the array";
   byte buf[] = s.getBytes();
   f.write(buf);
   System.out.println("Buffer as a string");
   System.out.println(f.toString());
   System.out.println("Into array");
   byte b[] = f.toByteArray();
   for (int i=0; i<b.length; i++) {
      System.out.print((char) b[i]);
   }
   System.out.println("\nTo an OutputStream()");
   OutputStream f2 = new FileOutputStream("test.txt");
   f.writeTo(f2);
```

```
    f2.close();
    System.out.println("Doing a reset");
    f.reset();
    for (int i=0; i<3; i++)
       f.write('X');
    System.out.println(f.toString());
}
}
```

Запустив эту программу, вы получите следующий вывод. Обратите внимание, что после вызова `reset()`, выводится в начале три буквы 'X'.

```
Buffer as a string
This should end up in the array
Into array
This should end up in the array
To an OutputStream()
Doing a reset
XXX
```

В этом примере для записи содержимого `test.txt` используется удобный метод `writeTo()`. Просмотр `test.txt`, созданного в предыдущем примере, покажет результат, который следовало ожидать:

```
This should end up in the array
```

Фильтруемые потоки байтов

Фильтруемые потоки байтов — это просто оболочки вокруг входных или выходных потоков, которые прозрачно предоставляют некоторый дополнительный уровень функциональности. Эти потоки обычно доступны методам, которые ожидают обобщенного потока, являющегося суперклассом фильтрованного потока. Типичными расширениями являются буферизация, преобразование символов и raw-данных. Фильтруемые потоки байтов — это `FilterInputStream` и `FilterOutputStream`. Их конструкторы показаны ниже.

```
FilterOutputStream(OutputStream os)
FilterInputStream(InputStream is)
```

Методы, представленные в этих классах, идентичны `InputStream` и `OutputStream`.

Буферизуемые потоки байтов

Для байт-ориентированных потоков буферизованные потоки расширяют класс фильтруемого потока, добавляя к нему буфер в памяти. Этот буфер позволяет Java выполнять операции ввода-вывода более чем по одному байту за раз, тем самым повышая производительность. Благодаря доступности буфера, возможны пропуск, маркировка и сброс потока. Классы буферизованных байтовых потоков — это `BufferedInputStream` и `BufferedOutputStream`. Класс `PushbackInputStream` также реализует буферизованный поток.

BufferedInputStream

Буферизация ввода-вывода — очень распространенный способ оптимизации производительности. Класс `BufferedInputStream` позволяет поместить в оболочку любой поток `InputStream` и достичь увеличения производительности.

`BufferedInputStream` имеет два конструктора:
`BufferedInputStream(InputStream` *`inputStream`*`)`
`BufferedInputStream(InputStream` *`inputStream`*`, int` *`bufSize`*`)`

Первая форма создает буферизованный поток, использующий размер буфера по умолчанию. Во втором размер буфера указывается в *`bufSize`*. Рекомендуется использовать размеры буфера, кратные размеру страницы памяти, дисковому блоку и тому подобному — это окажет существенное положительное влияние на производительность. С другой стороны, однако, это зависит от реализации. Необязательный размер буфера обычно зависит от принимающей операционной системы, объема доступной памяти и конфигурации машины. Чтобы добиться эффективного использования буферизации, не обязательно погружаться во все эти сложности. Хорошим предположением будет установить размер буфера для потока ввода-вывода в 8192 байта или даже меньше. Таким образом, низкоуровневая система сможет читать блоки данных с диска или из сети и сохранять результат в вашем буфере. То есть, даже если вы читаете данные по одному байту из `InputStream`, то большую часть времени будете иметь дело с быстрой памятью.

Следующий пример моделирует ситуацию, в которой мы можем использовать `mark()` для запоминания места во входном потоке, чтобы позднее вернуться к нему методом `reset()`. Этот пример разбирает поток, находя в нем HTML-конструкцию, указывающую на символ авторских прав. Такая ссылка начинается с амперсанда (`&`), заканчивается точкой с запятой (`;`) и не содержит какие-либо внутренние пробелы. Простой ввод содержит два амперсанда, чтобы показать случай, когда `reset()` срабатывает, а когда — нет.

```
// Использование буферизованного ввода.
import java.io.*;
class BufferedInputStreamDemo {
public static void main(String args[]) throws IOException {
  String s = "This is a &copy; copyright symbol " +
     "but this is &copy not.\n";
  byte buf[] = s.getBytes();
  ByteArrayInputStream in = new ByteArrayInputStream(buf);
  BufferedInputStream f = new BufferedInputStream(in);
  int c;
  boolean marked = false;
  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;
      }
    }
  }
}
```

Обратите внимание, что этот пример использует `mark(32)`, что сохраняет метку для чтения следующих 32 байт (чего достаточно для любых ссылок на сущности). Вот как выглядит вывод, генерируемый этой программой:

```
This is a (c) copyright symbol but this is &copy not.
```

BufferedOutputStream

Класс `BufferedOutputStream` подобен любому `OutputStream`, за исключением дополнительного метода `flush()`, используемого для обеспечения физической записи буферизуемых данных на реальное выходное устройство. Поскольку назначение `BufferedOutputStream` — увеличивать производительность за счет сокращения количества физических записей данных, вам может понадобиться вызывать `flush()`, чтобы инициировать немедленную запись всех данных из буфера.

В отличие от буферизованного ввода буферизованный вывод не предоставляет дополнительной функциональности. Буферы вывода в Java нужны для повышения производительности. Вот два доступных конструктора этого класса:

```
BufferedOutputStream(OutputStream outputStream)
BufferedOutputStream(OutputStream outputStream, int bufSize)
```

Первая форма создает буферизованный поток, используя размер буфера по умолчанию. Во второй форме размер буфера передается в *bufSize*.

PushbackInputStream

Одним из новшеств в буферизации является реализация обратного “вталкивания” (pushback). *Вталкивание* используется с потоками ввода, чтобы позволить чтение байта с последующим его возвратом (то есть “втолкнуть”) в поток. Класс `PushbackInputStream` реализует эту идею. Он представляет механизм для того, чтобы “заглянуть” во входной поток и увидеть, что оттуда поступит в следующий раз, не извлекая информации оттуда.

`PushbackInputStream` имеет следующие конструкторы:

```
PushbackInputStream(InputStream inputStream)
PushbackInputStream(InputStream inputStream, int numBytes)
```

Первая форма создает объект потока, позволяющий вернуть один байт во входной поток. Вторая форма создает поток, оснащенный буфером "вталкивания" длиной *numBytes*. Это позволяет вернуть во входной поток множество байт.

Помимо знакомых уже методов InputStream, PushbackInputStream предлагает метод unread(), показанный ниже:

```
void unread(int ch)
void unread(byte buffer[ ])
void unread(byte buffer, int offset, int numChars)
```

Первая форма вталкивает обратно в поток младший байт *ch*. После этого он вновь будет следующим байтом, возвращаемым последующим вызовом read(). Вторая форма возвратит затем байты из *buffer*. Третья же форма вталкивает *numChars* байт, начиная с позиции *offset* в *buffer*. Исключение IOException возбуждается в случае попытки вернуть байт, когда буфер вталкивания переполнен.

Рассмотрим пример, демонстрирующий, как синтаксический анализатор языка программирования может использовать PushbackInputStream и unread(), чтобы справиться с различием между операциями сравнения (==) и присваивания (=).

```
// Демонстрация применения unread().
import java.io.*;
class PushbackInputStreamDemo {
public static void main(String args[]) throws IOException {
  String s = "if (a == 4) a = 0;\n";
  byte buf[] = s.getBytes();
  ByteArrayInputStream in = new ByteArrayInputStream(buf);
  PushbackInputStream f = new PushbackInputStream(in);
  int c;
  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;
     }
  }
}
}
```

Ниже показан вывод этого примера. Обратите внимание, что == заменяется на .eq., а = — на <-.

```
if (a .eq. 4) a<- 0;
```

Внимание! *PushbackInputStream имеет побочный эффект, отражающийся в том, что делает недействительными методы mark() и reset() потока InputStream, использованного для его создания. Применяйте markSupported(), чтобы проверить каждый поток на предмет возможности использования mark()/reset().*

SequenceInputStream

Класс SequenceInputStream позволяет соединять вместе несколько экземпляров InputStream. Конструирование SequenceInputStream отличается от любого другого InputStream. Конструктор SequenceInputStream принимает в качестве аргумента либо пару InputStream, либо Enumeration из InputStream.

```
// Демонстрация последовательного ввода.
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[])
     throws IOException {
  int c;
  Vector<String> files = new Vector<String>();
  files.addElement("/autoexec.bat");
  files.addElement("/config.sys");
  InputStreamEnumerator e = new InputStreamEnumerator(files);
  InputStream input = new SequenceInputStream(e);
  while ((c = input.read()) != -1) {
     System.out.print((char) c);
  }
  input.close();
}
}
```

Этот пример создает Vector и затем добавляет к нему два имени файла. Затем этот вектор с именами передается классу InputStreamEnumerator, предназначенному служить оболочкой вектора, в которой элементы возвращаются не в виде имен файлов, а в виде открытых FileInputStream, созданных по этим именам. SequenceInputStream открывает каждый файл по очереди, и таким образом, этот пример печатает содержимое этих двух файлов.

PrintStream

Класс PrintStream предоставляет все возможности вывода, которыми мы пользуемся с дескриптором файла System — System.out с самого начала нашей книги. Это делает PrintStream одним из наиболее часто используемых классов Java. Он реализует интерфейсы Appendable, Closeable и Flushable.

PrintStream определяет несколько конструкторов. Для начала рассмотрим перечисленные ниже:

```
PrintStream(OutputStream outputStream)
PrintStream(OutputStream outputStream, boolean flushOnNewline)
PrintStream(OutputStream outputStream, boolean flushOnNewline, String charSet)
```

Здесь *outputStream* указывает открытый OutputStream, который будет принимать вывод. Параметр *flushOnNewline* управляет тем, будет ли выходной буфер автоматически сбрасываться при каждой записи символа новой строки (\n), записи байтового массива либо вызове println(). Если *flushOnNewline* равен true, происходит автоматический сброс. Если же он равен false, сброс будет неавтоматическим. Первый конструктор не включает автоматический сброс. Вы можете специфицировать кодировку символов, передав ее имя в *charSet*.

Следующий набор конструкторов предоставляет простые способы конструирования PrintStream, который пишет свой вывод в файл:

```
PrintStream(File outputFile) throws FileNotFoundException
PrintStream(File outputFile, String charSet)
   throws FileNotFoundException, UnsupportedEncodingException
PrintStream(String outputFileName) throws FileNotFoundException
PrintStream(String outputFileName, String charSet)
   throws FileNotFoundException, UnsupportedEncodingException
```

Они позволяют создавать PrintStream на основе объекта File либо имени файла. В любом случае файл создается автоматически. Любой существующий файл с тем же именем уничтожается. Будучи созданным, объект PrintStream управляет всем выводом в указанный файл. Кодировку символов можно указать в параметре charSet.

PrintStream поддерживает методы print() и println() для всех типов, включая Object. Если аргумент не относится к примитивному типу, то методы PrintStream вызывают метод объекта toString() и затем отображают его результат.

Не так давно (с появлением версии JDK 5) к PrintStream был добавлен метод printf(). Он позволяет специфицировать точный формат вывода записываемых данных. Метод printf() использует класс Formatter (описанный в главе 18) для форматирования данных. Затем он печатает эти данные в вызывающий поток. Хотя форматирование может выполняться вручную прямым вызовом Formatter, все же printf() существенно упрощает процесс. Он является аналогом функции C/C++ printf(), облегчая преобразование существующего кода C/C++ в Java. Откровенно говоря, printf() — весьма полезное дополнение к Java API, поскольку значительно упрощает вывод форматированных данных на консоль. Метод printf() имеет следующие общие формы:

```
PrintStream printf(String fmtString, Object ... args)
PrintStream printf(Locale loc, String fmtString, Object ... args)
```

Первая версия записывает *args* в стандартный вывод в формате, указанном *fmtString*, используя локальные установки по умолчанию. Второй позволяет специфицировать локаль. Оба возвращают вызывающий PrintStream.

В общем случае printf() работает в манере, подобной методу format(), который определен в Formatter. Параметр *fmtString* состоит из элементов двух типов. Первый тип состоит из символов, которые просто копируются в выходной буфер. Второй тип содержит спецификаторы формата, определяющие способ отображения последующих аргументов — *args*. За полной информацией о форматированном выводе, включая описание спецификаторов формата, обращайтесь к описанию класса Formatter в главе 18.

Поскольку System.out имеет тип PrintStream, вы можете вызывать printf() с System.out. Поэтому printf() может служить в качестве замены println(), когда необходимо выдавать на консоль форматированный вывод. Например, в следующей программе printf() используется для вывода числовых значений в различных форматах. В прошлом такое форматирование требовало существенной работы. С появлением printf() оно значительно упростилось.

```
// Демонстрация применения printf().
class PrintfDemo {
public static void main(String args[]) {
  System.out.println("Ниже следуют некоторые числовые значения " +
       "в различных форматах.\n");
  System.out.printf("Различные целочисленные форматы: ");
  System.out.printf("%d %(d %+d %05d\n", 3, -3, 3, 3);
  System.out.println();
  System.out.printf("Формат с плавающей точкой по умолчанию: %f\n",
       1234567.123);
  System.out.printf("Плавающая точка с запятыми: %,f\n",
       1234567.123);
  System.out.printf("Отрицательная плавающая точка по умолчанию: %,f\n",
       -1234567.123);
  System.out.printf("Опция отрицательной плавающей точки: %,(f\n",
       -1234567.123);
  System.out.println();
  System.out.printf("Строка из положительных и отрицательных значений:\n");
  System.out.printf("% ,.2f\n% ,.2f\n",
}
}
```

Вывод этой программы:

```
Ниже следуют некоторые числовые значения в различных форматах.
Различные целочисленные форматы: 3 (3) +3 00003
Формат с плавающей точкой по умолчанию: 1234567.123000
Плавающая точка с запятыми: 1,234,567.123000
Отрицательная плавающая точка по умолчанию: -1,234,567.123000
Опция отрицательной плавающей точки: (1,234,567.123000)
Строка из положительных и отрицательных значений:
 1,234,567.12
-1,234,567.12
```

В PrintStream также определен метод format(). Вот его общие формы:

```
PrintStream format(String fmtString, Object ... args)
PrintStream format(Locale loc, String fmtString, Object ... args)
```

DataOutputStream и DataInputStream

DataOutputStream и DataInputStream позволяют писать или читать примитивные данные в поток и из него. Они реализуют интерфейсы DataOutput и DataInput соответственно. Эти интерфейсы определяют методы, преобразующие примитивные значения в форму последовательности байт. Такие потоки облегчают сохранение в файле двоичных данных, таких как целочисленные значения или значения с плавающей точкой. Рассмотрим здесь и то, и другое.

DataOutputStream расширяет FilerOutputStream, который, в свою очередь, расширяет OutputStream.

В DataOutputStream определен следующий конструктор:

```
DataOutputStream(OutputStream outputStream)
```

Здесь *outputStream* специфицирует выходной поток, в который будут записаны данные.

DataOutputStream поддерживает все методы, определенные его суперклассами. Однако он реализует методы, определенные интерфейсом DataOutput, которые и делают его интересным. DataOutput определяет методы, преобразующие значения примитивных типов в последовательности байтов и затем записывающие их в лежащий в основе поток. Вот образцы этих методов:

```
final void writeDouble(double value) throws IOException
final void writeBoolean(boolean value) throws IOException
final void writeInt(int value) throws IOException
```

Здесь *value* — значение, записываемое в поток.

DataInputStream — это дополнение DataOutputStream. Он расширяет FilterInputStream, который в свою очередь, расширяет InputStream. DataInputStream определяет только один следующий конструктор:

```
DataInputStream(InputStream inputStream)
```

Здесь *inputStream* специфицирует входной поток, откуда будут читаться данные.

Как и DataOutputStream, DataInputStream поддерживает все методы своих суперклассов, наряду с методами, определенными интерфейсом DataInput, что и делает его уникальным. Эти методы читают последовательность байтов и преобразуют их в значения примитивных типов. Ниже показаны образцы этих методов:

```
double readDouble( ) throws IOException
boolean readBoolean( ) throws IOException
int readInt( ) throws IOException
```

В следующей программе демонстрируется использование DataOutputStream и DataInputStream.

```
import java.io.*;
class DataIODemo {
public static void main(String args[])
    throws IOException {
  FileOutputStream fout = new FileOutputStream("Test.dat");
  DataOutputStream out = new DataOutputStream(fout);

  out.writeDouble(98.6);
  out.writeInt(1000);
  out.writeBoolean(true);

  out.close();

  FileInputStream fin = new FileInputStream("Test.dat");
  DataInputStream in = new DataInputStream(fin);

  double d = in.readDouble();
  int i = in.readInt();
  boolean b = in.readBoolean();

  System.out.println("Вот значения: " + d + " " + i + " " + b);

  in.close();
}
}
```

Вывод приведен ниже:

```
Вот значения: 98.6 1000 true
```

RandomAccessFile

Класс `RandomAccessFile` инкапсулирует файл произвольного доступа. Он не наследуется от `InputStream` или `OutputStream`. Вместо этого он реализует интерфейсы `DataInput` и `DataOutput`, которые определяют базовые методы ввода-вывода. Также он реализует интерфейс `Closeable`. `RandomAccessFile` отличает его поддержка запросов на позиционирование — то есть вы можете установить указатель файла в любое место в пределах этого файла. Этот класс включает следующие два конструктора:

```
RandomAccessFile(File fileObj, String access)
   throws FileNotFoundException
RandomAccessFile(String filename, String access)
   throws FileNotFoundException
```

В первой форме *`fileObj`* специфицирует открываемый файл как объект `File`. Во второй форме имя файла передается в *`filename`*. В обоих случаях *`access`* определяет тип доступа. Если он равен "r", то файл может быть прочитан, но не может быть записан. Если "rw", то файл открывается в режиме чтения-записи. Если же access равен "rws", то файл открывается для операций чтения-записи и каждое изменение данных файла или его метаданных немедленно записывается на физическое устройство. Метод `seek()`, показанный ниже, используется для установки текущей позиции указателя внутри файла:

```
void seek(long newPos) throws IOException
```

Здесь *`newPos`* указывает новую позицию в байтах файлового указателя от начала файла. После вызова `seek()` следующая операция чтения или записи выполняется в этой новой позиции.

`RandomAccessFile` реализует стандартные методы ввода и вывода, которые вы можете использовать для чтения и записи файлов произвольного доступа. Кроме того, он включает несколько дополнительных методов. Одним из них является `setLength()`. Его сигнатура такова:

```
void setLength(long len) throws IOException
```

Этот метод устанавливает длину вызывающего файла равной указанному значению *`len`*. Метод может использоваться для удлинения или укорачивания файла. Если файл удлиняется, его добавочная порция является неопределенной.

Символьные потоки

В то время как классы байтовых потоков предоставляют достаточную функциональность для выполнения операций ввода-вывода любого типа, они не могут работать напрямую с символами Unicode. Поскольку одной из главных целей Java является поддержка философии "написано однажды, выполняется везде", понадобилось включить поддержку прямого ввода-вывода для символов. В этом разделе мы обсудим несколько классов символьного ввода-вывода. Как уже объяснялось ранее, в вершине иерархии символьных потоков находятся абстрактные классы `Reader` и `Writer`. С них и начнем.

На заметку! *Как было сказано в главе 13, классы символьного ввода-вывода были добавлены в версии Java 1.1. По этой причине вы все еще можете встретить унаследованный код, использующий байтовые потоки там, где более целесообразно было бы применить потоки символьные. Работая с таким кодом, будет неплохой идеей обновить его.*

Reader

`Reader` — абстрактный класс, определяющий символьного потокового ввода Java. Он реализует интерфейсы `Closeable` и `Readable`. Все методы этого класса (за исключением `markSupported()`) в случае ошибочных ситуаций возбуждают исключение `IOException`. В табл. 19.4 представлен краткий обзор методов класса `Reader`.

Таблица 19.4. Методы, определенные в `Reader`

Метод	Описание
`abstract void close()`	Закрывает входной поток. Последующие попытки чтения сгенерируют `IOException`.
`void mark(int numChars)`	Помещает метку в текущую позицию во входном потоке, которая остается корректной до тех пор, пока не будет прочитано `numChars` символов.
`boolean markSupported()`	Возвращает `true`, если поток поддерживает `mark()/reset()`.
`int read()`	Возвращает целочисленное представление следующего доступного символа вызывающего входного потока. При достижении конца файла возвращает `-1`.
`int read(char buffer[])`	Пытается прочитать до `buffer.length` символов в `buffer` и возвращает количество успешно прочитанных символов. При достижении конца файла возвращает `-1`.
`abstract int read(char buffer[], int offset, int numChars)`	Пытается прочитать до `numChars` символов в `buffer`, начиная с `buffer[offset]`, возвращает количество успешно прочитанных символов. При достижении конца файла возвращает `-1`.
`boolean ready()`	Возвращает `true`, если следующий запрос не будет ожидать. В противном случае возвращает `false`.
`void reset()`	Сбрасывает указатель ввода в ранее установленную позицию метки.
`long skip(long numChars)`	Пропускает `numChars` символов ввода, возвращая количество действительно пропущенных символов.

Writer

`Writer` — абстрактный класс, определяющий символьный потоковый вывод. Реализует интерфейсы `Closeable`, `Flushable` и `Appendable`. Все методы этого класса в ошибочных ситуациях возбуждают исключение `IOException`. Краткий обзор методов класса Writer представлен в табл. 19.5.

Таблица 19.5. Методы, определенные в `Writer`

Метод	Описание
`Writer append(char ch)`	Добавляет `ch` в конец вызывающего выходного потока. Возвращает ссылку на вызывающий поток.
`Writer append(CharSequence chars)`	Добавляет `chars` в конец вызывающего выходного потока. Возвращает ссылку на вызывающий поток.
`Writer append(CharSequence chars, int begin, int end)`	Добавляет поддиапазон `chars`, специфицированный посредством `begin` и `end-1`, в конец вызывающего выходного потока. Возвращает ссылку на вызывающий поток.
`abstract void close()`	Закрывает вызывающий поток. Последующие попытки записи генерируют `IOException`.
`abstract void flush()`	Финализирует выходное состояние так, что все буферы очищаются. То есть, сбрасывает выходные буферы.
`void write(int ch)`	Записывает единственный символ в вызывающий выходной поток. Обратите внимание, что параметр имеет тип `int`, что позволяет вызывать `write` с выражениями без необходимости приведения их обратно к `char`.
`void write(char buffer[])`	Записывает полный массив символов в вызывающий выходной поток.
`abstract void write(char buffer[], int offset, int numChars)`	Записывает поддиапазон `numChars` символов из массива *`buffer`*, начиная с *`buffer[offset]`*, в вызывающий выходной поток.
`void write(String str)`	Пишет *`str`* в вызывающий выходной поток.
`void write(String str, int offset, int numChars)`	Пишет поддиапазон `numChars` символов из строки *`str`*, начиная с указанного смещения `offset`.

FileReader

`FileReader` — класс, создающий `Reader`, который вы можете использовать для чтения содержимого файла. Два наиболее часто используемых его конструктора выглядят так:

```
FileReader(String filePath)
FileReader(File fileObj)
```

Оба могут возбуждать исключение `FileNotFoundException`. Здесь *`filePath`* — полное путевое имя файла, а *`fileObj`* — объект `File`, описывающий файл. Следующий пример показывает, как можно читать строки из файла и печатать их в стандартный выходной поток. Он читает собственный исходный файл, который должен находиться в текущем каталоге.

```
// Демонстрация применения FileReader.
import java.io.*;
class FileReaderDemo {
public static void main(String args[]) throws IOException {
  FileReader fr = new FileReader("FileReaderDemo.java");
  BufferedReader br = new BufferedReader(fr);
  String s;
```

```
  while((s = br.readLine()) != null) {
     System.out.println(s);
  }
  fr.close();
}
}
```

FileWriter

`FileWriter` создает `Writer`, который вы можете применять для записи файла. Его наиболее часто используемые конструкторы:

```
FileWriter(String filePath)
FileWriter(String filePath, boolean append)
FileWriter(File fileObj)
FileWriter(File fileObj, boolean append)
```

Все они могут возбуждать исключение `IOException`. Здесь *`filePath`* — полное путевое имя файла, а `fileObj` — объект `File`, описывающий файл. Если `append` равно `true`, то вывод добавляется в конец файла.

Создание `FileWriter` не зависит от того, существует ли файл. `FileWriter` создаст файл перед его открытием для вывода, когда вы создаете объект. В случае попытки открытия файла, доступного только для чтения, возбуждается исключение `IOException`.

Следующий пример представляет собой версию символьного потока примера, представленного ранее, когда речь шла о `FileOutputStream`. Эта версия создает простой буфер символов, сначала создавая `String`, а затем используя метод `getChars()` для извлечения эквивалентного символьного массива. Затем она создает три файла. Первый, `file.txt`, будет содержать каждый второй символ примера. Второй, `file2.txt`, будет хранить полный набор символов. И, наконец, третий, `file3.txt`, будет содержать только последнюю четверть символов.

```
// Демонстрация применения FileWriter.
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);
  FileWriter f0 = new FileWriter("file1.txt");

  for (int i=0; i < buffer.length; i += 2) {
     f0.write(buffer[i]);
  }
  f0.close();

  FileWriter f1 = new FileWriter("file2.txt");
  f1.write(buffer);
  f1.close();

  FileWriter f2 = new FileWriter("file3.txt");

  f2.write(buffer,buffer.length-buffer.length/4,buffer.length/4);
  f2.close();
}
}
```

CharArrayReader

CharArrayReader — реализация входного потока, использующего символьный массив в качестве источника. Этот класс имеет два конструктора, каждый из которых принимает символьный массив в качестве источника данных:

```
CharArrayReader(char array[])
CharArrayReader(char array[], int start, int numChars)
```

Здесь *array* — входной источник. Второй конструктор создает Reader из подмножества вашего символьного массива, начинающегося с символа в позиции, указанной *start*, и длиной *numChars*.

В следующем примере используется пара CharArrayReaders.

```
// Демонстрация применения CharArrayReader.
import java.io.*;
public class CharArrayReaderDemo {
public static void main(String args[]) throws IOException {
  String tmp = "abcdefghijklmnopqrstuvwxyz";
  int length = tmp.length();
  char c[] = new char[length];
  tmp.getChars(0, length, c, 0);
  CharArrayReader input1 = new CharArrayReader(c);
  CharArrayReader input2 = new CharArrayReader(c, 0, 5);
  int i;
  System.out.println("input1:");
  while((i = input1.read()) != -1) {
     System.out.print((char)i);
  }
  System.out.println();
  System.out.println("input2:");
  while((i = input2.read()) != -1) {
     System.out.print((char)i);
  }
  System.out.println();
}
}
```

Объект input1 конструируется из полного алфавита в нижнем регистре, в то время как input2 содержит только первые пять букв. Вот вывод этой программы:

```
input1:
abcdefghijklmnopqrstuvwxyz
input2:
abcde
```

CharArrayWriter

CharArrayWriter — реализация выходного потока, использующего в качестве места назначения вывода массив. CharArrayWriter имеет два следующих конструктора:

```
CharArrayWriter()
CharArrayWriter(int numChars)
```

В первой форме создается буфер с размером по умолчанию. Во второй буфер создается с размером, равным *numChars*. Буфер находится в поле buf класса CharArrayWriter.

Размер буфера будет при необходимости последовательно увеличиваться. Количество байт, содержащихся в буфере, находится в поле count того же класса. Оба поля — buf и count — являются защищенными.

В следующем примере демонстрируется использование CharArrayWriter в переделанной программе, рассмотренной ранее, когда речь шла о ByteArrayOutputStream. Она выдает тот же вывод, что и предыдущая версия.

```
// Демонстрация применения CharArrayWriter.
import java.io.*;
class CharArrayWriterDemo {
public static void main(String args[]) throws IOException {
  CharArrayWriter f = new CharArrayWriter();
  String s = "This should end up in the array";
  char buf[] = new char[s.length()];
  s.getChars(0, s.length(), buf, 0);
  f.write(buf);
  System.out.println("Буфер в виде 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("\nB FileWriter()");

  FileWriter f2 = new FileWriter("test.txt");
  f.writeTo(f2);
  f2.close();
  System.out.println("Выполнение reset()");
  f.reset();

  for (int i=0; i<3; i++)
     f.write('X');

  System.out.println(f.toString());
}
}
```

BufferedReader

BufferedReader увеличивает производительность за счет буферизации ввода. У него имеются два конструктора:

```
BufferedReader(Reader inputStream)
BufferedReader(Reader inputStream, int bufSize)
```

Первая форма создает буферизованный символьный поток, используя размер буфера по умолчанию. Во втором размер буфера составляет *bufSize*.

Как и в случае с байт-ориентированным потоком, буферизованный символьный входной поток также обеспечивает фундамент поддержки перемещения обратно по потоку в пределах доступного буфера. Чтобы поддержать это, BufferedReader реализует методы mark() и reset(), а BufferedReader.markSupported() возвращает true.

Следующий пример представляет собой переработанную версию примера BufferedInputStream, показанную выше, но использует символьный поток BufferedReader вместо буферизованного байтового потока. Как и ранее, он использует mark() и reset()

для разбора потока на предмет поиска HTML-конструкции с символом авторских прав. Такая ссылка начинается с амперсанда (`&`) и заканчивается точкой с запятой (`;`) без внутренних пробелов. Пример ввода содержит два амперсанда, чтобы продемонстрировать случай, когда `reset()` происходит, а когда нет. Вывод будет тем же, что и раньше.

```
// Использование буферизованного ввода.
import java.io.*;

class BufferedReaderDemo {
public static void main(String args[]) throws IOException {
  String s = "This is a &copy; copyright symbol " +
       "but this is &copy not.\n";
  char buf[] = new char[s.length()];

  s.getChars(0, s.length(), buf, 0);

  CharArrayReader in = new CharArrayReader(buf);
  BufferedReader f = new BufferedReader(in);

  int c;
  boolean marked = false;

  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;
         }
  }
}
}
```

BufferedWriter

BufferedWriter — это Writer, который буферизует вывод. Благодаря применению BufferedWriter, можно увеличить производительность за счет снижения количества операций физической записи в выходной поток.

BufferedWriter имеет следующие два конструктора:

```
BufferedWriter(Writer outputStream)
BufferedWriter(Writer outputStream, int bufSize)
```

Первая форма создает буферизованный поток, использующий буфер с размером по умолчанию. Во втором размер буфера передается в *bufSize*.

PushbackReader

Класс PushbackReader позволяет возвращать во входной поток один или более байт. Это позволяет "заглянуть" во входной буфер. Вот два его конструктора:

```
PushbackReader(Reader inputStream)
PushbackReader(Reader inputStream, int bufSize)
```

Первая форма создает буферизованный поток, позволяющий "втолкнуть" обратно один символ. Во второй форме размер буфера вталкивания передается в *bufSize*.

PushbackReader предоставляет метод unread(), который возвращает один или более символов в вызывающий входной поток. Доступны три формы этого метода:

```
void unread(int ch)
void unread(char buffer[])
void unread(char buffer[], int offset, int numChars)
```

Первая форма вталкивает символ, переданный в *ch*. Этот символ будет затем первым, который вернет последующий вызов read(). Вторая форма возвращает в поток символы из *buffer*. Третья форма вталкивает numChars символов, начиная со смещения *offset* в *buffer*. При попытке возврата символа в полный буфер возбуждается исключение IOException. Следующая программа представляет собой переделанный пример PushBackInputStream, в котором PushBackInputStream заменен PushbackReader. Как и ранее, он показывает, как синтаксический анализатор языка программирования может использовать "вталкивание" в поток для обнаружения отличия между операциями сравнения (==) и присваивания (=).

```
// Демонстрация применения unread().
import java.io.*;

class PushbackReaderDemo {
public static void main(String args[]) throws IOException {
  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);
  PushbackReader f = new PushbackReader(in);
  int c;

  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;
      }
    }
  }
}
```

PrintWriter

Класс `PrintWriter` — по сути, символ-ориентированная версия `PrintStream`. Он реализует интерфейсы `Appendable`, `Closeable` и `Flushable`. `PrintWriter` имеет несколько конструкторов. Для начала рассмотрим следующие конструкторы:

```
PrintWriter(OutputStream outputStream)
PrintWriter(OutputStream outputStream, boolean flushOnNewline)
PrintWriter(Writer outputStream)
PrintWriter(Writer outputStream, boolean flushOnNewline)
```

Здесь *outputStream* специфицирует открытый `OutputStream`, который примет вывод. Параметр *flushOnNewline* управляет автоматическим выталкиванием буфера при каждом вызове методов `println()`, `printf()` или `format()`. Если *flushOnNewline* равен `true`, то происходит автоматическое выталкивание буфера. Если же `false`, то выталкивание не автоматическое. Конструкторы, которые не принимают параметра *flushOnNewline*, не включают автоматического выталкивания.

Следующий набор конструкторов предоставляет простую возможность конструирования `PrintWriter`, который пишет свой вывод в файл.

```
PrintWriter(File outputFile) throws FileNotFoundException
PrintWriter(File outputFile, String charSet)
   throws FileNotFoundException, UnsupportedEncodingException
PrintWriter(String outputFileName) throws FileNotFoundException
PrintWriter(String outputFileName, String charSet)
   throws FileNotFoundException, UnsupportedEncodingException
```

Они позволяют создать `PrintWriter` на основе объекта `File` либо имени файла. В любом случае файл создается автоматически. Любой ранее существовавший файл с тем же именем уничтожается. Будучи созданным, объект `PrintWriter` направляет весь вывод в указанный файл. Вы можете специфицировать кодировку символов, передав ее имя в `charSet`.

`PrintWriter` поддерживает методы `print()` и `println()` для всех типов, включая `Object`. Если аргумент не относится к примитивному типу, то методы `PrintWriter` вызывают метод `toString()` такого объекта и затем выводят его результат.

`PrintWriter` также поддерживает метод `printf()`. Он работает точно так же, как и в классе `PrintStream`, описанном ранее: позволяя специфицировать точный формат данных. Метод `printf()` для `PrintWriter` объявлен следующим образом:

```
PrintWriter printf(String fmtString, Object ... args)
PrintWriter printf(Locale loc, String fmtString, Object ... args)
```

Первая версия пишет *args* в стандартный вывод в формате, указанном в *fmtString*, используя локаль по умолчанию. Второй позволяет специфицировать локаль. Оба возвращают вызывающий объект PrintWriter.

Метод format() также поддерживается. Его общие формы таковы:

```
PrintWriter format(String fmtString, Object ... args)
PrintWriter format(Locale loc, String fmtString, Object ... args)
```

Этот метод работает подобно printf().

Класс Console

В Java SE 6 появился класс Console. Он используется для чтения и записи информации на консоли, если таковая существует. Реализует интерфейс Flushable. Класс Console прежде всего введен для удобства, поскольку большая часть его функциональности доступна через System.in и System.out. Однако его применение позволяет упростить некоторые виды консольных итераций, особенно при чтении строк с консоли.

Console не поддерживает конструкторов. Вместо этого объект Console получается вызовом метода System.console(), показанного ниже:

```
static System.console()
```

Если консоль доступна, возвращается ссылка на нее. В противном случае возвращается null. Консоль не будет доступна во всех классах, поэтому если возвращается null, консольные операции ввода-вывода невозможны.

Console определяет методы, перечисленные в табл. 19.6. Обратите внимание, что методы ввода, такие как readLine(), возбуждают исключение IOException, когда возникают ошибки ввода. IOError — это новое исключение, добавленное в Java SE 6; оно является подклассом Error. Оно означает сбой ввода-вывода, который происходит вне контроля вашей программы. То есть, обычно вы не будете перехватывать IOError. Откровенно говоря, если IOError возникнет в процессе обращения к консоли, обычно это означает катастрофический сбой системы.

Также обратите внимание на методы readPassword(). Эти методы позволяют приложению считывать пароль, не отображая его на экране. Читая пароли, вы должны "обнулять" как массив, содержащий строку, введенную пользователем, так и массив, содержащий правильный пароль, с которой первую строку нужно сравнить. Это уменьшает шансы для вредоносной программы получить пароль посредством сканирования памяти.

Таблица 19.6. Методы, определенные в Console

Метод	Описание
`void flush()`	Выполняет физическую запись буферизованного вывода на консоль.
`Console format(String fmtString, Object... args)`	Выводит на консоль *args*, используя формат, указанный в *fmtString*.
`Console printf(String fmtString, Object... args)`	Выводит на консоль *args*, используя формат, указанный в *fmtString*.
`Reader reader()`	Возвращает ссылку на Reader, соединенный с консолью.

Окончание табл. 19.6

Метод	Описание
`String readLine()`	Читает и возвращает строку, введенную с клавиатуры. Ввод прекращается нажатием клавиши <ENTER>. Если достигнут конец входного потока консоли, возвращается `null`. В случае сбоя инициируется `IOError`.
`String readLine(String` *`fmtString`*`, Object...` *`args`*`)`	Отображает строку приглашения, форматированную в соответствии с *`fmtString`* и *`args`*, затем читает и возвращает строку, введенную с клавиатуры. Ввод прекращается, когда пользователь нажимает <ENTER>. Если достигнут конец входного потока консоли, возвращается `null`. В случае сбоя инициируется `IOError`.
`char[ ] readPassword()`	Читает и возвращает строку, введенную с клавиатуры. Ввод прекращается нажатием клавиши <ENTER>. При этом строка не отображается. Если достигнут конец входного потока консоли, возвращается `null`. В случае сбоя инициируется `IOError`.
`char[ ] readPassword(String` *`fmtString`*`, Object...` *`args`*`)`	Отображает строку приглашения, форматированную в соответствии с *`fmtString`* и *`args`*, затем читает и возвращает строку, введенную с клавиатуры. Ввод прекращается, когда пользователь нажимает <ENTER>. При этом строка не отображается. Если достигнут конец входного потока консоли, возвращается `null`. В случае сбоя инициируется `IOError`.
`PrintWriter writer()`	Возвращает ссылку на `Writer`, соединенный с консолью.

Рассмотрим пример, демонстрирующий класс `Console` в действии.

```
// Демонстрация применения Console.
import java.io.*;
class ConsoleDemo {
public static void main(String args[]) {
  String str;
  Console con;

  // Получить ссылку на консоль.
  con = System.console();

  // Если нет доступной консоли, выход.
  if(con == null) return;

  // Прочесть строку и отобразить ее.
  str = con.readLine("Введите строку: ");
  con.printf("Вот ваша строка: %s\n", str);
}
}
```

Вывод этого примера:

```
Введите строку: Это тест.
Вот ваша строка: Это тест.
```

Использование потокового ввода-вывода

В следующем примере демонстрируется применение некоторых классов символьных потоков Java, а также их методов. Эта программа реализует стандартную команду wc (счетчик слов). Она имеет два режима. Если никаких имен файлов в аргументах не указано, то программа работает со стандартным входным потоком. Если же указан один или более файлов, то программа обрабатывает каждый из них.

```
// Утилита подсчета слов.
import java.io.*;
class WordCount {
public static int words = 0;
public static int lines = 0;
public static int chars = 0;
public static void wc(InputStreamReader isr)
    throws IOException {
  int c = 0;
  boolean lastWhite = true;
  String whiteSpace = " \t\n\r";
  while ((c = isr.read()) != -1) {
    // Подсчет символов
    chars++;
    // Подсчет строк
    if (c == '\n') {
      lines++;
    }
    // Подсчет слов удалением начала слова
    int index = whiteSpace.indexOf(c);
    if(index == -1) {
      if(lastWhite == true) {
        ++words;
      }
      lastWhite = false;
    }
    else {
      lastWhite = true;
    }
  }
  if(chars != 0) {
    ++lines;
  }
}
public static void main(String args[]) {
  FileReader fr;
  try {
    if (args.length == 0) { // Работаем со стандартным входным потоком
      wc(new InputStreamReader(System.in));
    }
    else { // Работаем со списком файлов
      for (int i = 0; i < args.length; i++) {
           fr = new FileReader(args[i]);
           wc(fr);
      }
    }
  }
```

```
    catch (IOException e) {
      return;
    }
    System.out.println(lines + " " + words + " " + chars);
  }
}
```

Метод `wc()` работает с любым входным потоком и подсчитывает количество символов, строк и слов. Он отслеживает соотношение между словами и пробелами в переменной `lastNotWhite`.

При выполнении без аргументов `WordCount` создает объект `InputStream`, используя `System.in` в качестве источника потока. Затем этот поток передается методу `wc()`, который собственно выполняет подсчет. При запуске с одним или более аргументов `WordCount` предполагает, что каждый из них — имя файла, и создает `FileReader` для каждого из них, передавая результирующие объекты `FileReader` методу `wc()`. В любом случае перед выходом печатает результаты.

Усовершенствование `wc()` применением `StreamTokenizer`

Еще лучший способ нахождения шаблонов во входном потоке предлагает другой класс ввода-вывода Java, а именно — `StreamTokenizer`. Подобно `StringTokenizer` из главы 18, `StreamTokenizer` разбивает входной поток на *лексемы*, которые разделяются набором символов. Конструктор этого класса выглядит так:

```
StreamTokenizer(Reader inStream)
```

Здесь *inStream* должен быть некоторой формой `Reader`.

`StreamTokenizer` определяет несколько методов. В данном примере мы используем лишь некоторые из них. Чтобы очистить набор разделителей по умолчанию, используется метод `resetSyntax()`. Набор разделителей по умолчанию настроен для разбиения на слова Java-программ и потому слишком специализирован для нашего примера. Мы объявим, что наши лексемы, или "слова", будут состоять только из непрерывной последовательности видимых символов, отделенных с двух сторон пробелами.

Воспользуемся методом `eollsSignificant()` для обеспечения того, что символы новой строки будут рассматриваться как лексемы, чтобы мы могли подсчитать количество строк наравне с количеством слов. Вот его общая форма:

```
void eolIsSignificant(boolean eolFlag)
```

Если *eolFlag* равен `true`, то символы новой строки будут возвращаться как лексемы. Если же *eolFlag* равен `false`, то символы новой строки игнорируются.

Метод `wordChars()` используется для указания диапазона символов, которые могут встретиться в словах. Его общая форма такова:

```
void wordChars(int start, int end)
```

Здесь *start* и *end* задают диапазон допустимых символов. В данной программе символы с кодами в диапазоне от 33 до 255 представляют допустимые в составе слов.

Пробельные символы указываются с помощью метода `whitespaceChars()`. Его общая форма:

```
void whitespaceChars(int start, int end)
```

Здесь *start* и *end* задают диапазон допустимых пробельных символов.

Следующее слово получается из входного потока вызовом nextToken(). Этот метод возвращает образец лексемы.

StreamTokenizer определяет константы типа int: TT_EOF, TT_EOL, TT_NUMBER и TT_WORD. В нем есть три переменных экземпляра: nval — общедоступная типа double, используемая для хранения значений чисел, если таковые распознаются; sval — общедоступная типа String, служащая для хранения значений любых слов, если таковые распознаны; ttype — общедоступная типа int, содержащая тип только что извлеченной методом nextToken() лексемы. Если лексема является словом, ttype равен TT_WORD, если числом — TT_NUMBER. Если же лексема является одиночным символом, то ttype содержит его значение. При достижении конца строки ttype равен TT_EOL (в предположении, что ранее eolIsSignificant() был вызван с аргументом true). По достижении конца потока ttype принимает значение TT_EOF.

Программа подсчета слов, переделанная с использованием StreamTokenizer, показана ниже.

```
// Расширенная программа подсчета слов, в которой используется
StreamTokenizer
import java.io.*;
class WordCount {
public static int words=0;
public static int lines=0;
public static int chars=0;
public static void wc(Reader r) throws IOException {
  StreamTokenizer tok = new StreamTokenizer(r);
  tok.resetSyntax();
  tok.wordChars(33, 255);
  tok.whitespaceChars(0, ' ');
  tok.eolIsSignificant(true);
  while (tok.nextToken() != tok.TT_EOF) {
     switch (tok.ttype) {
        case StreamTokenizer.TT_EOL:
          lines++;
          chars++;
          break;
        case StreamTokenizer.TT_WORD:
          words++;
          default: // FALLSTHROUGH
          chars += tok.sval.length();
          break;
     }
  }
}
public static void main(String args[]) {
  if (args.length == 0) { // Работаем с stdin
     try {
        wc(new InputStreamReader(System.in));
        System.out.println(lines + " " + words + " " + chars);
     } catch (IOException e) {};
  } else { // Работаем со списком файлов
     int twords = 0, tchars = 0, tlines = 0;
     for (int i=0; i<args.length; i++) {
        try {
           words = chars = lines = 0;
           wc(new FileReader(args[i]));
```

```
            twords += words;
            tchars += chars;
            tlines += lines;
            System.out.println(args[i] + ": " +
            lines + " " + words + " " + chars);
          } catch (IOException e) {
            System.out.println(args[i] + ": ошибка.");
          }
        }
        System.out.println("всего: " +
        tlines + " " + twords + " " + tchars);
      }
    }
    }
```

Сериализация

Сериализация — это процесс записи состояния объекта в байтовый поток. Она удобна, когда нужно сохранить состояние вашей программы в области постоянного хранения, таком как файл. Позднее вы можете восстановить эти объекты, используя процесс десериализации.

Сериализация также необходима в реализации удаленного вызова методов (Remote Method Invocation — RMI). RMI позволяет объекту Java на одной машине обращаться к методу объекта Java на другой машине. Объект может быть применен как аргумент этого удаленного метода. Посылающая машина сериализует объект и передает его. Принимающая машина десериализует его. (Подробнее о RMI будет рассказано в главе 27.)

Предположим, что объект, подлежащий сериализации, ссылается на другие объекты, которые, в свою очередь, имеют ссылки на еще какие-то объекты. Такой набор объектов и отношений между ними формирует ориентированный граф. В этом графе могут присутствовать и циклические ссылки. То есть, объект X может содержать ссылку на объект Y, а объект Y — обратную ссылку на X. Объекты также могут содержать ссылки на самих себя. Средства сериализации и десериализации объектов устроены так, что могут корректно работать во всех этих сценариях. Если вы попытаетесь сериализовать объект, находящийся на вершине такого графа объектов, то все прочие объекты, на которые имеются ссылки, также будут рекурсивно найдены и сериализованы. Аналогично, во время процесса десериализации все эти объекты и их ссылки корректно восстанавливаются.

Ниже приведен обзор интерфейсов и классов, поддерживающих сериализацию.

Serializable

Только объект, реализующий интерфейс `Serializable`, может быть сохранен и восстановлен средствами сериализации. Интерфейс `Serializable` не содержит никаких членов. Он просто используется для того, чтобы указать, что класс может быть сериализован. Если класс является сериализуемым, все его подклассы также сериализуемы.

Переменные, объявленные как `transient`, не сохраняются средствами сериализации. Также не сохраняются переменные `static`.

Externalizable

Средства Java для сериализации и десериализации спроектированы так, что большая часть работы по сохранению и восстановлению состояния объекта выполняется автома-

тически. Однако бывают случаи, когда программисту нужно управлять этим процессом. Например, может окащаться желательным использовать технологии сжатия и шифрования. Интерфейс Externalizable предназначен именно для таких ситуаций.

Интерфейс Externalizable определяет следующие два метода:

```
void readExternal(ObjectInput inStream)
   throws IOException, ClassNotFoundException
void writeExternal(ObjectOutput outStream)
   throws IOException
```

В этих методах *inStream* — это байтовый поток, из которого объект может быть прочитан, а *outStream* — байтовый поток, куда он записывается.

ObjectOutput

Интерфейс ObjectOutput расширяет интерфейс DataOutpit и поддерживает сериализацию объектов. Он определяет методы, показанные в табл. 19.7. Особо отметим метод writeObject(). Он вызывается для сериализации объекта. Все методы этого интерфейса возбуждают исключение IOException в ошибочных ситуациях.

Таблица 19.7. Методы, определенные в ObjectOutput

Метод	Описание
void close()	Закрывает вызывающий поток. Последующие попытки записи генерируют IOException.
void flush()	Финализирует выходное состояние, так что все буферы очищаются. То есть все выходные буферы сбрасываются.
void write(byte *buffer*[])	Записывает массив байт в вызывающий поток.
void write(byte *buffer*[], int *offset*, int *numBytes*)	Записывает поддиапазон numBytes байт из массива *buffer*, начиная с *buffer*[*offset*].
void write(int *b*)	Записывает одиночный байт в вызывающий поток. Из аргумента *b* записывается только младший байт.
void writeObject(Object *obj*)	Записывает объект *obj* в вызывающий поток.

ObjectOutputStream

Класс ObjectOutputStream расширяет класс OutputStream и реализует интерфейс ObjectOutput. Этот класс отвечает за запись объекта в поток. Конструктор его выглядит так:

```
ObjectOutputStream(OutputStream outStream) throws IOException
```

Аргумент *outStream* представляет собой выходной поток, в который могут быть записаны сериализуемые объекты.

Несколько часто используемых методов класса перечислено в табл. 19.8. Все они в ошибочных ситуациях возбуждают исключение IOException. Присутствует также вложенный в ObjectOutputStream класс по имени PutField. Он обслуживает запись постоянных полей, и описание его применения выходит за рамки настоящей книги.

Таблица 19.8. Часто используемые методы `ObjectOutputStream`

Метод	Описание
`void close()`	Закрывает вызывающий поток. Последующие попытки записи генерируют `IOException`.
`void flush()`	Финализирует выходное состояние, так что все буферы очищаются. То есть все выходные буферы сбрасываются.
`void write(byte buffer[])`	Записывает массив байт в вызывающий поток.
`void write(byte buffer[], int offset, int numBytes)`	Записывает поддиапазон *numBytes* байт из массива *buffer*, начиная с *buffer*[*offset*].
`void write(int b)`	Записывает одиночный байт в вызывающий поток. Из аргумента `b` записывается только младший байт.
`void writeBoolean(boolean b)`	Записывает значение `boolean` в вызывающий поток.
`void writeByte(int b)`	Записывает значение `byte` в вызывающий поток. Записываемый байт — младший из аргумента `b`.
`void writeBytes(String str)`	Записывает байты, составляющие строку `str`, в вызывающий поток.
`void writeChar(int c)`	Записывает значение `char` в вызывающий поток.
`void writeChars(String str)`	Записывает символы, составляющие строку *str*, в вызывающий поток.
`void writeDouble(double d)`	Записывает значение `double` в вызывающий поток.
`void writeFloat(float f)`	Записывает значение `float` в вызывающий поток.
`void writeInt(int i)`	Записывает значение `int` в вызывающий поток.
`void writeLong(long l)`	Записывает значение `long` в вызывающий поток.
`final void writeObject(Object obj)`	Записывает объект *obj* в вызывающий поток.
`void writeShort(int i)`	Записывает значение `short` в вызывающий поток.

ObjectInput

Интерфейс `ObjectInput` расширяет интерфейс `DataInput` и определяет методы, перечисленные в табл. 19.9. Он поддерживает сериализацию объектов. Особо стоит отметить метод `readObject()`. Он вызывается для десериализации объекта. Все эти методы возбуждают исключение `IOException` в ошибочных ситуациях. Метод `readObject()` также может возбудить `ClassNotFoundException`.

ObjectInputStream

Класс `ObjectInputStream` расширяет класс `InputStream` и реализует интерфейс `ObjectInput`. `ObjectInputStream` отвечает за чтение объектов из потока. Ниже показан конструктор этого класса.

```
ObjectInputStream(InputStream inStream)
   throws IOException
```

Аргумент `inStream` — это входной поток, из которого должен быть прочитан сериализованный объект.

Таблица 19.9. Методы, определенные в ObjectInput

Метод	Описание
`int available()`	Возвращает количество байт, которые доступны во входном буфере в настоящий момент.
`void close()`	Закрывает вызывающий поток. Последующие попытки чтения вызовут генерацию `IOException`.
`int read()`	Возвращает целочисленное представление следующего доступного байта во вводе. При достижении конца файла возвращается `-1`.
`int read(byte buffer[])`	Пытается прочитать до `buffer.length` байт в `buffer`, начиная с `buffer[offset]`, возвращая количество байт, которые удалось прочитать. При достижении конца файла возвращается `-1`.
`int read(byte buffer[], int offset, int numBytes)`	Пытается прочитать до `numBytes` байт в `buffer`, начиная с `buffer[offset]`, возвращая количество байт, которые удалось прочитать. При достижении конца файла возвращается `-1`.
`Object readObject()`	Читает объект из вызывающего потока.
`Long skip(long numBytes)`	Игнорирует (т.е. пропускает) `numBytes` байт вызывающего потока, возвращая количество действительно пропущенных байт.

Несколько часто используемых методов этого класса показано в табл. 19.10. Все они возбуждают исключение `IOException` в ошибочных ситуациях. Метод `readObject()` также может возбудить `ClassNotFoundException`. Также в `ObjectInputStream` присутствует вложенный класс по имени `GetField`. Он обслуживает чтение постоянных полей, и описание его применения выходит за рамки настоящей книги.

Таблица 19.10. Часто используемые методы, определенные в ObjectInputStream

Метод	Описание
`int available()`	Возвращает количество байт, доступных в данный момент во входном буфере.
`void close()`	Закрывает вызывающий поток. По следующие попытки чтения вызовут генерацию `IOException`.
`int read()`	Возвращает целочисленное представление следующего доступного байта ввода. При достижении конца файла возвращается `-1`.
`int read(byte buffer[], int offset, int numBytes)`	Пытается прочитать до `numBytes` байт в `buffer`, начиная с `buffer[offset]`, возвращая количество байт, которые удалось прочитать. При достижении конца файла возвращается `-1`.
`boolean readBoolean()`	Читает и возвращает значение `boolean` из вызывающего потока.
`byte readByte()`	Читает и возвращает значение `byte` из вызывающего потока.
`char readChar()`	Читает и возвращает значение `char` из вызывающего потока.
`double readDouble()`	Читает и возвращает значение `double` из вызывающего потока.
`double readFloat()`	Читает и возвращает значение `float` из вызывающего потока.
`void readFully(byte buffer[])`	Читает `buffer.length` байт в `buffer`. Возвращает управление, только когда все байты прочитаны.

Окончание табл. 19.10

Метод	Описание
`void readFully(byte buffer[], int offset, int numBytes)`	Читает *numBytes* байт в *buffer*, начиная с *buffer[offset]*. Возвращает управление, только когда прочитано *numBytes* байт.
`int readInt()`	Читает и возвращает значение `int` из вызывающего потока.
`int readLong()`	Читает и возвращает значение `long` из вызывающего потока.
`final Object readObject()`	Читает и возвращает объект из вызывающего потока.
`short readShort()`	Читает и возвращает значение `short` из вызывающего потока.
`int readUnsignedByte()`	Читает и возвращает значение `unsigned byte` из вызывающего потока.
`int readUnsignedShort()`	Читает и возвращает значение `unsigned short` из вызывающего потока.

Пример сериализации

В следующей программе показано, как использовать сериализацию и десериализацию объектов. Начинается он с создания экземпляра объекта `MyClass`. Этот объект имеет три переменных экземпляра с типами `String`, `int` и `double`. Именно эту информацию мы хотим сохранять и восстанавливать.

В программе создается `FileOutputStream`, который ссылается на файл по имени "serial", и для этого файлового потока создается `ObjectOutputStream`. Метод `writeObject()` этого `ObjectOutputStream` затем используется для сериализации объекта. Объект выходного потока очищается и закрывается.

Далее создается `FileInputStream` , который ссылается на файл по имени "serial", и для этого файлового потока создается `ObjectInputStream`. Метод `readObject()` класса `ObjectInputStream` затем используется для десериализации объекта. После этого входной поток закрывается.

Обратите внимание, что `MyClass` определен с реализацией интерфейса `Serializable`. Если бы этого не было, возбуждалось бы исключение `NotSerializableException`. Поэкспериментируйте с этой программой, объявляя некоторые переменные экземпляра `MyClass` как `transient`. Эти данные не будут сохраняться при сериализации.

```
import java.io.*;
public class SerializationDemo {
public static void main(String args[]) {
  // Сериализация объекта
  try {
     MyClass object1 = new MyClass("Hello", -7, 2.7e10);
     System.out.println("object1: " + object1);
     FileOutputStream fos = new FileOutputStream("serial");
     ObjectOutputStream oos = new ObjectOutputStream(fos);
     oos.writeObject(object1);
     oos.flush();
     oos.close();
  }
  catch(IOException e) {
     System.out.println("Во время сериализации возникло исключение: " + e);
     System.exit(0);
  }
```

```
    // Десериализация объекта
    try {
       MyClass object2;
       FileInputStream fis = new FileInputStream("serial");
       ObjectInputStream ois = new ObjectInputStream(fis);
       object2 = (MyClass)ois.readObject();
       ois.close();
       System.out.println("object2: " + object2);
    }
    catch(Exception e) {
       System.out.println("Во время сериализации возникло исключение: " + 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 объектов `object1` и `object2`. Вот ее вывод:

```
object1: s=Hello; i=-7; d=2.7E10
object2: s=Hello; i=-7; d=2.7E10
```

Преимущества потоков

Потоковый интерфейс ввода-вывода в Java предоставляет чистую абстракцию для сложных и часто обременительных задач. Композиция классов фильтрующих потоков позволяет динамически строить собственные настраиваемые потоковые интерфейсы, которые отвечают вашим требованиям к передаче данных. Программы Java, использующие эти абстрактные высокоуровневые классы — `InputStream`, `OutputStream`, `Reader` и `Writer` — будут корректно функционировать в будущем, даже когда появятся новые усовершенствованные конкретные потоковые классы. Как вы увидите в следующей главе, эта модель работает очень хорошо, когда мы переключаемся от набора потоков на основе файлов к сетевым потокам и потокам сокетов. И, наконец, сериализация объектов играет важную роль в Java-программах различных типов. Классы сериализации ввода-вывода Java обеспечивают переносимое решение этой непростой задачи.

20 ГЛАВА

Сеть

Как известно читателям, Java — практически синоним программирования для Internet. К тому есть множество причин, и не последние из них — способность генерировать безопасный, межплатформенный и переносимый код. Однако одна из наиболее важных причин того, что Java является великолепным языком для сетевого программирования, кроется в классах, определенных в пакете `java.net`. Они обеспечивают легкие в использовании средства, с помощью которых программисты всех уровней квалификации могут обращаться к сетевым ресурсам.

Эта глава посвящена пакету `java.net`. Важно подчеркнуть, что сети — очень обширная и сложная тема. В настоящей книге невозможно полностью охватить все средства, содержащиеся в `java.net`. Поэтому в данной главе мы сосредоточим внимание лишь на некоторых основополагающих классах и интерфейсах.

Основы работы с сетью

Прежде чем начать, полезно будет получить представление о ключевых концепциях и терминах, связанных с сетями. В основе сетевой поддержки Java лежит концепция *сокета* (socket). Сокет идентифицирует конечную точку сети. Парадигма сокета появилась в версии 4.2BSD Berkley UNIX в самом начале 80-х гг. По этой причине также используется термин *сокет Беркли*. Сокеты — основа современных сетей, поскольку сокет позволяет отдельному компьютеру обслуживать одновременно как множество разных клиентов, так и множество различных типов информации. Это достигается за счет использования *порта* (port) — нумерованного сокета на определенной машине. Говорят, что серверный процесс "слушает" порт до тех пор, пока клиент не соединится с ним. Сервер в состоянии принять множество клиентов, подключенных к одному и тому же номеру порта, хотя каждый сеанс является уникальным. Чтобы обработать множество клиентских соединений, серверный процесс должен быть многопоточным либо обладать какими-то другими средствами обработки одновременного ввода-вывода.

Сокетные коммуникации происходят по определенному протоколу. *Internet-протокол* (Internet Protocol — IP) — это низкоуровневый маршрутизирующий протокол, который разбивает данные на небольшие пакеты и посылает их через сеть по определенному адресу, что не гарантирует доставки всех этих пакетов по этому адресу. *Протокол управления*

передачей (Transmission Control Protocol — TCP) является протоколом более высокого уровня, обеспечивающий надежную сборку этих пакетов, сортировку и повторную передачу, необходимую для надежной доставки данных. Третий протокол — *протокол пользовательских дейтаграмм* (User Datagram Protocol — UDP), стоящий непосредственно за TCP, может быть использован непосредственно для поддержки быстрой, не требующей постоянного соединения и ненадежной транспортировки пакетов.

Как только соединение установлено, применяется высокоуровневый протокол, зависящий от используемого порта. TCP/IP резервирует первые 1024 порта для специфических протоколов. Многие из них покажутся вам знакомыми, если вы хоть какое-то время потратили на путешествия в океане Internet. Порт номер 21 — для FTP, 23 — для Telnet, 25 — для электронной почты, 43 — для whois; 79 — для finger, 80 — для HTTP, 119 — для netnews; список можно продолжать. На каждый из протоколов возлагается определение того, как клиент должен взаимодействовать с портом.

Например, HTTP — это протокол, используемый серверами и Web-браузерами для передачи гипертекста и графических изображений. Это довольно простой протокол для базового постраничного просмотра информации, предоставляемой Web-серверами. Посмотрим, как оно работает. Когда клиент запрашивает файл с сервера HTTP, это действие известно как *попадание* (hit) и состоит в простой отправке имени файла в определенном формате на предопределенный порт с последующим чтением содержимого этого файла. Сервер также сообщает код состояния, чтобы известить клиент о том, был ли запрос обработан или нет, и по какой причине.

Ключевым компонентом Internet является *адрес*. Каждый компьютер в Internet обладает собственным адресом. Адрес Internet представляет собой число, уникально идентифицирующее каждый компьютер в Internet. Изначально все Internet-адреса состояли из 32-битных значений, организованных в четыре 8-битных значения. Адрес такого типа определен IPv4 (Internet-протокол версии 4). Однако в последнее время на сцену выступает новая схема адресации, называемая IPv6, которая предназначена для того, чтобы поддержать гораздо большее адресное пространство, чем IPv4.

Для обеспечения обратной совместимости с IPv4 младшие 32 бита адреса IPv6 могут содержать в себе корректный адрес IPv4. Таким образом, адресация IPv4 совместима снизу вверх с IPv6. К счастью, имея дело с Java, вам обычно не придется беспокоиться о том, используется адрес IPv4 или IPv6, поскольку Java позаботится обо всех деталях.

Точно так же, как IP-адрес описывает сетевую иерархию, имя адреса Internet, называемое *доменным именем*, описывает местонахождение машины в пространстве имен. Например, `www.osborne.com` относится к домену `com` (зарезервированному для коммерческих сайтов США), имеет имя `osborne` (по названию компании), а `www` идентифицирует сервер, обрабатывающий Web-запросы. Доменное имя Internet отображается на IP-адрес посредством *службы доменных имен* (Domain Name Service — DNS). Это позволяет пользователям работать с доменными именами, в то время как Internet оперирует IP-адресами.

Сетевые классы и интерфейсы

Java поддерживает TCP/IP как за счет расширения уже имеющихся интерфейсов потокового ввода-вывода, представленных в главе 19, так и за счет добавления средств, необходимых для построения объектов ввода-вывода в сети. Java поддерживает семейства протоколов как TCP , так и UDP. TCP применяется для надежного потокового ввода-вывода по сети. UDP поддерживает более простую, а потому быструю модель передачи дейтаграмм от точки к точке. Классы, содержащиеся в пакете `java.net`, перечислены ниже.

Authenticator	Inet6Address	ServerSocket
CacheRequest	InetAddress	Socket
CacheResponse	InetSocketAddress	SocketAddress
ContentHandler	InterfaceAddress (добавлен в Java SE 6)	SocketImpl
CookieHandler	JarURLConnection	SocketPermission
CookieManager (добавлен в Java SE 6)	MulticastSocket	URI
DatagramPacket	NetPermission	URL
DatagramSocket	NetworkInterface	URLClassLoader
DatagramSocketImpl	PasswordAuthentication	URLConnection
HttpCookie (добавлен в Java SE 6)	Proxy	URLDecoder
HttpURLConnection	ProxySelector	URLEncoder
IDN (добавлен в Java SE 6)	ResponseCache	URLStreamHandler
Inet4Address	SecureCacheResponse	

Интерфейсы пакета `java.net` перечислены далее:

ContentHandlerFactory	DatagramSocketImplFactory	SocketOptions
CookiePolicy (добавлен в Java SE 6)	FileNameMap	
CookieStore (добавлен в Java SE 6)	SocketImplFactory	

В следующем разделе мы рассмотрим основные сетевые классы и продемонстрируем несколько примеров их применения. Как только вы поймете устройство сетевых классов, то сможете строить собственные на их основе.

InetAddress

Класс `InetAddress` используется для инкапсуляции как числового IP-адреса, так и доменного имени для этого адреса. Вы взаимодействуете с классом, используя имя IP-хоста, что намного удобнее и понятнее, чем IP-адрес. Класс `InetAddress` скрывает внутри себя число. Он может работать как с адресами IPv4, так и с IPv6.

Методы-фабрики

Класс `InetAddress` не имеет видимых конструкторов. Чтобы создать объект `InetAddress`, вы должны использовать один из доступных методов-фабрик. *Методы-фабрики* (factory method) — это просто соглашение, в соответствии с которым статические методы класса возвращают экземпляр этого класса. Это делается вместо перегрузки конструктора с различными списками параметров, когда наличие уникальных имен методов делает результат более ясным. Ниже приведены три часто используемых метода-фабрики `InetAddress`.

```
static InetAddress getLocalHost( )
   throws UnknownHostException
static InetAddress getByName(String hostName)
   throws UnknownHostException
static InetAddress[ ] getAllByName(String hostName)
   throws UnknownHostException
```

Метод `getLocalHost()` просто возвращает объект `InetAddress`, представляющий локальный хост. Метод `getByName()` возвращает `InetAddress` хоста, чье имя ему передано. Если эти методы оказываются не в состоянии получить имя хоста, они возбуждают исключение `UnknownHostException`.

Когда одно имя используется для представления нескольких машин в Internet — это обычное явление. В мире Web-серверов это единственный путь предоставления некоторой степени масштабируемости. Метод-фабрика `getAllByName()` возвращает массив `InetAddress`, представляющий все адреса, в которые преобразуется конкретное имя. Он также возбуждает исключение `UnknownHostException` в случае, если не может преобразовать имя в хотя бы один адрес.

`InetAddress` также включает фабричный метод `getDyAddress()`, который принимает IP-адрес и возвращает объект `InetAddress`. Причем могут использоваться как адреса IPv4, так и IPv6.

В следующем примере распечатываются адреса и имена локальной машины, а также двух широко известных Internet-сайтов.

```
// Демонстрация применения 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("osborne.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 показан вывод, сгенерированный этой программой (конечно, код, который вы увидите на своей машине, может несколько отличаться).

```
default/206.148.209.138
osborne.com/198.45.24.162
www.nba.com/64.5.96.214
www.nba.com/64.5.96.216
```

Методы экземпляра

В классе `InetAddress` также имеется несколько других методов, которые могут быть использованы с объектами, возвращенными методами, о которых мы говорили только что. Некоторые из наиболее часто применяемых методов перечислены в табл. 20.1.

Поиск Internet-адресов осуществляется в серии иерархических кэшированных служб. Это значит, что ваш локальный компьютер может получить определенное отображение имени на IP-адрес автоматически, как для себя, так и для ближайших серверов. Для всех прочих имен он может обращаться к DNS-серверам, откуда получит информацию об IP-адресах. Если такой сервер не имеет информации об определенном адресе, он может обратиться к следующему удаленному сайту и запросить эту информацию у него. Это может продолжаться вплоть до корневого сервера, и упомянутый процесс может потребовать длительного времени, так что разумно построить структуру вашего кода таким образом, чтобы информация об IP-адресах локально кэшировалась, и ее не приходилось искать каждый раз заново.

Таблица 20.1. Часто используемые методы класса `InetAddress`

Метод	Описание
`boolean equals(Object other)`	Возвращает `true`, если объект имеет тот же адрес Internet, что и *other*.
`byte[] getAddress()`	Возвращает байтовый массив, представляющий IP-адрес в порядке байт сети.
`String getHostAddress()`	Возвращает строку, представляющую адрес хоста, ассоциированного с объектом `InetAddress`.
`String getHostName()`	Возвращает строку, представляющую имя хоста, ассоциированного с объектом `InetAddress`.
`boolean isMulticastAddress()`	Возвращает `true`, если адрес является групповым, в противном случае возвращает `false`.
`String toString()`	Возвращает строку, включающую имя хоста и IP-адрес для удобства.

`Inet4Address` и `Inet6Address`

Начиная с версии 1.4, в Java включена поддержка адресов IPv6. В связи с этим были созданы два подкласса `InetAddress`: `Inet4Address` и `Inet6Address`. `Inet4Address` представляет традиционные адреса IPv4, а `Inet6Address` инкапсулируют адреса IPv6 нового стиля. Поскольку оба они являются подклассами `InetAddress`, ссылки `InetAddress` могут указывать на них. Это единственный способ, благодаря которому удалось добавить в Java функциональность IPv6, не нарушая работы существующего кода и не добавляя большого количества новых классов. В большинстве случаев вы просто можете использовать `InetAddress`, работая с IP-адресами, поскольку этот класс приспособлен для обоих стилей.

Клиентские сокеты TCP/IP

Сокеты TCP/IP применяются для реализации надежных двунаправленных, постоянных соединений между точками — хостами в Internet на основе потоков. Сокет может использоваться для подключения системы ввода-вывода Java к другим программам, которые могут находиться как на локальной машине, так и на любой другой машине в Internet.

На заметку! *Аплеты могут устанавливать сокетные соединения только с тем хостом, с которого они были загружены. Это ограничение введено в связи с тем, что было бы опасно для аплетов, загруженных через брандмауэр, иметь доступ к любой произвольной машине.*

В Java существуют два вида сокетов TCP. Один — для серверов, другой — для клиентов. Класс `ServerSocket` предназначен быть "слушателем", который ожидает подключения клиентов прежде, чем что-либо делать. То есть `ServerSocket` предназначен для серверов. Класс `Socket` предназначен для клиентов. Он разработан так, чтобы соединяться с серверными сокетами и инициировать обмен по протоколу. Поскольку клиентские советы наиболее часто применяются в Java-приложениях, их мы и рассмотрим здесь. В табл. 20.2 описаны два конструктора, используемые для создания клиентских советов.

Таблица 20.2. Конструкторы класса `Socket`

Конструктор	Описание
`Socket(String hostName, int port) throws UnknownHostException, IOException`	Создает сокет, подключенный к именованному хосту и порту.
`Socket(InetAddress ipAddress, int port) throws IOException`	Создает сокет, используя ранее существующий объект `InetAddress` и порт.

`Socket` определяет несколько методов экземпляров. Например, `Socket` может быть просмотрен в любое время на предмет извлечения информации об адресе и порте, ассоциированной с ним. Для этого применяются методы, перечисленные в табл. 20.3.

Таблица 20.3. Методы экземпляра `Socket`

Метод	Описание
`InetAddress getInetAddress()`	Возвращает `InetAddress`, ассоциированный с объектом `Socket`. В случае если сокет не подключен, возвращает `null`.
`int getPort()`	Возвращает удаленный порт, к которому подключен вызывающий объект `Socket`. Если сокет не подключен, возвращает 0.
`int getLocalPort()`	Возвращает локальный порт, к которому привязан вызывающий объект `Socket`. Если сокет не привязан, возвращает `-1`.

Вы можете получить доступ к входному и выходному потокам, ассоциированным с `Socket`, с использованием методов `getInputStream()` и `getOuptutStream()`, которые описаны в табл. 20.4. Каждый из них может возбуждать исключение `IOException`, если сокет стал недействительным из-за утери соединения. Эти потоки используются точно так же, как потоки ввода-вывода, рассмотренные в главе 19, для получения и приема данных.

Таблица 20.4. Методы доступа к входному и выходному потокам, ассоциированным с `Socket`

Метод	Описание
`InputStream getInputStream() throws IOException`	Возвращает `InetAddress`, ассоциированный с вызывающим сокетом.
`OutputStream getOutputStream() throws IOException`	Возвращает `OutputStream`, ассоциированный с вызывающим сокетом.

Доступно также еще несколько других методов, включая `connect()`, позволяющий специфицировать новое подключение, `isConnected()`, возвращающий `true`, если сокет подключен к серверу, `isBound()`, возвращающий `true`, если сокет привязан к адресу, и `isClosed()`, возвращающий `true`, когда сокет закрыт.

Следующая программа представляет простой пример применения `Socket`. Она открывает соединение с портом whois (порт 43) на сервере InterNIC, посылает сокету аргументы командной строки, а затем печатает возвращенные данные. InterNIC пытается трактовать аргумент как зарегистрированное доменное имя Internet, а затем возвращает IP-адрес и контактную информацию для этого сайта.

```
// Демонстрация работы с сокетами.
import java.net.*;
import java.io.*;
```

```
class Whois {
public static void main(String args[]) throws Exception {
  int c;

  // Создает сокетное соединение с internic.net, порт 43.
  Socket s = new Socket("internic.net", 43);

  // Получает входной и выходной потоки.
  InputStream in = s.getInputStream();
  OutputStream out = s.getOutputStream();

  // Конструирует строку запроса.
  String str = (args.length == 0 ? "osborne.com" : args[0]) + "\n";

  // Преобразует в байты.
  byte buf[] = str.getBytes();

  // Посылает запрос.
  out.write(buf);

  // Читает и отображает ответ.
  while ((c = in.read()) != -1) {
     System.out.print((char) c);
  }
  s.close();
}
}
```

Если, к примеру, вы запросите информацию об `osborne.com`, то получите нечто вроде следующего:

```
Whois Server Version 1.3

Domain names in the .com, .net, and .org domains can now be registered
with many different competing registrars. Go to http://www.internic.net
for detailed information.

   Domain Name: OSBORNE.COM
   Registrar: NETWORK SOLUTIONS, INC.
   Whois Server: whois.networksolutions.com
   Referral URL: http://www.networksolutions.com
   Name Server: NS1.EPPG.COM
   Name Server: NS2.EPPG.COM
.
.
.
```

Вот как работает эта программа. Сначала конструируется `Socket`, специфицирующий имя хоста “internic.net” и номер порта 43. `Internic.net` — это Web-сайт InterNIC, обрабатывающий запросы whois. Порт 43 предназначен именно для этой службы. Затем и входной, и выходной потоки открываются в сокете. Далее конструируется строка, содержащая имя Web-сайта, информацию о котором вы хотите получить. В данном случае, если никакой сайт не указан в командной строке, используется “osborne.com”. Строка преобразуется в байтовый массив и отправляется в сокет. После этого ответ читается из сокета и результат отображается на экране.

Класс URL

Предыдущий пример довольно-таки невразумителен, поскольку в настоящее время в Internet не ассоциируется со старыми протоколами, такими как whois, finger и FTP. Здесь царствует WWW — всемирная паутина (World Wide Web). Web — слабо связанная коллекция высокоуровневых протоколов и форматов файлов, унифицированным образом используемых Web-браузерами. Одним из наиболее важных аспектов Web является то, что Тим Бернерс-Ли (Tim Berners-Lea) предложил масштабируемый способ нахождения всех ресурсов в Internet. Как только вы можете однозначно именовать что-либо, это становится очень мощной парадигмой. Именно это и делает унифицированный локатор ресурсов (Uniform Resource Locator — URL).

URL обеспечивает довольно четкую форму уникальной идентификации адресной информации в Web. Внутри библиотеки классов Java класс `URL` представляет простой согласованный программный интерфейс для доступа к информации по всей сети Internet посредством использования URL.

Все URL разделяют один и тот же базовый формат, хотя и допускающий некоторые вариации. Приведем два примера: `http://www.osborne.com/` и `http://www.osborne.com:80/index.htm`. Спецификация URL основана на четырех компонентах. Первый — используемый протокол, отделяемый от остальной части локатора двоеточием (`:`). Распространенными протоколами являются HTTP, FTP, gopher и file, хотя в наши дни почти все осуществляется через HTTP (фактически большинство браузеров корректно работают, даже если вы исключите из спецификации URL фрагмент “http://”). Второй компонент — имя хоста или IP-адрес, используемый хостом; он отделяется слева двойным слэшем (`//`), а справа — слэшем (`/`) или, необязательно — двоеточием (`:`). Третий компонент — номер порта, является необязательным параметром, отделяемым слева от имени хоста двоеточием, а справа — слэшем (`/`) (Если 80 является портом по умолчанию для протокола HTTP, то указывать “:80” излишне.) Четвертая часть — действительный путь к файлу. Большинство серверов HTTP добавляют имя файла `index.html` или `index.htm` к URL, которые указывают непосредственно на какой-то каталог. Таким образом, `http://www.osborne.com/` — это то же самое, что и `http://www.osborne.com/index.htm`.

Java-класс `URL` имеет несколько конструкторов; каждый из них может возбуждать исключение `MalformedURLException`. Одна из часто используемых форм специфицирует URL в виде строки, идентичной тому, что вы видите в браузере:

```
URL(String urlSpecifier) throws MalformedURLException
```

Следующие две формы конструктора позволяют вам разбить URL на части-компоненты:

```
URL(String protocolName, String hostName, int port, String path)
   throws MalformedURLException
URL(String protocolName, String hostName, String path)
   throws MalformedURLException
```

Другой часто используемый конструктор позволяет указывать существующий URL в качестве ссылочного контекста, и затем создать из этого контекста новый URL. Хотя это звучит несколько запутано, на самом деле это очень просто и удобно.

```
URL(URL urlObj, String urlSpecifier) throws MalformedURLException
```

Следующий пример создает URL страницы загрузки Osborne, а затем просматривает его свойства:

```
// Демонстрация применения URL.
import java.net.*;
class URLDemo {
public static void main(String args[]) throws MalformedURLException {
  URL hp = new URL("http://www.osborne.com/downloads");
  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("Целиком: " + hp.toExternalForm());
}
}
```

Запустив это, вы получите:

```
Протокол: http
Порт: -1
Хост: www.osborne
Файл: /downloads
Целиком: http://www.osborne/downloads
```

Обратите внимание на порт `-1`; это означает, что порт явно не установлен. Передав объект `URL`, вы можете извлечь данные, ассоциированные с ним. Чтобы получить доступ к действительным битам или информации по `URL`, создайте из него объект `URLConnection`, используя его метод `openConnection()`, как показано ниже:

```
urlc = url.openConnection()
```

`openaConnection()` имеет следующую общую форму:

```
URLConnection openConnection() throws IOException
```

Он возвращает объект `URLConnection`, ассоциированный с вызывающим объектом `URL`. Обратите внимание, что он может возбуждать исключение `IOException`.

URLConnection

`URLConnection` — это класс общего назначения, предназначенный для доступа к атрибутам удаленного ресурса. Однажды установив соединение с удаленным сервером, вы можете использовать `URLConnection` для просмотра свойств удаленного объекта, прежде чем транспортировать его локально. Эти атрибуты представлены в спецификации протокола HTTP и, как таковые, имеют смысл только для объектов `URL`, использующих протокол HTTP.

`URLConnection` определяет несколько методов. Некоторые из них перечислены в табл. 20.5.

Обратите внимание, что `URLConnection` определяет несколько методов, управляющих заголовочной информацией. Заголовок состоит из пар ключей и значений, представленных в виде строк. Используя `getHeaderField()`, вы можете получить значение, ассоциированное с ключом заголовка. Вызывая `getHeaderField()`, можно получить карту, содержащую все заголовки. Несколько стандартных заголовочных полей доступны непосредственно через такие методы, как `getDate()` и `getContentType()`.

Таблица 20.5. Некоторые методы класса `URLConnection`

Метод	Описание
`int getContentLength()`	Возвращает размер содержимого, ассоциированного с ресурсом. Если длина недоступна, возвращается `-1`.
`String getContentType()`	Возвращает тип содержимого, найденного в ресурсе. Это значение поля заголовка `content-type`. Возвращает `null`, если тип содержимого недоступен.
`long getDate()`	Возвращает время и дату ответа, представленное в миллисекундах, прошедших с 1 января 1970 г.
`long getExpiration()`	Возвращает время и дату устаревания ресурса, представленное в миллисекундах, прошедших с 1 января 1970 г. Если дата устаревания недоступна, возвращается ноль.
`String getHeaderField(int idx)`	Возвращает значение заголовочного поля по индексу *idx*. (Индексы полей заголовка нумеруются, начиная с 0.) Возвращает `null`, если значение *idx* превышает количество полей.
`String getHeaderField(String fieldName)`	Возвращает значение заголовочного поля, чье имя указано в *fieldName*. Возвращает `null`, если указанное поле не найдено.
`String getHeaderFieldKey(int idx)`	Возвращает ключ заголовочного поля по индексу *idx*. (Индексы полей заголовка нумеруются, начиная с 0.) Возвращает `null`, если значение *idx* превышает количество полей.
`Map<String, List<String>> getHeaderFields()`	Возвращает карту, содержащую все заголовочные поля вместе с их значениями.
`long getLastModified()`	Возвращает время и дату последней модификации ресурса, представленные в миллисекундах, прошедших после 1 января 1970 г. Если эта информация недоступна, возвращается ноль.
`InputStream getInputStream() throws IOException`	Возвращает `InputStream`, привязанный к ресурсу. Данный поток может использоваться для получения содержимого ресурса.

Следующий пример создает `URLConnection`, используя метод `openConnection()` объекта `URL`, а затем применяет его для проверки свойств и содержимого документа:

```
// Демонстрация применения 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("Нет информации о дате.");
    else
      System.out.println("Дата: " + new Date(d));

    // получить тип содержимого
    System.out.println("Тип содержимого: " + hpCon.getContentType());

    // получить дату устаревания
    d = hpCon.getExpiration();
    if(d==0)
      System.out.println("Нет информации о сроке действия.");
    else
      System.out.println("Устареет: " + new Date(d));

    // получить дату последней модификации
    d = hpCon.getLastModified();
    if(d==0)
      System.out.println("Нет информации о дате последней модификации.");
    else
      System.out.println("Дата последней модификации: " + new Date(d));

    // получить длину содержимого
    int len = hpCon.getContentLength();
    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();
      int i = len;
      while (((c = input.read()) != -1)) { // && (--i > 0)) {
        System.out.print((char) c);
      }
      input.close();
    } else {
      System.out.println("Содержимое недоступно.");
    }
  }
  }
```

Эта программа устанавливает HTTP-соединение с `www.internic.net` через порт 80. Затем она отображает несколько заголовочных значений и извлекает содержимое. Приведем первые строки вывода (точное их содержание будет меняться со временем):

```
Дата: Thu Jun 08 14:41:35 CDT 2006
Тип содержимого: text/html
Нет информации о сроке действия.
Дата последней модификации: Wed Oct 05 19:49:29 CDT 2005
Длина содержимого: 4917
=== Содержимое ===
<html>
<head>
<title>InterNIC | The Internet's Network Information Center</title>
<meta name="keywords"
  content="internic,network information, domain registration">
```

```
<style type="text/css">
<!--
p, li, td, ul { font-family: Arial, Helvetica, sans-serif}
-->
</style>
</head>
```

HttpURLConnection

Java предлагает подкласс `URLConnection`, обеспечивающий поддержку соединений HTTP. Этот класс называется `HttpURLConnection`. Вы получаете `HttpURLConnection` точно так же, как было показано — вызовом `openConnection()` объекта `URL`, но результат следует приводить к типу `HttpURLConnection`. (Конечно, необходимо убедиться в том, что вы действительно открыли соединение HTTP.) Получив ссылку на объект `HttpURLConnection`, вы можете вызывать любые его методы, унаследованные от `URLConnection`. Вы также можете использовать любые методы, определенные в `HttpURLConnection`. Некоторые методы перечислены в табл. 20.6.

Таблица 20.6. Некоторые методы класса `HttpURLConnection`

Метод	Описание
`static boolean getFollowRedirects()`	Возвращает `true`, если автоматически следует перенаправление, и `false` в противном случае.
`String getRequestMethod()`	Возвращает строковое представление метода выполнения запроса. По умолчанию используется метод GET. Доступны другие методы, такие как POST.
`int getResponseCode() throws IOException`	Возвращает код ответа HTTP. Если код ответа не может быть получен, возвращается `-1`. При разрыве соединения возбуждается исключение `IOException`.
`String getResponseMessage() throws IOException`	Возвращает сообщение ответа, ассоциированное с кодом ответа. Если никакого сообщения недоступно, возвращает `null`.
`static void setFollowRedirects(boolean` *how*`)`	Если *how* равно `true`, значит, перенаправление осуществляется автоматически. Если же *how* равно `false`, значит, этого не происходит. По умолчанию перенаправление осуществляется автоматически.
`void setRequestMethod(String` *how*`) throws ProtocolException`	Устанавливает метод, которым выполняются HTTP-запросы, в соответствии с указанным в *how*. По умолчанию принят метод GET, но доступны также другие варианты, такие как POST. Если указано неправильное значение *how*, возбуждается исключение `ProtocolException`.

В следующей программе демонстрируется работа с `HttpURLConnection`. Сначала она устанавливает соединение с `www.google.com`. Затем отображает метод запроса, код ответа и сообщение ответа. И, наконец, отображает ключи и значения в заголовке ответа.

```
// Демонстрация применения 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();

  // Отображение метода запроса.
  System.out.println("Метод запроса: " + hpCon.getRequestMethod());

  // Отображение кода ответа.
  System.out.println("Код ответа: " + hpCon.getResponseCode());

  // Отображение сообщения ответа.
  System.out.println("Сообщение ответа: " + hpCon.getResponseMessage());

  // Получить список полей заголовка и набор его ключей.
  Map<String, List<String>> hdrMap = hpCon.getHeaderFields();
  Set<String> hdrField = hdrMap.keySet();
  System.out.println("\nЗдесь следует заголовок:");

  // Отобразить все ключи и значения заголовка.
  for(String k : hdrField) {
     System.out.println("Ключ: " + k + " Значение: " + hdrMap.get(k));
  }
}
}
```

Вывод этой программы показан ниже (разумеется, точный ответ, возвращенный `www.google.com`, будет меняться с течением времени).

```
Метод запроса: GET
Код ответа: 200
Сообщение ответа: OK

Здесь следует заголовок:
Ключ: Set-Cookie Value:
[PREF=ID=4fbe939441ed966b:TM=1150213711:LM=1150213711:S=Qk81
WCVtvYkJOdh3; expires=Sun, 17-Jan-2038 19:14:07 GMT; path=/;
   domain=.google.com]
Ключ: null Value: [HTTP/1.1 200 OK]
Ключ: Date Value: [Tue, 13 Jun 2006 15:48:31 GMT]
Ключ: Content-Type Value: [text/html]
Ключ: Server Value: [GWS/2.1]
Ключ: Transfer-Encoding Value: [chunked]
Ключ: Cache-Control Value: [private]
```

Обратите внимание, как отображаются ключи и значения заголовка. Во-первых, карта ключей и заголовков получается вызовом `getHeaderFields()` (который унаследован от `URLConnection`). Затем набор ключей заголовка извлекается вызовом `keySet()` карты. Далее выполняется циклический проход по всему набору посредством стиля “for-each” цикла `for`. Значение, ассоциированное с каждым ключом, получается из карты вызовом `get()`.

Класс URI

Относительно недавним дополнением к Java стал класс URI, инкапсулирующий *универсальный идентификатор ресурса* (Uniform Resource Identifier — URI). URI очень похож на URL. На самом деле URL представляет собой подмножество URI. URI предоставляет стандартный способ идентификации ресурсов. URL также описывает доступ к ресурсу.

Cookie-наборы

Пакет java.net включает классы и интерфейсы, помогающие управлять cookie-наборами, которые могут использоваться для создания HTTP-сеансов с поддержкой состояния (в противоположность таковым без поддержки состояния). К таким классам относятся CookieHandler, CookieManager и HttpCookie, а к интерфейсам — CookiePolicy и CookieStore. Все, кроме CookieHandler, были добавлены в Java SE 6. (CookieHandler появился в JDK 5.) Создание HTTP-сеанса с поддержкой состояния выходит за рамки настоящей книги.

На заметку! *Информацию о применении cookie-наборов с сервлетами читайте в главе 31.*

Серверные сокеты TCP/IP

Как уже упоминалось, в Java имеются различные классы сокетов, которые должны применяться для создания серверных приложений. Класс ServerSocket используется для создания серверов, которые прослушивают обращения как локальных, так и удаленных клиентских программ, желающих установить соединения с ними через открытые порты. ServerSocket довольно-таки сильно отличается от обычных Socket. Когда вы создаете ServerSocket, он регистрирует себя в системе в качестве заинтересованного в клиентских соединениях. Конструкторы ServerSocket отражают номер порта, через который вы хотите принимать соединения, а также — необязательно — длину очереди для данного порта. Длина очереди сообщает системе о том, сколько клиентских соединений можно удерживать, прежде чем начать просто отклонять попытки подключения. По умолчанию установлено 50. При определенных условиях конструкторы могут возбуждать исключение IOException. Конструкторы этого класса описаны в табл. 20.7.

Таблица 20.7. Конструкторы класса ServerSocket

Конструктор	Описание
ServerSocket(int *port*) throws IOException	Создает серверный сокет на указанном порте с длиной очереди 50.
ServerSocket(int *port*, int *maxQueue*) throws IOException	Создает серверный сокет на указанном порте с максимальной длиной очереди в *maxQueue*.
ServerSocket(int *port*, int *maxQueue*, InetAddress *localAddress*) throws IOException	Создает серверный сокет на указанном порте с максимальной длиной очереди в *maxQueue*. На групповом хосте *localAddress* указывает IP-адрес, к которому привязан сокет.

ServerSocket включает метод по имени accept(), представляющий собой блокирующий вызов, который будет ожидать от клиента инициации соединений, и затем возвратит нормальный объект Socket, который далее может служить для взаимодействия с клиентом.

Дейтаграммы

Сетевое взаимодействие в стиле TCP/IP подходит для большинства сетевых нужд. Оно обеспечивает сериализуемые, предсказуемые и надежные потоки пакетов данных. Тем не менее, это обходится далеко не даром. TCP включает множество сложных алгоритмов управления потоками в нагруженных сетях, а также самые пессимистические предположения относительно утери пакетов. Это порождает в некоторой степени неэффективный способ транспортировки данных. В качестве альтернативы можно использовать дейтаграммы.

Дейтаграммы (datagramms) — это порции информации, передаваемые между машинами. В некотором отношении они подобны сильным броскам тренированного, но подслеповатого кетчера в сторону третьего бейсмена. Как только дейтаграмма запущена в сторону нужной цели, нет никаких гарантий, что она достигнет цели, или кто-нибудь окажется на месте, чтобы ее подхватить. Точно так же, когда дейтаграмма принимается, нет никакой гарантии, что она не была повреждена в пути, или что ее отправитель все еще ожидает ответа.

Java реализует дейтаграммы поверх протокола UDP, используя для этого два класса: DatagramPacket — контейнер данных, и DatagramSocket — механизм, используемый для обслуживания DatagramPacket. Рассмотрим их более подробно.

DatagramSocket

DatagramSocket определяет четыре общедоступных конструктора:

```
DatagramSocket( ) throws SocketException
DatagramSocket(int port) throws SocketException
DatagramSocket(int port, InetAddress ipAddress) throws SocketException
DatagramSocket(SocketAddress address) throws SocketException
```

Первый конструктор создает DatagramSocket, связанный с любым незанятым портом локального компьютера. Второй создает DatagramSocket, связанный с портом, указанным в *port*. Третий конструирует DatagramSocket, связанный с указанным портом и InetAddress. Четвертый конструирует DatagramSocket, связанный с заданным SocketAddress. SocketAddress — абстрактный класс, реализуемый конкретным классом InetSocketAddress. InetSocketAddress инкапсулирует IP-адрес с номером порта. Все конструкторы могут возбуждать исключение SocketException в случае возникновения ошибок во время создания сокета.

DatagramSocket определяет множество методов. Два наиболее важных из них — это send() и receive(), которые представлены ниже:

```
void send(DatagramPacket packet) throws IOException
void receive(DatagramPacket packet) throws IOException
```

Метод send() отправляет порту пакет, указанный в *packet*. Метод приема ожидает приема через порт пакета, указанного в *packet*, и возвращает результат.

Другие методы предоставляют вам доступ к различным атрибутам, ассоциированным с DatagramSocket. Эти методы перечислены в табл. 20.8.

Таблица 20.8. Методы класса DatagramSocket

Метод	Описание
`InetAddress getInetAddress()`	Если сокет подключен, возвращается адрес. В противном случае возвращается `null`.
`int getLocalPort()`	Возвращает номер локального порта.
`int getPort()`	Возвращает номер порта, к которому подключен сокет. Возвращает `-1`, если сокет не подключен ни к какому порту.
`boolean isBound()`	Возвращает `true`, если сокет привязан к адресу, в противном случае возвращает `false`.
`boolean isConnected()`	Возвращает `true`, если сокет подключен к серверу, в противном случае возвращает `false`.
`void setSoTimeout(int millis) throws SocketException`	Устанавливает период таймаута в число миллисекунд, переданное в *millis*.

DatagramPacket

DatagramPacket определяет множество конструкторов. Вот четыре из них:

```
DatagramPacket(byte data[ ], int size)
DatagramPacket(byte data[ ], int offset, int size)
DatagramPacket(byte data[ ], int size, InetAddress ipAddress, int port)
DatagramPacket(byte data[ ], int offset, int size, InetAddress ipAddress, int port)
```

Первый конструктор специфицирует буфер, который будет принимать данные, и размер пакета. Он используется для приема данных через DatagramSocket. Вторая форма позволяет вам указать смещение в буфере, куда должны быть помещены данные. Третья форма указывает целевой адрес и порт, используемые DatagramSocket для определения того, куда данные пакета будут отправлены. Четвертая форма передает пакеты, начиная с указанного смещения в данных. Первые две формы следует воспринимать как построение "ящика", а вторые две — как "начинку" и адресацию на конверте.

DatagramPacket определяет несколько методов, включая представленные здесь, которые предоставляют доступ к адресу и номеру порта пакета, наряду с raw-данными и их длиной (табл. 20.9). В общем случае методы get используются в принимаемых пакетах, а методы set — в отправляемых.

Таблица 20.9. Методы класса DatagramPacket

Метод	Описание
`InetAddress getAddress()`	Возвращает адрес источника (для принимаемых дейтаграмм) или места назначения (для отправляемых дейтаграмм).
`byte[] getData()`	Возвращает байтовый массив данных, содержащихся в дейтаграмме. В основном используется для извлечения данных из дейтаграммы после ее приема.

Окончание табл. 20.9

Метод	Описание
`int getLength()`	Возвращает длину корректных данных, содержащихся в байтовом массиве, который должен быть возвращен из метода `getData()`. Это может не совпадать с полной длиной байтового массива.
`int getOffset()`	Возвращает начальный индекс данных.
`int getPort()`	Возвращает номер порта.
`void setAddress(InetAddress` *`ipAddress`*`)`	Устанавливает адрес, по которому отправляется пакет. Адрес указывается в *`ipAddress`*.
`void setData(byte[]` *`data`*`)`	Устанавливает данные в *`data`*, смещение — в ноль, а длину — в количество байт *`data`*.
`void setData(byte[]` *`data`*`, int` *`idx`*`, int` *`size`*`)`	Устанавливает данные в *`data`*, смещение — в *`idx`*, а длину в *`size`*.
`void setLength(int` *`size`*`)`	Устанавливает длину пакета в *`size`*.
`void setPort(int` *`port`*`)`	Устанавливает порт в *`port`*.

Пример работы с дейтаграммами

В следующем примере реализуется очень простое сетевое взаимодействие: клиент и сервер. Сообщения набираются в окне на сервере и передаются по сети на сторону клиента, где и отображаются.

```
// Демонстрация применения 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 завершил работу.");
         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 ограничен конструктором `DtagramSocket` для запуска между портами локальной машины. Чтобы использовать программы, выполните следующую команду:

```
java WriteServer
```

в одном окне; это будет клиент. Затем в другом окне запустите

```
java WriteServer 1
```

Это будет сервер. Все, что вы напечатаете в окне сервера, после получения символа перевода строки будет отправлено в клиентское окно.

21 ГЛАВА

Класс Applet

В настоящей главе рассматривается класс `Applet`, который обеспечивает основу для создания аплетов. Класс `Applet` содержится в пакете `java.applet`. `Applet` включает несколько методов, которые предоставляют возможность тонкого контроля выполнения аплета. В дополнение `java.applet` также определяет три интерфейса: `AppletContext`, `AudioClip` и `AppletStub`.

Два типа аплетов

Важно подчеркнуть, что существует два варианта аплетов. Первый основан непосредственно на классе `Applet`, описанном в настоящей главе. Эти аплеты используют Abstract Window Toolkit (AWT) для предоставления пользовательского графического интерфейса (или вообще его не используют). Этот вид аплетов доступен с самого начала существования Java.

Второй тип аплетов основан на Swing-классе `JApplet`. Аплеты Swing используют классы Swing для построения графического интерфейса пользователя. Swing предлагает более богатый и часто более легкий в применении пользовательский интерфейс, чем AWT. Поэтому аплеты на основе Swing в настоящее время наиболее популярны. Однако традиционные аплеты на основе AWT все еще применяются, особенно тогда, когда требуется построить очень простой пользовательский интерфейс. А потому и аплеты на основе AWT, и аплеты на основе Swing совершенно законны.

Поскольку `JApplet` наследуется от `Applet`, все средства `Applet` также доступны в `JApplet`, и большая часть информации настоящей главы касается обоих типов аплетов. А потому, даже если вас интересуют только аплеты Swing, информация, приведенная в этой главе, все равно будет полезна и необходима. Однако следует понимать, что при создании аплетов на основе Swing существуют некоторые дополнительные ограничения, которые будут описаны далее в книге, когда пойдет речь о Swing.

На заметку! *За информацией о построении аплетов с использованием Swing обращайтесь в главу 29.*

Основы аплетов

В главе 13 была представлена общая форма аплета и шаги, необходимые для его компиляции и запуска. Начнем с пересмотра этой информации.

Все аплеты являются подклассами (прямыми или непрямыми) класса `Applet`. Аплеты не являются самостоятельными программами. Вместо этого они выполняются либо внутри Web-браузера, либо в средстве просмотра аплетов. Иллюстрации, которые приведены в этой главе, были созданы с применением стандартного средства просмотра аплетов под названием `appletviewer`, входящего в состав JDK. Однако вы можете использовать любой другой браузер или средство просмотра по своему вкусу.

Выполнение аплета не начинается с `main()`. На самом деле лишь немногие аплеты имеют методы `main()`. Вместо этого выполнение аплета запускается и управляется посредством совершенно другого механизма, который будет описан ниже. Вывод окна вашего аплета не осуществляется посредством `System.out.println()`. Вместо этого в не-Swing аплетах вывод обрабатывается различными методами AWT, такими как `drawString()`, который направляет строку в точку с указанными координатами X,Y. Ввод также обрабатывается иначе, чем в консольных приложениях. (Помните, что аплеты на основе Swing используют классы Swing для обработки взаимодействия с пользователем, и они также будут описаны далее в настоящей книге.)

Для применения аплета его необходимо специфицировать в файле HTML. Одним из способов сделать это является дескриптор `APPLET`. (Дескриптор `OBJECT` также может быть использован, но Sun в настоящее время рекомендует дескриптор `APPLET`, и именно этот дескриптор мы используем в примерах настоящей книги.) Аплет будет запущен Web-браузером с поддержкой Java, когда он встретит дескриптор `APPLET` внутри файла HTML. Чтобы просматривать и тестировать аплет более удобно, просто включите комментарий в заголовок файла вашего исходного кода Java, который содержит дескриптор `APPLET`. Таким образом, код будет документирован HTML-конструкциями, необходимыми вашему аплету, и вы сможете протестировать скомпилированный аплет, запустив средство просмотра аплетов с файлом исходного кода Java, специфицированным в качестве целевого. Вот пример такого комментария:

```
/*
<applet code="MyApplet" width=200 height=60>
</applet>
*/
```

Этот комментарий включает дескриптор `APPLET`, который запустит аплет по имени `MyApplet` в окне размером 200 пикселей в ширину и 60 пикселей в высоту. Поскольку включение команды `APPLET` облегчает тестирование аплетов, все аплеты, показанные в этой книге, будут содержать соответствующий дескриптор `APPLET`, помещенный в комментарий.

Класс `Applet`

Класс `Applet` определяет методы, показанные в табл. 21.1. `Applet` обеспечивает всю необходимую поддержку для выполнения аплета, такую как его запуск и останов. Кроме того, он предоставляет методы для загрузки и отображения графических образов, а также методы для загрузки и воспроизведения аудиоклипов. `Applet` расширяет AWT-класс `Panel`. В свою очередь, `Panel` расширяет `Container`, который расширяет `Component`. Все эти классы обеспечивают поддержку графического оконного интерфейса на базе Java.

Таким образом, Applet обеспечивает всю необходимую поддержку деятельности на основе окон. (AWT будет подробно рассматриваться в последующих главах.)

Таблица 21.1. Методы, определенные в Applet

Метод	Описание
`void destroy()`	Вызывается браузером непосредственно перед уничтожением аплета. Ваш аплет переопределяет этот метод, если нуждается в выполнении некоторых действий по очистке перед уничтожением.
`AccessibleContext getAccessibleContext()`	Возвращает контекст доступности для вызывающего объекта.
`AppletContext getAppletContext()`	Возвращает контекст, ассоциированный с аплетом.
`String getAppletInfo()`	Возвращает строку, описывающую аплет.
`AudioClip getAudioClip(URL url)`	Возвращает объект AudioClip, инкапсулирующий аудиоклип, находящийся в месте, специфицированном *url*.
`AudioClip getAudioClip(URL url, String clipName)`	Возвращает объект AudioClip, инкапсулирующий аудиоклип, находящийся в месте, специфицированном *url*, и имеющий имя, указанное в *clipName*.
`URL getCodeBase()`	Возвращает URL, ассоциированный с вызывающим аплетом.
`URL getDocumentBase()`	Возвращает URL документа HTML, вызвавшего аплет.
`Image getImage(URL url)`	Возвращает объект Image, инкапсулирующий графическое изображение, найденное в месте, указанном в *url*.
`Image getImage(URL url, String imageName)`	Возвращает объект Image, инкапсулирующий графическое изображение, найденное в месте, указанном в *url*, и имеющий имя, указанное в *imageName*.
`Locale getLocale()`	Возвращает объект Locale, используемый различными чувствительными к локали классами и методами.
`String getParameter(String paramName)`	Возвращает параметр, ассоциированный с *paramName*. Если указанный параметр не найден, возвращается null.
`String[] [] getParameterInfo()`	Возвращает таблицу String, описывающую параметры, распознаваемые аплетом. Каждое вхождение в таблице должно состоять из трех строк, содержащих имя параметра, описание его типа и/или диапазон, а также все необходимые пояснения.
`void init()`	Вызывается при запуске выполнения аплета. Это — первый метод, вызываемый аплетом.
`boolean isActive()`	Возвращает true, если аплет запущен. Возвращает false, если аплет был остановлен.
`static final AudioClip newAudioClip(URL url)`	Возвращает объект AudioClip, инкапсулирующий аудиоклип, который расположен в месте, указанном *url*. Этот метод подобен getAudioClip() во всем, за исключением того, что является статическим и может быть выполнен без объекта Applet.

Окончание табл. 21.1

Метод	Описание
`void play(URL url)`	Если аудио-клип найден в месте, указанном `url`, он выполняется.
`void play(URL url, String clipName)`	Если аудио-клип по имени `clipName` найден в месте, указанном `url`, он выполняется.
`void resize(Dimension dim)`	Изменяет размер аплета согласно измерениям, указанным в `dim`. `Dimension` — это класс, находящийся внутри `java.awt`. Он содержит два целочисленных поля: `width` и `height`.
`void resize(int width, int height)`	Изменяет размер аплета согласно измерениям, указанным в `width` и `height`.
`final void setStub(AppletStub stubObj)`	Делает `stubObj` заглушкой (stub) для аплета. Этот метод применяется исполняющей системой и никогда не вызывается непосредственно вашим аплетом. *Заглушка* — это маленький кусочек кода, который обеспечивает связь вашего аплета с браузером.
`void showStatus(String str)`	Отображает `str` в строке состояния окна браузера или средства просмотра аплетов. Если браузер не поддерживает окна состояния, то ничего не происходит.
`void start()`	Вызывается браузером, когда аплет должен начать (или возобновить) выполнение. Автоматически вызывается после `init()` в начале работы аплета.
`void stop()`	Вызывается браузером для приостановки выполнения аплета. Будучи остановленным, аплет перезапускается вызовом `start()`.

Архитектура аплетов

Аплет — это оконная программа, и потому его архитектура отличается от консольных программ, показанных в первой части нашей книги. Если вы знакомы с программированием под Windows, то при написании аплетов почувствуете себя буквально как дома. Если же нет, то вы должны понимать некоторые концепции, которые будут описаны ниже.

Во-первых, аплеты управляются событиями. Хотя мы не будем рассматривать обработку событий до следующей главы, важно понимать в общем, как управляемая событиями архитектура влияет на дизайн аплетов. Аплет включает в себя набор служебных процедур, обрабатывающих прерывания. Вот как это работает. Аплет ожидает, пока не произойдет событие. Исполняющая система извещает аплет о событии, вызывая обработчик события, предоставленный аплетом. Как только это случилось, аплет должен предпринять соответствующие действия и немедленно возвратить управление. Это — важнейший момент. Большей частью ваш аплет не должен входить в "режим" операций, в которых он будет удерживать управление в течение длительного периода. Вместо этого он должен выполнить определенные действия в ответ на события и затем вернуть управление исполняющей системе. В тех ситуациях, когда вашему аплету нужно выполнять какое-то повторяющееся действие (например, отображать в окне прокручивающееся сообщение), вы должны запустить дополнительный поток выполнения (Далее в настоящей главе вы увидите соответствующий пример.)

Во-вторых, пользователь инициирует взаимодействие с аплетом — и никакого пути в обход! Как вы знаете, в неоконных программах, когда программе необходим ввод, она выводит приглашение пользователю и затем вызывает некоторый метод ввода, например, `readLine()`. В аплетах все работает не так. Вместо этого пользователь взаимодействует с аплетом так, как он желает и когда хочет. Эти взаимодействия отсылаются аплету в виде событий, на которые тот должен отреагировать. Например, когда пользователь щелкает кнопкой мыши внутри окна аплета, генерируется событие щелчка. Если пользователь нажимает клавишу, когда окно аплета имеет фокус ввода, генерируется событие нажатия клавиши. Как вы увидите в последующих главах, аплеты могут содержать различные элементы управления, такие как экранные кнопки и флажки. Всякий раз, когда пользователь взаимодействует с одним из этих элементов управления, генерируются события.

Хотя архитектуру аплетов понять не так просто, как архитектуру консольных программ, Java облегчает это, насколько возможно. Если вы писали программы под Windows, то знаете, насколько устрашающе может выглядеть эта среда. К счастью, Java предоставляет в ваше распоряжение намного более ясный подход, которым можно овладеть гораздо быстрее.

Скелет аплета

Все аплеты, кроме только наиболее тривиальных, переопределяют методы, обеспечивающие базовые механизмы взаимодействия браузера или средства просмотра аплетов с самим аплетом и управляющие его выполнением. Четыре из этих методов — `init()`, `start()`, `stop()` и `destroy()` — применимы ко всем аплетам; они определены в классе `Applet`. Предусмотрены реализации по умолчанию всех этих методов. Однако в их переопределении не нуждаются лишь очень простые аплеты.

Аплеты на базе AWT (вроде тех, что обсуждаются в настоящей главе) также будут переопределять метод `paint()`, который определен в AWT-классе `Component`. Этот метод вызывается, когда вывод аплета должен быть заново отображен. (Аплеты на основе Swing используют другой механизм для решения этой задачи.) Эти пять методов могут быть собраны в "скелет", показанный ниже.

```
// Скелет аплета.
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/

public class AppletSkel extends Applet {
   // Вызывается первым.
      public void init() {
         // инициализация
      }

   /* Вызывается вторым, после init(). Также вызывается
      при перезапуске аплета. */
   public void start() {
      // запускает или возобновляет выполнение
   }
```

```
    // Вызывается при останове аплета.
    public void stop() {
       // приостановка выполнения
    }

    /* Вызывается перед уничтожением аплета. Это - последний
       выполняемый метод. */
    public void destroy() {
       // выполняет завершающие действия
    }

    // Вызывается, когда окно аплета должно быть восстановлено.
    public void paint(Graphics g) {
       // перерисовка содержимого окна
    }
  }
```

Хотя этот скелет ничего и не делает, его можно скомпилировать и запустить. Будучи запущенным, он генерирует показанное на рис. 21.1 окно при выводе в средстве просмотра аплетов.

Рис. 21.1. Окно, отображаемое скелетом аплета

Инициализация и прекращение работы аплета

Важно понимать порядок, в котором вызываются различные методы, показанные выше в скелете. Когда аплет стартует, вызываются по порядку следующие методы:

1. `init()`
2. `start()`
3. `paint()`

Когда работа аплета прекращается, происходит следующая последовательность вызовов:

1. `stop()`
2. `destroy()`

Рассмотрим эти методы более подробно.

init()

Метод `init()` — первый в цепочке вызовов. Именно в нем вы должны инициализировать переменные. На протяжении времени выполнения аплета этот метод вызывается лишь однажды.

start()

Метод `start()` вызывается после `init()`. Он также вызывается для перезапуска аплета после его останова. В то время как `init()` вызывается лишь однажды, при первоначальной загрузке аплета, `start()` вызывается каждый раз, когда HTML-документ, содержащий аплет, отображается на экране. Поэтому если пользователь покидает Web-страницу, а затем возвращается обратно, каждый раз аплет возобновляет свою работу в методе `start()`.

paint()

Метод `paint()` вызывается каждый раз, когда вывод аплета должен быть перерисован. Эта ситуация возникает в нескольких случаях. Например, окно, в котором выполняется аплет, может быть перекрыто другим окном, а затем вновь открыто. Или же окно аплета может быть минимизировано, а затем восстановлено. Метод `paint()` также вызывается, когда аплет начинает выполнение. В любом случае всякий раз, когда нужно перерисовать аплет, вызывается его метод `paint()`. Метод `paint()` принимает один параметр типа `Graphics`. Этот параметр будет содержать графический контекст, описывающий графическую среду, в которой выполняется аплет. Этот контекст используется всякий раз, когда запрашивается вывод аплета.

stop()

Метод `stop()` вызывается, когда Web-браузер покидает HTML-документ, содержащий аплет. Например, когда осуществляется переход на другую страницу. Когда вызывается `stop()`, аплет, возможно, работает. Вы должны использовать `stop()` для приостановки потоков, который не должен выполняться, когда аплет не видим. Вы можете перезапустить их, когда вызывается `start()`, при возврате пользователя на страницу.

destroy()

Метод `destroy()` вызывается, когда среда определяет, что ваш аплет должен быть полностью удален из памяти. В этот момент вы должны освободить все ресурсы, которые мог использовать ваш аплет. Вызову метода `destroy()` всегда предшествует вызов `stop()`.

Переопределение `update()`

В некоторых ситуациях вашему аплету может понадобиться переопределить другой метод, определенный AWT — `update()`. Этот метод вызывается, когда ваш аплет запрашивает перерисовку определенной части окна. Версия `update()` по умолчанию просто вызывает `paint()`. Однако вы можете переопределить `update()`, чтобы он выполнял более тонкую перерисовку. Вообще переопределение `update()` — это специализированная техника, которая не применяется во всех аплетах, и примеры, приведенные в настоящей книге, не переопределяют `update()`.

Простые методы отображения аплетов

Как мы уже упоминали, аплеты отображаются в окне, и аплеты на основе AWT используют AWT для выполнения ввода и вывода. Хотя мы будем подробно рассматривать методы, процедуры и приемы, используемые для полного управления оконной средой AWT, в последующих главах, все же несколько из них опишем уже здесь, поскольку они понадобятся для разработки примеров аплетов. (Помните, что аплеты на основе Swing будут рассматриваться в книге далее.)

Как было сказано в главе 13, для того, чтобы выполнить вывод строки в аплете, используется `drawstring()`, являющийся членом класса `Graphics`. Обычно этот метод вызывается внутри `update()` или `paint()`. Он имеет следующую обобщенную форму:

```
void drawString(String message, int x, int y)
```

Здесь *message* — строка, которая должна быть выведена, начиная с точки с координатами *x,y*. В окне Java левый верхний угол имеет координаты 0,0. Метод `drawString()` не распознает символы переноса строки. Если вы хотите начать вывод текста с новой строки, то должны сделать это явно, указав точные координаты X,Y точки, с которой нужно начать вывод этой новой строки текста (как вы увидите в последующих главах, существуют приемы, позволяющие облегчить этот процесс).

Чтобы установить цвет фона для окна аплета, используйте `setBackground()`. Чтобы установить цвет переднего плана (цвет вывода текста, например), используйте `setForeground()`. Эти методы определены в `Component` и имеют следующую общую форму:

```
void setBackground(Color newColor)
void setForeground(Color newColor)
```

Здесь *newColor* специфицирует новый цвет. Класс `Color` определяет перечисленные ниже константы, которые могут быть использованы для указания цвета:

```
Color.black                    Color.magenta
Color.blue                     Color.orange
Color.cyan                     Color.pink
Color.darkGray                 Color.red
Color.gray                     Color.white
Color.green                    Color.yellow
Color.lightGray
```

Также определены версии этих констант в верхнем регистре.

В следующем примере устанавливается зеленый цвет фона и красный цвет текста:

```
setBackground(Color.green);
setForeground(Color.red);
```

Подходящее место для установки цветов фона и переднего плана является метод `init()`. Конечно, во время выполнения вашего аплета вы можете изменять эти цвета так часто, как хотите.

Получить текущие установки цветов переднего плана и текста можно вызовами методов `getBackground()` и `getForeground()` соответственно. Они также определены в `Component` и показаны ниже:

```
Color getBackground()
Color getForeground()
```

Рассмотрим пример очень простого аплета, который устанавливает в качестве цвета фона циан, а цвета переднего плана — красный, и затем отображает сообщение, иллюстрирующее порядок вызова методов `init()`, `start()` и `paint()` при запуске аплета.

```
/* Простой аплет, устанавливающий цвета фона
   и переднего плана, и отображающий 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 отображаемую строку.
public void start() {
  msg += " Inside start( ) --";
}

// Отобразить msg в окне аплета.
public void paint(Graphics g) {
  msg += " Inside paint( ).";
  g.drawString(msg, 10, 30);
}
}
```

Этот аплет генерирует окно, показанное на рис. 21.2.

Рис. 21.2. Окно аплета, выводящего строку

Методы `stop()` и `destroy()` не переопределены, поскольку в таком простом аплете это не требуется.

Запрос перерисовки

В качестве главного правила запомните, что аплет пишет в свое окно только тогда, когда его методы `update()` или `paint()` вызываются AWT. Это вызывает интересный вопрос: как сам аплет может инициировать собственное обновление, когда изменяется его информация? Например, если аплет отображает двигающийся баннер, какой механизм заставит аплет обновлять окно при каждом шаге прокрутки этого баннера? Помните, что одним из наиболее важных ограничений, накладываемых на аплет, является то, что он должен быстро возвращать управление исполняющей системе. Например, он не может создавать циклы внутри `paint()`, которые будут непрерывно прокручивать баннер. Это помешало бы передаче управления обратно в AWT. Имея такое ограничение, можно подумать, что вывод в окно вашего аплета, как минимум, существенно затруднен. К счастью это не так. Всякий раз, когда ваш аплет нуждается в обновлении отображаемой в его окне информации, он просто вызывает `repaint()`.

Метод `repaint()` определен в AWT. Он заставляет исполняющую систему AWT осуществлять вызов метода `update()` вашего аплета, который в реализации по умолчанию обращается к `paint()`. Таким образом, для того, чтобы другая часть вашего аплета могла выполнять вывод в его окно, просто сохраните вывод и вызовите `repaint()`. Затем AWT выполнит вызов метода `paint()`, который может отобразить сохраненную информацию. Например, если часть вашего аплета нуждается в выводе строки, она может сохранить ее в переменной `String` и затем вызвать `repaint()`. Внутри `paint()` вы выведите строку с помощью `drawString()`.

Метод `repaint()` имеет четыре формы. Рассмотрим их по порядку. Следующая версия специфицирует область, подлежащую перерисовке:

```
void repaint(int left, int top, int width, int height)
```

Здесь координаты правого верхнего угла области указаны в *`left`* и *`top`*, а ширина и высота области — в *`width`* и *`height`*. Эти измерения указаны в пикселях. Вы экономите время, указывая область для перерисовки. Обновление окон обходится дорого в смысле затрат времени. Если вам нужно обновить только небольшую часть окна, то более эффективно будет обновить только эту область, а не всю поверхность окна.

Вызов `repaint()` — это, по сути, запрос вашего аплета на скорейшее обновление. Однако если ваша система медленна или занята, `update()` может и не вызваться немедленно. Множественные запросы на перерисовку, которые поступают за краткий период времени, могут быть слиты вместе AWT, так что `update()` вызывается лишь время от времени. Во многих ситуациях это может представлять проблему, включая вывод анимации, когда существенно время обновления. Одним из решений этой проблемы может быть использование следующих форм `repaint()`:

```
void repaint(long maxDelay)
void repaint(long maxDelay, int x, int y, int width, int height)
```

Здесь *`maxDelay`* указывает максимальное количество миллисекунд, которые могут пройти до того, как будет вызван `update()`. Однако следует иметь в виду, что если время истечет прежде, чем системе удастся вызвать `update()`, этот метод не будет вызван. Никакого возвращаемого значения или возбуждаемого исключения нет, поэтому будьте осторожны.

На заметку! *Существует возможность вывода в окно аплета методами, отличными от* `paint()` *или* `update()`*. Чтобы сделать это, такой метод должен получить графический контекст для вывода в окно. Однако для большинства исключений лучше и проще маршрутизировать вывод окна через* `paint()` *и вызвать* `repaint()` *при изменении содержимого окна.*

Простой аплет с баннером

Чтобы продемонстрировать `repaint()`, разработаем простой аплет для демонстрации баннера. Этот аплет будет прокручивать сообщение слева направо, через все окно аплета. Поскольку прокрутка окна — повторяющаяся задача, она будет выполняться в отдельном потоке, созданном аплетом при инициализации. Вот исходный текст нашего баннерного аплета:

```
/* Простой аплет баннера.
Этот аплет создает поток, прокручивающий
сообщение, содержащееся в msg, справа налево
в поле окна аплета.
*/
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.";
Thread t = null;
int state;
boolean stopFlag;
// Установить цвета и инициализировать поток.
public void init() {
  setBackground(Color.cyan);
  setForeground(Color.red);
}
// Запустить поток
public void start() {
  t = new Thread(this);
  stopFlag = false;
  t.start();
}
// Точка входя для потока, прокручивающего баннер.
public void run() {
  char ch;
  // Отобразить баннер
  for( ; ; ) {
     try {
       repaint();
       Thread.sleep(250);
       ch = msg.charAt(0);
       msg = msg.substring(1, msg.length());
       msg += ch;
       if(stopFlag)
            break;
     } catch(InterruptedException e) {}
  }
}
```

```
// Пауза в выводе баннера.
public void stop() {
  stopFlag = true;
  t = null;
}
// Отображение баннера.
public void paint(Graphics g) {
  g.drawString(msg, 50, 30);
}
}
```

Окно этого аплета будет выглядеть, как показано на рис. 21.3.

Рис. 21.3. Окно аплета с прокруткой сообщения

Давайте рассмотрим внимательней, как работает этот аплет. Для начала обратите внимание, что `SimpleBanner` расширяет `Applet`, как и следовало ожидать. Но, кроме того, он реализует `Runnable`. Это необходимо, поскольку аплет создаст второй поток выполнения, который будет использован для прокрутки баннера. Внутри `init()` устанавливаются цвета фона и переднего плана аплета.

После инициализации исполняющая система вызывает `start()` для запуска выполнения аплета. Внутри `start()` создается новый поток выполнения и присваивается переменной `t` типа `Thread`. Затем устанавливается в `false` булевская переменная `stopFlag`, которая управляет выполнением аплета. Затем поток запускается вызовом `t.start()`. Помните, что `t.start()` вызывает метод, определенный `Thread`, то есть начинает выполнять `run()`. Это не инициирует вызова версии метода `start()`, определенного в `Applet`. Это два отдельных метода.

Внутри `run()` символы строки, содержащейся в `msg`, непрерывно прокручиваются влево. Между каждым шагом прокрутки вызывается метод `repaint()`. Это в конечном итоге приводит к вызову метода `paint()`, и текущее содержимое `msg` отображается. Между итерациями `run()` засыпает на четверть секунды. В результате такой работы `run()` содержимое `msg` прокручивается слева направо в режиме непрерывного движения. Переменная `stopFlag` проверяется на каждом шаге итерации. Когда она принимает значение `true`, метод `run()` прерывается.

Если браузер отображает аплет во время просмотра новой страницы, вызывается метод `stop()`, который устанавливает `stopFlag` в `true`, прерывая тем самым выполнение `run()`. Этот механизм служит для остановки потока, когда страница более не просматривается. Когда же аплет снова оказывается в поле зрения, вновь вызывается его метод `start()`, который запускает новый поток для прокрутки баннера.

Использование строки состояния

В дополнение к отображению информации в своем собственном окне аплет может также выводить сообщение в строку состояния браузера или средства просмотра аплета, в котором он запущен. Чтобы сделать это, вызовите `showStatus()` со строкой, которую хотите отобразить. Строка состояния — хорошее место для того, чтобы дать пользователю представление о том, что происходит в аплете, показать опции или выводить сообщения о некоторых типах ошибок. Строка состояния также и отличный инструмент отладки, поскольку дает простой способ вывода информации о вашем аплете.

В следующем аплете демонстрируется применение `showStatus()`.

```
// Использование окна состояния.
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);
}

// Отобразить msg в окне аплета.
public void paint(Graphics g) {
  g.drawString("This is in the applet window.", 10, 20);
  showStatus("This is shown in the status window.");
  // g.drawString("Это в окне аплета.", 10, 20);
  // showStatus("Это в строке состояния.");
}
}
```

Окно этого аплета показано на рис. 21.4.

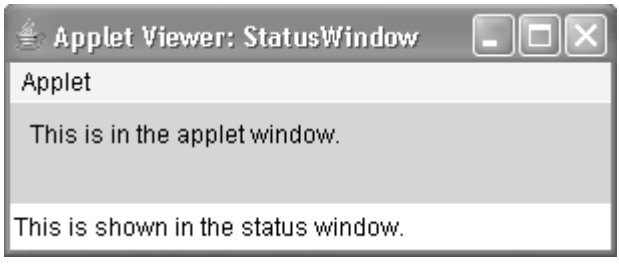

Рис. 21.4. Окно аплета, использующего строку состояния

HTML-дескриптор `APPLET`

Как уже упоминалось ранее, в последнее время Sun рекомендует использовать дескриптор `APPLET` как для запуска аплета из HTML-документа, так и в средстве просмотра аплетов. Средство просмотра аплетов выполнит каждый дескриптор `APPLET`, который он

обнаружит, в отдельном окне, в то время как Web-браузеры позволяют выполнять много аплетов на одной странице. До сих пор мы использовали только упрощенную форму дескриптора APPLET. Теперь пришло время рассмотреть его более внимательно.

Синтаксис полной формы дескриптора APPLET показан ниже. Элементы, взятые в квадратные скобки, являются необязательными.

```
<APPLET
[CODEBASE = URL_базы_кода]
CODE = файл_аплета
[ALT = альтернативный_текст]
[NAME = имя_экземпляра_аплета]
WIDTH = пикселей HEIGHT = пикселей
[ALIGN = выравнивание]
[VSPACE = пикселей] [HSPACE = пикселей]
>
[<PARAM NAME = Имя_атрибута VALUE = Значение_атрибута>]
[<PARAM NAME = Имя_атрибута2 VALUE = Значение_атрибута>]
. . .
[HTML-код, отображаемый при отсутствии Java]
</APPLET>
```

Теперь рассмотрим каждую часть по очереди.

- CODEBASE — необязательный атрибут, специфицирующий базовый URL кода аплета, который представляет собой каталог, где будет выполняться поиск исполняемого файла класса (указанного в дескрипторе CODE). Если этот атрибут не задан явно, в качестве CODEBASE используется каталог URL HTML-документа. CODEBASE не обязательно должен находиться на том же хосте, откуда прочитан HTML-документ.
- CODE — обязательный атрибут, который задает имя файла, содержащего скомпилированный файл .class вашего аплета. Имя этого файла относительно к базовому URL кода аплета, который является каталогом, где располагается HTML-файл, либо каталогом, указанным в CODEBASE.
- ALT — необязательный атрибут, используемый для указания краткого текстового сообщения, которое должно быть отображено, если браузер распознает дескриптор APPLET, но в данный момент не может выполнять Java-аплеты. Это отличается от альтернативного HTML-кода, который вы предоставляете для браузеров, вообще не поддерживающих аплеты.
- NAME — необязательный атрибут, используемый для спецификации имени экземпляра аплета. Аплеты должны именоваться таким образом, чтобы другие аплеты на той же странице могли находить их по именам и взаимодействовать с ними. Чтобы получить аплет по имени, используйте метод getApplet(), определенный в интерфейсе AppletContext.
- WIDTH и HEIGHT — обязательные атрибуты, задающие размер (в пикселях) отображаемой области аплета.
- ALIGN — необязательный атрибут, задающий выравнивание аплета. Этот атрибут трактуется точно так же, как в HTML-дескрипторе IMG, и имеет следующие значения: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE и ABSBOTTOM.
- VSPACE и HSPACE — необязательные атрибуты. VSPACE специфицирует пространство в пикселях над и под аплетом, а HSPACE — пространство в пикселях по бокам аплета. Они трактуются точно так же, как и атрибуты VSPACE и HSPACE дескриптора IMG.

- PARAM NAME и VALUE — дескриптор PARAM позволяет указывать специфичные для аплета аргументы на HTML-странице. Аплеты получают доступ к этим атрибутам посредством метода getParametr().

Среди прочих допустимых атрибутов еще можно упомянуть ARCHIVE, позволяющий специфицировать один или более архивных файлов, и OBJECT, указывающий сохраненную версию аплета. В общем случае дескриптор APPLET должен включать только атрибут CODE или OBJECT, но не оба сразу.

Передача параметров аплетам

Как только что упоминалось, HTML-дескриптор APPLET позволяет передавать параметры аплету. Для извлечения параметра служит метод getParameter(). Он возвращает значение указанного параметра в форме объекта String. Таким образом, для числовых и булевских значений вам придется преобразовывать их строковые представления во внутренние форматы. Рассмотрим пример передачи параметров.

```
// Использование параметров.
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;
// Инициализация строки для отображения.
public void start() {
  String param;
  fontName = getParameter("fontName");
  if(fontName == null)
     fontName = "Not Found";
     // fontName = "Не найден";
  param = getParameter("fontSize");
  try {
     if(param != null) // если не найден
       fontSize = Integer.parseInt(param);
     else
       fontSize = 0;
  } catch(NumberFormatException e) {
     fontSize = -1;
  }
  param = getParameter("leading");
  try {
     if(param != null) // не найден
       leading = Float.valueOf(param).floatValue();
```

```
      else
        leading = 0;
    } catch(NumberFormatException e) {
      leading = -1;
    }
    param = getParameter("accountEnabled");
    if(param != null)
      active = Boolean.valueOf(param).booleanValue();
  }

  // Отобразить параметры.
  public void paint(Graphics g) {
    g.drawString("Font name: " + fontName, 0, 10);
    g.drawString("Font size: " + fontSize, 0, 26);
    g.drawString("Leading: " + leading, 0, 42);
    g.drawString("Account Active: " + active, 0, 58);
    // g.drawString("Имя шрифта: " + fontName, 0, 10);
    // g.drawString("Размер шрифта: " + fontSize, 0, 26);
    // g.drawString("Отступ: " + leading, 0, 42);
    // g.drawString("Активный счет: " + active, 0, 58);
  }
  }
```

Пример результирующего окна этого аплета показана на рис. 21.5.

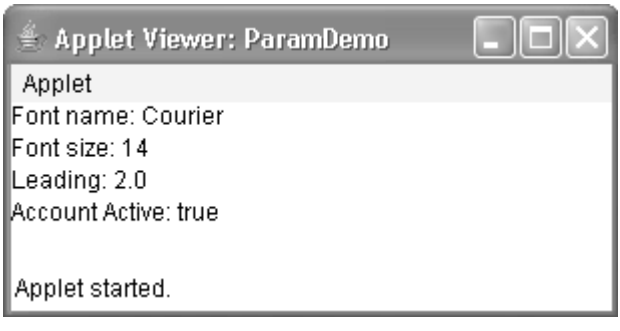

Рис. 21.5. Окно аплета, принимающего параметры

Как видно в приведенном выше коде, вы должны проверять возвращаемые значения `getParameter()`. Если параметр недоступен, `getParameter()` вернет `null`. Также должна быть предпринята попытка преобразования к числовым типам в операторе `try`, который перехватывает исключение `NumberFormatException`. Внутри аплетов никогда нельзя допускать необработанных исключений.

Усовершенствование баннерного аплета

Параметр можно использовать для усовершенствования баннерного аплета, показанного ранее. В предыдущей версии отображаемое сообщение было жестко закодировано в аплете. Передача сообщения в виде параметра позволяет баннерному аплету отображать различные сообщения при каждом выполнении. Ниже приведена усовершенствованная версия. Обратите внимание, что дескриптор `APPLET` в начале файла теперь специфицирует параметр по имени `message`, связанный со строкой в кавычках.

```
// Параметризованный баннер.
import java.awt.*;
import java.applet.*;
/*
<applet code="ParamBanner" width=300 height=50>
<param name=message value="Java заставляет Web двигаться!">
</applet>
*/

public class ParamBanner extends Applet implements Runnable {
String msg;
Thread t = null;
int state;
boolean stopFlag;

// Установить цвета и инициализировать поток.
public void init() {
  setBackground(Color.cyan);
  setForeground(Color.red);
}

// Запуск потока
public void start() {
  msg = getParameter("message");
  if(msg == null) msg = "Message not found.";
  msg = " " + msg;
  t = new Thread(this);
  stopFlag = false;
  t.start();
}

// Точка входа для потока, запускающего баннер.
public void run() {
  char ch;
  // Отобразить баннер
  for( ; ; ) {
     try {
        repaint();
        Thread.sleep(250);
        ch = msg.charAt(0);
        msg = msg.substring(1, msg.length());
        msg += ch;
        if(stopFlag)
          break;
     } catch(InterruptedException e) {}
  }
}

// Пауза в отображении баннера.
public void stop() {
  stopFlag = true;
  t = null;
}

// Отобразить баннер.
public void paint(Graphics g) {
  g.drawString(msg, 50, 30);
}
}
```

getDocumentBase() и getCodeBase()

Часто приходится создавать аплеты, которые должны явно загружать медиаинформацию и текст. Java позволяет аплетам загружать данные из каталога, содержащего HTML-файл, который запускает аплет (*база документа*), а также из каталога, из которого загружается класс аплета (*база кода*). Эти каталоги возвращаются как объекты `URL` (описанные в главе 20) из методов `getDocumentBase()` и `getCodeBase()`. Они могут быть соединены со строкой — именем файла, который вы хотите загрузить. Чтобы действительно загрузить другой файл, вы используете метод `showDocument()`, определенный в интерфейсе `AppletContext`, о котором мы поговорим в следующем разделе.

В приведенном ниже аплете иллюстрируется применение этих методов.

```
// Отображение баз документа и кода.
import java.awt.*;
import java.applet.*;
import java.net.*;
/*
<applet code="Bases" width=300 height=50>
</applet>
*/

public class Bases extends Applet{
   // Отображение базы документа и базы кода.
   public void paint(Graphics g) {
      String msg;
      URL url = getCodeBase();      // получить базу кода
      msg = "Code base: " + url.toString();
      // msg = "База кода: " + url.toString();
      g.drawString(msg, 10, 20);
      url = getDocumentBase();      // получить базу документа
      msg = "Document base: " + url.toString();
      // msg = "База документа: " + url.toString();
      g.drawString(msg, 10, 40);
  }
}
```

Примерный вывод этого аплета показан на рис. 21.6.

Рис. 21.6. Окно аплета, отображающего базы документа и кода

AppletContext и showDocument()

Одним из применений Java является использование активных изображений и анимации для обеспечения графических средств навигации в Web, которые более интересны, чем простые текстовые ссылки. Чтобы позволить вашему аплету передавать управление другому URL, вы должны использовать метод `showDocument()`, определенный в интерфейсе `AppletContext`. `AppletContext` — это интерфейс, позволяющий получать информацию от исполняющей среды аплета. Методы, определенные в `AppletContext`, перечислены в табл. 21.2. Контекст текущего выполняемого аплета получается вызовом метода `getAppletContext()`, определенного в `Applet`.

Таблица 21.2. Методы, определенные интерфейсом `AppletContext`

Метод	Описание
`Applet getApplet(String appletName)`	Возвращает аплет, специфицированный именем *appletName*, если он находится внутри контекста текущего аплета. В противном случае возвращается `null`.
`Enumeration<Applet> getApplets()`	Возвращает перечисление, содержащее все аплеты из контекста текущего аплета.
`AudioClip getAudioClip(URL url)`	Возвращает объект `AudioClip`, инкапсулирующий аудиоклип, находящийся в месте, указанном *url*.
`Image getImage(URL url)`	Возвращает объект `Image`, инкапсулирующий графическое изображение, находящееся в месте, указанном *url*.
`InputStream getStream(String key)`	Возвращает поток, связанный с *key*. Ключи привязываются к потокам с помощью метода `setStream()`. Если связанного с *key* потока не существует, возвращается `null`.
`Iterator<String> getStreamKeys()`	Возвращает итератор для ключей, ассоциированных с вызывающим объектом. Ключи привязаны к потокам. См. также `getStream()` и `setStream()`.
`void setStream(String key, InputStream strm)`	Связывает поток, специфицированный *strm*, с ключом, переданным в *key*. *key* удаляется из вызывающего объекта, если *strm* равен `null`.
`void showDocument(URL url)`	Отображает в представлении документ, расположенный по URL-адресу, переданному в *url*. Этот метод может не поддерживаться средствами просмотра аплетов.
`void showDocument(URL url, String where)`	Отображает в представлении документ, расположенный по URL-адресу, переданному в *url*. Этот метод может не поддерживаться средствами просмотра аплетов. Расположение документа указано в *where*, как описано в тексте.
`void showStatus(String str)`	Отображает *str* в окне (строке) состояния

Как только вы получаете контекст аплета, то сразу можете перенести в отображаемое представление другой документ, вызвав для этого `showDocument()`. Этот метод не имеет возвращаемого значения и не возбуждает исключений в случае неудачи, поэтому используйте его с осторожностью. Доступны два метода `showDocument()`. Метод `showDocument(URL)` отображает документ, расположенный по указанному `URL`. Метод `showDocument(URL, String)` отображает указанный документ, находящийся в указанном месте внутри окна браузера. Корректными аргументами *where* являются "_self" (отобразить в текущем фрейме), "_parent" (отобразить в родительском фрейме), "_top" (отобразить

во фрейме наивысшего уровня) и "_blank" (отобразить в новом окне браузера). Вы можете также специфицировать имя, что заставит документ отобразиться в новом окне браузера по его имени. В следующем аплете демонстрируется применение `AppletContext` и `showDocument()`. При выполнении он получает контекст текущего аплета и использует его для передачи управления файлу по имени `Test.html`. Этот файл должен располагаться в том же каталоге, что и аплет. `Test.html` может содержать любой допустимый гипертекст.

```
/* Использование контекста аплета, getCodeBase()
   и showDocument() для отображения HTML-файла.
*/
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 аплета
      try {
         ac.showDocument(new URL(url+"Test.html"));
      } catch(MalformedURLException e) {
         showStatus("URL not found"); // showStatus("URL не найден");
      }
   }
}
```

Интерфейс AudioClip

Интерфейс `AudioClip` определяет три метода: `play()` (воспроизведение клипа с начала), `stop()` (останов воспроизведения клипа) и `loop()` (непрерывное циклическое воспроизведение). После того, как вы загрузите аудиоклип с помощью `getAudioClip()`, эти методы можно применять для его воспроизведения.

Интерфейс AppletStub

Интерфейс `AppletStub` предоставляет средства, которыми аплет и браузер (или средство просмотра аплетов) взаимодействуют между собой. Обычно ваш код не должен реализовывать этот интерфейс.

Консольный вывод

Хотя вывод в окно аплета должен осуществляться методами графического интерфейса пользователя, такими как `drawString()`, аплеты также могут использовать консольный вывод — в частности, для отладочных целей. Когда в аплете вызывается метод вроде `System.out.println()`, его вывод не посылается в окно аплета. Вместо этого вывод появляется либо в сеансе консоли, из которой вы запустили средство просмотра аплета, либо в консоли Java, которая доступна в некоторых браузерах. Использование консольного вывода в целях, отличных от отладки, не рекомендуется, поскольку он нарушает принципы дизайна графического интерфейса, к которым привыкло большинство пользователей.

22 ГЛАВА

Обработка событий

Настоящая глава посвящена важнейшему аспекту Java: событиям. Обработка событий является фундаментальной для всего программирования Java, поскольку это — основа аплетов и прочих типов программ с графическим пользовательским интерфейсом (graphical user interface — GUI). Как объяснялось в главе 21, аплеты представляют собой управляемые событиями программы, использующие графический интерфейс для взаимодействия с пользователем. Более того, любая программа с графическим пользовательским интерфейсом, такая как Java-приложение, написанное для Windows, является управляемой событиями. То есть вы не можете написать такую программу без четкого представления об обработке событий. События поддерживаются множеством пакетов, включая `java.util`, `java.awt` и `java.event`.

Большинство событий, на которые будет реагировать ваша программа, генерируются, когда пользователь взаимодействует с программой на основе GUI. Именно события такого рода и рассматриваются в настоящей главе. События попадают в вашу программу множеством различных путей, каждый из которых зависит от конкретного события. Существует несколько типов событий, включая генерируемые мышью, клавиатурой и различными элементами управления GUI, такими как кнопка, линейка прокрутки или флажок.

Мы начнем эту главу с обзора механизма управления событиями Java. Затем рассмотрим основные классы и интерфейсы событий, используемые AWT, и разработаем несколько примеров, демонстрирующих основы обработки событий. Эта глава также расскажет о том, как использовать классы адаптеров, вложенные классы и анонимные вложенные классы, чтобы упростить код обработки событий. Примеры, приведенные в остальной части книги, будут часто использовать эти приемы.

На заметку! *Настоящая глава сосредоточена на событиях, имеющих отношение к программам на основе GUI. Однако иногда события также используются в целях, не имеющих прямого отношения к программам подобного рода. Тем не менее, во всех случаях применяются одни и те же приемы обработки событий.*

Два механизма обработки событий

Прежде чем приступить к обсуждению обработки событий, сделаем одно важное замечание. Способ обработки событий существенно изменился от исходной версии Java (1.0) до более современных версий, начиная с 1.1. Метод обработки событий, принятый в 1.0, все еще поддерживается, хотя и не рекомендован для применения в новых программах. К тому же многие методы, поддерживающие старую модель событий 1.0, теперь объявлены устаревшими (deprecated). Современный подход состоит в том, что события должны обрабатываться всеми программами так, как описано в настоящей книге.

Модель делегации событий

Современный подход к обработке событий основан на *модели делегации событий* (delegation event model), определяющей стандартные и согласованные механизмы для генерации и обработки событий. Его концепция проста: *источник* генерирует событие и посылает его одному или более *слушателей* (listener). В этой схеме слушатель просто ожидает до тех пор, пока не получит событие. Как только событие получено, слушатель обрабатывает его и возвращает управление. Преимущество такого дизайна в том, что логика приложения, обрабатывающая события, четко отделена от логики пользовательского интерфейса, генерирующего эти события. Элемент пользовательского интерфейса может "делегировать" обработку события отдельному фрагменту кода.

В модели делегации событий слушатели должны регистрироваться источником для того, чтобы получать извещения о событиях. Это обеспечивает важное преимущество: уведомления посылаются только тем слушателям, которые желают получать их. Это более эффективный способ обработки событий, нежели тот, что был принят в старом подходе Java 1.0. Раньше событие распространялось по всей иерархии вложенности до тех пор, пока не было обработано компонентом. Это вынуждало все компоненты получать события, которые они могли и не обрабатывать, что приводило к значительным затратам времени. Модель делегации событий исключила подобную расточительность.

На заметку! *Java также позволяет обрабатывать события без применения модели делегации. Это может быть сделано посредством расширения компонента AWT. Такая техника обсуждается в конце главы 24. Однако модель делегации событий более предпочтительна по описанным выше причинам.*

В последующих разделах мы определим события и опишем роли источников и слушателей.

События

В модели делегации *событие* — это объект, описывающий изменение состояния источника. Он может быть сгенерирован в результате взаимодействия пользователя с элементов в графическом интерфейсе. Некоторые из действий, вызывающих генерацию события, — это щелчок на экранной кнопке, ввод символа с клавиатуры, выбор элемента в списке и щелчок кнопкой мыши. Многие другие пользовательские операции также могут служить подобными примерами.

События могут также возникать и не как прямое следствие взаимодействия с пользовательским интерфейсом. Например, событие может быть сгенерировано по истечении

времени таймера, превышения счетчиком некоторого значения, программного или аппаратного сбоя либо завершения некоторой операции. Вы можете определять собственные события, отвечающие специфике вашего приложения.

Источники событий

Источник — это объект, генерирующий событие. Генерация происходит, когда некоторым образом меняется внутреннее состояние объекта. Источники могут генерировать более одного типа событий.

Источник должен регистрировать слушателей, чтобы эти слушатели получали извещение о событиях определенного рода. Каждый тип события имеет собственный метод регистрации. Вот его общая форма:

```
public void addTypeListener(TypeListener el)
```

Здесь *Type* — имя события, а *el* — ссылка на слушатель событий. Например, метод, регистрирующий слушатель событий клавиатуры, называется `addKeyListener()`, а метод, регистрирующий слушатель движения мыши — `addMouseMotionListener()`. Когда событие возникает, все зарегистрированные слушатели получают копию объекта события. Это называется *групповой рассылкой* события. Во всех случаях уведомления отправляются только тем слушателям, которые зарегистрировались на их получение.

Некоторые источники допускают регистрацию только одного слушателя. Общая форма такого метода показана ниже.

```
public void addTypeListener(TypeListener el)
   throws java.util.TooManyListenersException
```

Здесь *Type* — это имя объекта события, а *el* — ссылка на слушателя события. Когда такое событие происходит, зарегистрированный слушатель получает уведомление. Это называется *индивидуальной рассылкой* события.

Источник должен также предоставлять метод, позволяющий слушателю отменить регистрацию для определенного типа событий. Общая форма такого метода такова:

```
public void removeTypeListener(TypeListener el)
```

И снова *Type* — это имя объекта события, а *el* — ссылка на слушателя события. Например, для того, чтобы удалить слушатель клавиатуры, вы должны вызвать `removeKeyListener()`.

Метод, который добавляет или удаляет слушателей, предоставляется источником, генерирующим события. Например, класс `Component` предлагает методы для добавления и удаления слушателей клавиатуры и мыши.

Слушатели событий

Слушатель — это объект, уведомляемый о возникновении события. К нему предъявляются два основных требования. Во-первых, он должен быть зарегистрирован одним или более источниками событий, чтобы получать уведомления о событиях определенного рода. Во-вторых, он должен реализовать методы для получения и обработки таких уведомлений.

Методы, принимающие и обрабатывающие события, определены в наборе интерфейсов, находящихся в `java.awt.event`. Например, интерфейс `MouseMotionListener` определяет два метода для получения уведомлений при перетаскивании или перемещении мыши.

Любой объект может принимать и обрабатывать одно или оба этих события, если предоставляет реализацию этого интерфейса. В этой и последующих главах мы еще поговорим о многих других интерфейсах слушателей.

Классы событий

Классы, представляющие события, находятся в ядре механизма обработки событий Java. А потому дискуссия об обработке событий должна начинаться с классов событий. Важно понимать, однако, что Java определяет несколько типов событий, и что не все классы событий будут упомянуты в настоящей главе. Наиболее широко используемыми событиями являются те, что определены AWT и Swing. Здесь мы сосредоточимся на событиях AWT. (Многие из них также касаются и Swing.) Несколько специфичных для Swing событий будут описаны в главе 29, когда речь пойдет о Swing.

В корне иерархии классов событий Java лежит `EventObject`, расположенный в `java.util`. Это — суперкласс для всех событий. Его единственный конструктор выглядит следующим образом:

```
EventObject(Object src)
```

Параметр *src* здесь представляет объект, генерирующий событие.

`EventObject` содержит два метода: `getSource()` и `toString()`. Метод `getSource()` возвращает источник события. Его общая форма такова:

```
Object getSource()
```

Как можно ожидать, `toString()` возвращает строку — эквивалент события.

Класс `AWTEvent`, определенный в пакете `java.awt`, рассматривается в конце главы 24. А пока важно только знать, что все классы, о которых пойдет речь в этом разделе, являются подклассами `AWTEvent`.

Итак, подытожим:

- `EventObject` — суперкласс для всех событий;
- `AWTEvent` — суперкласс всех событий AWT, обрабатываемых моделью делегации событий.

Пакет `java.awt.event` определяет множество типов событий, генерируемых многими элементами пользовательского интерфейса. В табл. 22.1 показано несколько широко используемых классов событий и представлено краткое описание того, когда они генерируются. Часто используемые конструкторы и методы каждого класса описаны в последующих разделах.

Класс `ActionEvent`

`ActionEvent` генерируется по нажатию кнопки, двойному щелчку на элементе списка либо при выборе пункта меню. Класс `ActionEvent` определяет четыре целочисленных константы, которые могут быть использованы для идентификации любых модификаторов, ассоциированных с событием действия: `ALT_MASK`, `CTRL_MASK`, `META_MASK` и `SHIFT_MASK`. Вдобавок имеется целочисленная константа `ACTION_PERFORMED`, которая может служить для идентификации событий действия.

Таблица 22.1. Основные классы событий в `java.awt.event`

Класс события	Описание
`ActionEvent`	Генерируется при нажатии кнопки, двойном щелчке на элементе списка либо выборе пункта меню.
`AdjustmentEvent`	Генерируется при манипуляциях с линейкой прокрутки.
`ComponentEvent`	Генерируется при сокрытии компонента, его перемещении, изменении его размера либо включении видимости.
`ContainerEvent`	Генерируется при добавлении или исключении компонента из контейнера.
`FocusEvent`	Генерируется, когда компонент получает или теряет фокус клавиатурного ввода.
`InputEvent`	Абстрактный суперкласс для всех классов ввода компонентов.
`ItemEvent`	Генерируется при щелчке на флажке или элементе списка; также возникает при выборе элемента списка, отметке или снятии отметки пункта меню.
`KeyEvent`	Генерируется при получении ввода с клавиатуры.
`MouseEvent`	Генерируется при перетаскивании, перемещении, щелчках, нажатии и отпускании кнопок мыши; также возникает, когда курсор мыши входит на компонент либо покидает его.
`MouseWheelEvent`	Генерируется при прокрутке колесика мыши.
`TextEvent`	Генерируется при изменении значения текстовой области или текстового поля.
`WindowEvent`	Генерируется при активизации либо закрытии окна, а также при деактивизации, свертывании, распахивании или выходе из него.

`ActionEvent` имеет три конструктора:

```
ActionEvent(Object src, int type, String cmd)
ActionEvent(Object src, int type, String cmd, int modifiers)
ActionEvent(Object src, int type, String cmd, long when, int modifiers)
```

Здесь *`src`* — это ссылка на объект, генерирующий событие. Тип события указан в *`type`*, а его командная строка — в *`cmd`*. Аргумент *`modifiers`* указывает на нажатие модифицирующих клавиш (<ALT>, <CTRL>, <META> и/или <SHIFT>) в момент генерации события.

Вы можете получить имя команды для вызывающего объекта `ActionEvent`, используя метод `getActionCommand()`, показанный ниже:

```
String getActionCommand()
```

Например, когда кнопка нажата, генерируется событие действия, которое имеет имя команды, соответствующее метке экранной кнопки.

Метод `getModifiers()` возвращает значение, указывающее на то, какие модифицирующие клавиши (<ALT>, <CTRL>, <META> и/или <SHIFT>) были нажаты в момент генерации события. Его форма такова:

```
int getModifiers()
```

Метод `getWhen()` возвращает время совершения события. Это называется *временной меткой* события. Метод `getWhen()` выглядит следующим образом:

```
long getWhen()
```

Класс AdjustmentEvent

Событие AdjustmentEvent генерируется линейкой прокрутки. Существуют пять типов событий настройки (adjustment). Класс AdjustmentEvent определяет целочисленные константы, которые могут использоваться для идентификации. Константы и их описания представлены в табл. 22.2.

Таблица 22.2. Константы, определенные классом AdjustmentEvent

Константа	Описание
BLOCK_DECREMENT	Пользователь щелкнул внутри линейки прокрутки для уменьшения ее значения.
BLOCK_INCREMENT	Пользователь щелкнул внутри линейки прокрутки для увеличения ее значения.
TRACK	Передвинут бегунок линейки.
UNIT_DECREMENT	Был выполнен щелчок на кнопке в конце линейки для уменьшения ее значения.
UNIT_INCREMENT	Был выполнен щелчок на кнопке в конце линейки для увеличения ее значения.

В дополнение к ним имеется также целочисленная константа ADJUSTMENT_VALUE_CHANGED, которая указывает на то, что произошло изменение.

Вот единственный конструктор AdjustmentEvent:

```
AdjustmentEvent(Adjustable src, int id, int type, int data)
```

Здесь *src* — ссылка на объект, сгенерировавший событие. Параметр *id* специфицирует событие. Тип настройки указан в *type*, а ассоциированные с ней данные — в *data*.

Метод getAdjustable() возвращает объект, сгенерировавший событие. Его форма представлена ниже:

```
Adjustable getAdjustable()
```

Тип события настройки может быть получен методом getAdjustmentType(). Он возвращает одну из констант, определенных в AdjustmentEvent. Его общая форма такова:

```
int getAdjustmentType()
```

Величина настройки может быть получена методом getValue(), показанным ниже:

```
int getValue()
```

Например, когда выполняются манипуляции с линейкой прокрутки, этот метод возвращает значение, представленное изменением.

Класс ComponentEvent

ComponentEvent генерируется при изменении размера, положения или видимости компонента. Существуют четыре типа событий компонентов. Для их идентификации класс ComponentEvent определяет целочисленные константы. Константы и их описания представлены в табл. 22.3.

ComponentEvent имеет следующий конструктор:

```
ComponentEvent(Component src, int type)
```

Здесь *src* — ссылка на объект, генерирующий событие. Тип события указан в *type*.

Таблица 22.3. Константы, определенные классом `ComponentEvent`

Константа	Описание
`COMPONENT_HIDDEN`	Компонент скрыт.
`COMPONENT_MOVED`	Компонент перемещен.
`COMPONENT_RESIZED`	Изменен размер компонента.
`COMPONENT_SHOWN`	Компонент стал видимым.

`ComponentEvent` — это прямой или непрямой суперкласс для `ContainerEvent`, `FocusEvent`, `KeyEvent`, `MouseEvent` и `WindowEvent`.

Метод `getComponent()` возвращает компонент, генерирующий событие. Выглядит он так:

```
Component getComponent()
```

Класс `ContainerEvent`

`ContainerEvent` генерируется при добавлении компонента в контейнер или его удаления оттуда. Существуют два типа событий контейнера. Класс `ContainerEvent` определяет `int` константы, которые могут использоваться для его идентификации: `COMPONENT_ADDED` и `COMPONENT_REMOVED`. Они определяют, добавлен ли компонент к контейнеру или же исключен из него.

`ContainerEvent` — подкласс `ComponentEvent`, имеющий следующий конструктор:

```
ContainerEvent(Component src, int type, Component comp)
```

Здесь *src* — ссылка на контейнер, который сгенерировал событие. Тип события указан в *type*, а компонент, добавляемый в контейнер либо удаляемый из него — в *comp*.

Вы можете получить ссылку на контейнер, сгенерировавший это событие, из метода `getContainer()`, показанного ниже:

```
Container getContainer()
```

Метод `getChild()` возвращает ссылку на компонент, который был добавлен или удален из контейнера. Его общая форма такова:

```
Component getChild()
```

Класс `FocusEvent`

Класс `FocusEvent` генерируется, когда компонент получает или теряет фокус. Эти события определяются целочисленными константами `FOCUS_GAINED` и `FOCUS_LOST`.

`FocusEvent` — подкласс `ComponentEvent`, имеющий следующие конструкторы:

```
FocusEvent(Component src, int type)
FocusEvent(Component src, int type, boolean temporaryFlag)
FocusEvent(Component src, int type, boolean temporaryFlag, Component other)
```

Здесь *src* — ссылка на компонент, сгенерировавший событие. Тип события указан в *type*. Аргумент *temporaryFlag* установлен в `true`, если событие фокуса является временным. В противном случае он устанавливается в `false`. (Событие временного фокуса генерируется в результате другой операции пользовательского интерфейса. Например,

предположим, что фокус находится на текстовом поле. Если пользователь перемещает мышь, чтобы подвинуть линейку прокрутки, то фокус временно теряется.)

Другой компонент, участвующий в изменении фокуса и называемый *противоположным компонентом*, передается в *other*. Таким образом, если происходит событие FOCUS_GAINED, то *other* будет ссылаться на компонент, теряющий фокус. В противоположность этому, если происходит событие FOCUS_LOST, то *other* ссылается на компонент, получающий фокус.

Вы можете определить другой компоненты вызовом getOppositeComponent(), показанным ниже:

```
Component getOppositeComponent()
```

Метод возвращает противоположный компонент.

Метод isTemporary() указывает на то, является ли изменение фокуса временным. Его форма такова:

```
boolean isTemporary()
```

Метод возвращает true, если изменение временно. В противном случае он возвращает false.

Класс InputEvent

Абстрактный класс InputEvent является подклассом ComponentEvent и суперклассом для событий ввода компонента. Его подклассы — KeyEvent и MouseEvent.

InputEvent определяет несколько строковых констант, представляющих любые модификаторы, такие как признак нажатия управляющих клавиш, которые могут быть ассоциированы с событием. Изначально класс InputEvent определяет следующие восемь значений для представления модификаторов:

ALT_MASK	BUTTON2_MASK	META_MASK
ALT_GRAPH_MASK	BUTTON3_MASK	SHIFT_MASK
BUTTON1_MASK	CTRL_MASK	

Однако, поскольку существует возможность конфликтов в используемых модификаторах между клавиатурными событиями, событиями мыши, а также другие проблемы, были добавлены следующие расширенные значения модификаторов:

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	

При написании нового кода рекомендуется использовать новые расширенные модификаторы вместо исходных.

Чтобы протестировать, какая клавиша модификатора была нажата в момент генерации события, используйте методы isAltDown(), isAltGraphDown(), isControlDown(), isMetaDown() и isShiftDown(). Формы этих методов показаны ниже.

```
boolean isAltDown()
boolean isAltGraphDown()
boolean isControlDown()
boolean isMetaDown()
boolean isShiftDown()
```

Вы можете получить значение, содержащее все флаги исходных модификаторов, вызовом метода `getModifiers()`. Он выглядит так:

```
int getModifiers()
```

Расширенные модификаторы можно получить методом `getModifiersEx()`, показанным ниже:

```
int getModifiersEx()
```

Класс `ItemEvent`

`ItemEvent` генерируется при щелчке на флажке или элементе списка либо при выборе отмечаемого пункта меню. (Флажки и окна списков будут описаны далее в нашей книге.) Существуют два типа событий элементов, которые идентифицируются целочисленными константами, описанными в табл. 22.4.

Таблица 22.4. Константы, определенные классом `ItemEvent`

Константа	Описание
`DESELECTED`	Пользователь отменил выбор элемента.
`SELECTED`	Пользователь выбрал элемент.

Вдобавок `ItemEvent` определяет одну целочисленную константу `ITEM_STATE_CHANGED`, которая сигнализирует об изменении состояния.

`ItemEvent` имеет следующий конструктор:

```
ItemEvent(ItemSelectable src, int type, Object entry, int state)
```

Здесь *src* — ссылка на компонент, сгенерировавший событие. Например, это может быть список или элемент выбора. Тип события указывается в *type*. Конкретный элемент, сгенерировавший событие элемента, передается в *entry*. Текущее состояние элемента содержится в *state*.

Метод `getItem()` может быть использован для получения ссылки на элемент, сгенерировавший событие. Его сигнатура выглядит так:

```
Object getItem()
```

Метод `getItemSelectable()` может быть использован для получения ссылки на объект `ItemSelectable`, сгенерировавший событие. Его общая форма показана ниже.

```
ItemSelectable getItemSelectable()
```

Списки и выборы (флажки) являются примерами элементов пользовательского интерфейса, реализующих интерфейс `ItemSelectable`.

Метод `getStateChanged()` возвращает измененное состояние (то есть `SELECTED` или `DESELECTED`) для события. Выглядит он так:

```
int getStateChanged()
```

Класс `KeyEvent`

`KeyEvent` генерируется при клавиатурном вводе. Существуют три типа клавиатурных событий, идентифицируемые целочисленными константами: `KEY_PRESSED`,

KEY_RELEASED и KEY_TYPED. Первые два типа событий генерируются при нажатии и отпускании клавиши. Последний же тип происходит при генерации символа. Помните, что не все клавиши дают в результате символ. Например, нажатие <SHIFT> не генерирует никакого символа.

Класс KeyEvent определяет множество других целочисленных констант. Например, от VK_0 до VK_9 и от VK_A до VK_Z определяют ASCII-эквиваленты чисел и букв. А вот еще несколько других констант:

```
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 VK специфицируют *коды виртуальных клавиш*, не зависящие от любых модификаторов вроде <CONTROL>, <ALT> или <SHIFT>.

Класс KeyEvent является подклассом InputEvent. Вот один из его конструкторов:

```
KeyEvent(Component src, int type, long when, int modifiers, int code, char ch)
```

Здесь *src* — ссылка на компонент, генерирующий событие. Тип события указывается в *type*. Системное время, когда была нажата клавиша, передается в *when*. Аргумент *modifiers* указывает, какие модификаторы были нажаты в момент возникновения события клавиши. Код виртуальной клавиши, такой как VK_UP, VK_A и так далее, передается в *code*. Символьный эквивалент (если есть) передается в *ch*. Если никакого корректного символа событие не содержит, то *ch* равно CHAR_UNDEFINED. Для событий KEY_TYPED параметр *code* будет содержать VK_UNDEFINED.

Класс KeyEvent определяет несколько методов, но наиболее часто используемые из них — это getKeyChar(), возвращающий введенный символ, и getKeyCode(), возвращающий код ключа. Их общая форма представлена ниже:

```
char getKeyChar()
int getKeyCode()
```

Если никаких корректных символов нажатие клавиши не генерирует, то getKeyChar() возвращает CHAR_UNDEFINED. Когда возникает событие KEY_TYPED, то при этом getKeyCode() возвращает VK_UNDEFINED.

Класс MouseEvent

Существуют восемь типов событий мыши. Класс MouseEvent определяет для их идентификации набор целочисленных констант, которые описаны в табл. 22.5.

Таблица 22.5. Константы, определенные классом MouseEvent

Константа	Описание
MOUSE_CLICKED	Пользователь щелкнул мышью.
MOUSE_DRAGGED	Пользователь перетащил мышь.
MOUSE_ENTERED	Курсор мыши вошел в компонент.
MOUSE_EXITED	Курсор мыши покинул компонент.
MOUSE_MOVED	Мышь перемещена.
MOUSE_PRESSED	Нажата кнопка мыши.
MOUSE_RELEASED	Отпущена кнопка мыши.
MOUSE_WHEEL	Выполнена прокрутка колесика мыши.

MouseEvent — это подкласс InputEvent. Вот один из его конструкторов:

```
MouseEvent(Component src, int type, long when, int modifiers,
       int x, int y, int clicks, boolean triggersPopup)
```

Здесь *src* — ссылка на компонент, генерирующий событие. Тип события указан в *type*. Системное время, когда произошло событие, передается в *when*. Аргумент *modifiers* указывает, какие клавиши модификаторов были нажаты в момент возникновения события мыши. Координаты курсора мыши передаются в *x* и *y*. Счетчик щелчков передается в *clicks*. Флаг *triggersPopup* указывает на то, должно ли данное событие вызывать всплывающее меню на данной платформе.

Два часто используемых метода этого класса — это getX() и getY(). Они возвращают координаты X и Y курсора мыши на момент возникновения события. Их форма такова:

```
int getX()
int getY()
```

В качестве альтернативы для получения координат курсора мыши вы можете использовать метод getPoint(). Он показан ниже:

```
Point getPoint()
```

Этот метод возвращает объект Point, содержащий координаты X, Y в своих целочисленных членах x и y.

Метод translatePoint() изменяет местоположение события. Его форма показана ниже:

```
void translatePoint(int x, int y)
```

Здесь аргументы *x* и *y* добавляются к первоначальным координатам события.

Метод getClickCount() получает количество щелчков мыши для данного события. Его сигнатура показана ниже:

```
int getClickCount()
```

Метод isPopupTrigger() проверяет, вызывает ли данное событие всплывающее меню на данной платформе. Его форма такова:

```
boolean isPopupTrigger()
```

Также доступен метод getButton(), показанный ниже:

```
int getButton()
```

Он возвращает значение, представляющее кнопку, вызвавшую событие. Возвращаемое значение будет одной из констант, определенных в MouseEvent:

- NOBUTTON
- BUTTON1
- BUTTON2
- BUTTON3

Значение NOBUTTON указывает на то, что никакая кнопка не была нажата или отпущена.

В Java SE 6 были добавлены три метода к MouseEvent, которые получают координаты мыши относительно экрана, а не компонента. Они показаны ниже:

```
Point getLocationOnScreen()
int getXOnScreen()
int getYOnScreen()
```

Метод `getLocationOnScreen()` возвращает объект `Point`, содержащий обе координаты — X и Y. Другие два метода возвращают по одной координате соответственно.

Класс `MouseWheelEvent`

Класс `MouseWheelEvent` инкапсулирует событие колесика мыши. Он является подклассом `MouseEvent`. Не все мыши оснащены колесиками. Если у мыши есть колесико, оно располагается между левой и правой кнопками. Эти колесики используются для прокрутки (изображения, текста, таблиц и тому подобного). `MouseWheelEvent` определяет две целочисленных константы, описанные в табл. 22.6.

Таблица 22.6. Константы, определенные классом `MouseWheelEvent`

Константа	Описание
`WHEEL_BLOCK_SCROLL`	Возникло событие прокрутки на страницу вверх или вниз.
`WHEEL_UNIT_SCROLL`	Возникло событие прокрутки на строку вверх или вниз.

Вот один из конструкторов, определенных в `MouseWheelEvent`:

```
MouseWheelEvent(Component src, int type, long when, int modifiers,
                int x, int y, int clicks, boolean triggersPopup,
                int scrollHow, int amount, int count)
```

Здесь *`src`* — ссылка на компонент, генерирующий событие. Тип события указан в *`type`*. Системное время, когда произошло событие, передается в *`when`*. Аргумент *`modifiers`* указывает, какие клавиши модификаторов были нажаты в момент возникновения события мыши. Координаты курсора мыши передаются в *`x`* и *`y`*. Счетчик щелчков передается в *`clicks`*. Флаг *`triggersPopup`* указывает на то, должно ли данное событие вызывать всплывающее меню на данной платформе. Значение *`scrollHow`* должно быть либо `WHEEL_UNIT_SCROLL`, либо `WHEEL_BLOCK_SCROLL`. Количество единиц прокрутки передается в *`amount`*. Параметр *`count`* указывает количество единиц прокрутки, на которое повернуто колесико.

Класс `MouseWheelEvent` определяет методы, дающие вам доступ к событию колесика. Чтобы получить количество единиц прокрутки, вызывайте метод `getWheelRotation()`, показанный ниже:

```
int getWheelRotation()
```

Он возвращает количество единиц прокрутки. Если значение положительно, значит, колесико повернуто против часовой стрелки, если же отрицательно — то по часовой.

Чтобы получить тип прокрутки, вызывается метод `getScrollType()`:

```
int getScrollType()
```

Он возвращает либо `WHEEL_UNIT_SCROLL`, либо `WHEEL_BLOCK_SCROLL`. Если типом прокрутки является `WHEEL_UNIT_SCROLL`, то вы получите количество единиц прокрутки вызовом `getScrollAmount()`. Этот метод выглядит следующим образом:

```
int getScrollAmount()
```

Класс TextEvent

Экземпляры этого класса описывают текстовые события. Они генерируются текстовыми полями и областями, когда в них осуществляется ввод — пользователем или программой. TextEvent определяет целочисленную константу TEXT_VALUE_CHANGED.

Единственный конструктор этого класса показан ниже:

```
TextEvent(Object src, int type)
```

Здесь *src* — ссылка на объект, генерирующий событие. Тип события указывается в *type*.

Объект TextEvent не включает символов, находящихся в данный момент в текстовом компоненте, сгенерировавшем событие. Вместо этого ваша программа должна использовать другие методы, ассоциированные с текстовым компонентом, для извлечения информации. Эта операция отличается от других объектов событий, о которых речь пойдет в следующем разделе. По этой причине никакие методы класса TextEvent мы пока не обсуждаем. Об уведомлениях текстового события следует думать, как о сигнале слушателю о том, что последний должен извлечь информацию из указанного текстового компонента.

Класс WindowEvent

Существуют десять типов событий окон. Класс WindowEvent определяет целочисленные константы, которые могут использоваться для их идентификации. Эти константы вместе с их значениями перечислены в табл. 22.7.

Таблица 22.7. Константы, определенные классом WindowEvent

Константа	Описание
WINDOW_ACTIVATED	Окно было активизировано.
WINDOW_CLOSED	Окно было закрыто.
WINDOW_CLOSING	Пользователь запросил закрытие окна.
WINDOW_DEACTIVATED	Окно деактивизировано.
WINDOW_DEICONIFIED	Окно развернуто.
WINDOW_GAINED_FOCUS	Окно получило фокус ввода.
WINDOW_ICONIFIED	Окно свернуто.
WINDOW_LOST_FOCUS	Окно утратило фокус ввода.
WINDOW_OPENED	Окно было открыто.
WINDOW_STATE_CHANGED	Состояние окна изменилось.

WindowEvent — это подкласс ComponentEvent. Он определяет несколько конструкторов. Первый из них:

```
WindowEvent(Window src, int type)
```

Здесь *src* — ссылка на объект, сгенерировавший событие. Тип события передается в *type*. Следующие три конструктора предоставляют более тонкий контроль:

```
WindowEvent(Window src, int type, Window other)
WindowEvent(Window src, int type, int fromState, int toState)
WindowEvent(Window src, int type, Window other, int fromState, int toState)
```

Здесь *other* специфицирует противоположное окно при возникновении событий фокуса или активизации. Параметр *fromFocus* специфицирует предыдущее состояние окна, а *toState* — новое состояние, которое окно получает при смене состояния.

Часто используемый метод этого класса — `getWindow()`. Он возвращает объект `Window`, сгенерировавший событие. Вот его общая форма:

```
Window getWindow()
```

`WindowEvent` также определяет методы, возвращающие противоположное окно (при событиях фокуса или активизации), предыдущее состояние окна, а также его текущее состояние. Эти методы показаны ниже:

```
Window getOppositeWindow()
int getOldState()
int getNewState()
```

Источники событий

В табл. 22.8 перечислены некоторые компоненты пользовательского интерфейса, которые могут генерировать события, описанные в предыдущем разделе. В дополнение к элементам графического пользовательского интерфейса, любой класс, унаследованный от `Component`, такой как `Applet`, может генерировать события.

Таблица 22.8. Примеры источников событий

Источник событий	Описание
Кнопка	Генерирует события действия при нажатии кнопки.
Флажок	Генерирует события элемента при отметке и снятии отметки с флажка.
Переключатель	Генерирует события элемента при изменении выбора.
Список	Генерирует события действия при двойном щелчке на элементе; генерирует события элемента при выделении или снятии выделения с элемента.
Пункт меню	Генерирует события действия при выборе пункта меню; генерирует события элемента при выделении и снятии выделения помечаемого пункта меню.
Линейка прокрутки	Генерирует события настройки при манипуляциях с линейкой прокрутки.
Текстовые компоненты	Генерирует текстовые события, когда пользователь вводит символ.
Окно	Генерирует события окна при активизации, закрытии, деактивизации, развертывании, свертывании, открытии или выходе из окна.

Например, вы можете получать события клавиатуры и мыши от аплета. (Также вы можете строить свои собственные компоненты, которые генерируют события.) В этой главе мы будем иметь дело только с событиями клавиатуры и мыши, но в следующих главах пойдет речь об обработке событий от источников, перечисленных в таблице 22.8.

Интерфейсы слушателей событий

Как уже объяснялось, модель делегации событий состоит из двух частей: источников и слушателей. Слушатели создаются посредством реализации одного или более интерфейсов, определенных в пакете `java.awt.event`. Когда возникает событие, источник со-

бытия вызывает соответствующий метод, определенный в слушателе, и передает объект события в качестве аргумента. В табл. 22.9 перечислены часто используемые интерфейсы слушателей и представлено краткое описание методов, определяемых ими. В следующих разделах рассматриваются специфические методы, содержащиеся в каждом интерфейсе.

Интерфейс `ActionListener`

Этот интерфейс определяет метод `actionPerformed()`, вызываемый при возникновении события действия. Его общая форма показана ниже:

```
void actionPerformed(ActionEvent ae)
```

Таблица 22.9. Часто используемые интерфейсы слушателей событий

Интерфейс	Описание
`ActionListener`	Определяет один метод для принятия событий действия.
`AdjustmentListener`	Определяет один метод для принятия событий настройки.
`ComponentListener`	Определяет четыре метода для определения сокрытия, перемещения, изменения размера или отображения компонента.
`ContainerListener`	Определяет два метода для определения того, когда компонент добавляется в контейнер либо исключается из него.
`FocusListener`	Определяет два метода для определения получения или утраты компонентом фокуса клавиатурного ввода.
`ItemListener`	Определяет один метод, определяющий момент изменения состояния элемента.
`KeyListener`	Определяет три метода, распознающих нажатие, отпускание клавиши либо ввод символа.
`MouseListener`	Определяет пять методов, распознающих щелчок мыши, момент входа курсора мыши на поле компонента, его выхода за пределы компонента, нажатия и отпускания кнопок мыши.
`MouseMotionListener`	Определяет два метода для распознавания перетаскивания или движения мыши.
`MouseWheelListener`	Определяет один метод, распознающий движение колесика мыши.
`TextListener`	Определяет один метод, распознающий изменение текстового значения.
`WindowFocusListener`	Определяет два метода для распознавания получения или утраты окном фокуса ввода.
`WindowListener`	Определяет семь методов, распознающих активизацию окна, закрытие, деактивизацию, развертывание, свертывание, открытие или выход.

Интерфейс `AdjustmentListener`

Этот интерфейс определяет метод `adjustmentValueChanged()`, вызываемый при возникновении события настройки. Его общая форма такова:

```
void adjustmentValueChanged(AdjustmentEvent ae)
```

Интерфейс ComponentListener

Этот интерфейс определяет четыре метода, вызываемых при изменении размера компонента, его перемещении, отображении или сокрытии. Их общая форма представлена ниже:

```
void componentResized(ComponentEvent ce)
void componentMoved(ComponentEvent ce)
void componentShown(ComponentEvent ce)
void componentHidden(ComponentEvent ce)
```

Интерфейс ContainerListener

Этот интерфейс содержит два метода. Когда компонент добавляется к контейнеру, вызывается `componentAdded()`. Когда же компонент удаляется из контейнера, вызывается `componentRemoved()`. Их общие формы выглядят следующим образом:

```
void componentAdded(ContainerEvent ce)
void componentRemoved(ContainerEvent ce)
```

Интерфейс FocusListener

Этот интерфейс определяет два метода. Когда компонент получает фокус клавиатурного ввода, вызывается `focusGained()`. Когда компонент теряет фокус ввода, вызывается `focusLost()`. Обобщенная форма этих методов показана ниже:

```
void focusGained(FocusEvent fe)
void focusLost(FocusEvent fe)
```

Интерфейс ItemListener

Этот интерфейс определяет метод `itemStateChanged()`, вызываемый при изменении состояния элемента. Его общая форма такова:

```
void itemStateChanged(ItemEvent ie)
```

Интерфейс KeyListener

Этот интерфейс определяет три метода. Методы `keyPressed()` и `keyReleased()` вызываются, соответственно, при нажатии и отпускании клавиш. Метод `keyTyped()` вызывается при вводе символа.

Например, если пользователь нажимает и отпускает клавишу A, генерируют последовательно три события: нажатие клавиши, ввод символа, отпускание клавиши. Если пользователь нажимает и отпускает клавишу <HOME>, генерируются последовательно только два события: нажатие клавиши и ее отпускание.

Общие формы этих методов выглядят следующим образом:

```
void keyPressed(KeyEvent ke)
void keyReleased(KeyEvent ke)
void keyTyped(KeyEvent ke)
```

Интерфейс MouseListener

Этот интерфейс определяет пять методов. Если кнопка мыши нажата и отпущена в одной и той же точке, вызывается `mouseClicked()`. Когда курсор мыши входит на поле

компонента, вызывается метод mouseEntered(). Когда же он покидает поле компонента, вызывается метод mouseExited(). Методы mousePressed() и mouseReleased() вызываются, соответственно, когда кнопка мыши нажимается и отпускается. Общие формы этих методов перечислены ниже:

```
void mouseClicked(MouseEvent me)
void mouseEntered(MouseEvent me)
void mouseExited(MouseEvent me)
void mousePressed(MouseEvent me)
void mouseReleased(MouseEvent me)
```

Интерфейс MouseMotionListener

Этот интерфейс определяет два метода. Метод mouseDragged() вызывается множество раз при выполнении мышью перетаскивания. Метод mouseMoved() вызывается множество раз при перемещении мыши. Их общая форма такова:

```
void mouseDragged(MouseEvent me)
void mouseMoved(MouseEvent me)
```

Интерфейс MouseWheelListener

Этот интерфейс определяет метод mouseWheelMoved(), вызываемый при прокрутке колесика мыши. Его общая форма показана ниже:

```
void mouseWheelMoved(MouseWheelEvent mwe)
```

Интерфейс TextListener

Этот интерфейс определяет метод textChanged(), вызываемый при изменении содержимого текстовой области или текстового поля. Его общая форма выглядит следующим образом:

```
void textChanged(TextEvent te)
```

Интерфейс WindowFocusListener

Этот интерфейс определяет два метода: windowGainedFocus() и windowLostFocus(). Они вызываются, когда окно получает и утрачивает фокус ввода. Их общая форма показана ниже:

```
void windowGainedFocus(WindowEvent we)
void windowLostFocus(WindowEvent we)
```

Интерфейс WindowListener

Этот интерфейс определяет семь методов. Методы windowActivated() и windowDeactivated() вызываются, когда окно, соответственно, активизируется и деактивизируется.

Если окно сворачивается в пиктограмму, вызывается метод windowIconified(). Когда окно разворачивается, вызывается метод windowDeIconified(). Когда окно открывается и закрывается, вызываются методы windowOpened() и windowClosed(). Метод windowClosing() вызывается при закрытии окна. Общие формы этих методов выглядят следующим образом:

```
void windowActivated(WindowEvent we)
void windowClosed(WindowEvent we)
void windowClosing(WindowEvent we)
void windowDeactivated(WindowEvent we)
void windowDeiconified(WindowEvent we)
void windowIconified(WindowEvent we)
void windowOpened(WindowEvent we)
```

Использование модели делегации событий

Теперь, когда вы ознакомились с теорией, лежащей в основе модели делегации событий, и получили представление о различных ее компонентах, наступило время обратиться к практике. Использовать модель делегации событий довольно легко. Нужно просто выполнить следующие два шага.

1. Реализовать соответствующий интерфейс в слушателе, чтобы он мог принимать события нужного типа.
2. Реализовать код регистрации и отмены регистрации (при необходимости) слушателя как получателя уведомлений о событии.

Следует помнить, что источник может генерировать несколько типов 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 увидеть, как модель делегации событий работает на практике, мы разберем примеры, обрабатывающие два часто используемых генератора событий — событий мыши и клавиатуры.

Обработка событий мыши

Чтобы обработать события мыши, вы должны реализовать интерфейсы `MouseListener` и `MouseMotionListener`. (Вы можете также решить реализовать `MouseWheelListener`, но мы не станем делать этого здесь.) Следующий аплет демонстрирует весь процесс. Он отображает текущие координаты мыши в строке состояния. Всякий раз, когда нажимается кнопка, отображается слово “Down” в точке, где находится курсор мыши. При каждом отпускании кнопки отображается слово “Up”. При щелчке по кнопке отображается сообщение “Mouse clicked” в левом верхнем углу отображаемой области аплета.

При входе и выходе курсора мыши на поле окна аплета в левом верхнем углу отображаемой области аплета также выводится соответствующее сообщение. При выполнении мышью операции перетаскивания отображается символ “*”, который сопровождает курсор мыши при перетаскивании. Обратите внимание, что две переменных — `mouseX` и `mouseY` — сохраняют местоположение мыши, когда возникают события нажатия, отпускания кнопки, а также события перетаскивания. Эти координаты затем используются `paint()` для отображения вывода в точке возникновения события.

```
// Использование обработчиков событий 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);
}

// Обработка щелчка мыши.
public void mouseClicked(MouseEvent me) {
  // сохранить координаты
  mouseX = 0;
  mouseY = 10;
  msg = "Mouse clicked.";
  // msg = "Щелчок кнопкой мыши.";
  repaint();
}

// Обработка входа мышиного курсора.
public void mouseEntered(MouseEvent me) {
  // сохранить координаты
  mouseX = 0;
  mouseY = 10;
  msg = "Mouse entered.";
  // msg = "Курсор мыши вошел.";
  repaint();
}

// Обработка выхода мышиного курсора.
public void mouseExited(MouseEvent me) {
  // сохранить координаты
  mouseX = 0;
  mouseY = 10;
  msg = "Mouse exited.";
  // msg = "Курсор мыши вышел.";
  repaint();
}

// Обработка нажатия кнопки.
public void mousePressed(MouseEvent me) {
  // сохранить координаты
  mouseX = me.getX();
  mouseY = me.getY();
  msg = "Down";
  // msg = "Вниз";
  repaint();
}

// Обработка отпускания кнопки.
public void mouseReleased(MouseEvent me) {
  // сохранить координаты
  mouseX = me.getX();
  mouseY = me.getY();
  msg = "Up";
  // msg = "Вверх";
  repaint();
}
```

```
// Обработка перетаскивания мыши.
public void mouseDragged(MouseEvent me) {
  // сохранить координаты
  mouseX = me.getX();
  mouseY = me.getY();
  msg = "*";
  showStatus("Dragging mouse at " + mouseX + ", " + mouseY);
  // showStatus("Перетаскивание мыши в " + mouseX + ", " + mouseY);
  repaint();
}
// Обработка движения мыши.
public void mouseMoved(MouseEvent me) {
  // Показать состояние
  showStatus("Moving mouse at " + me.getX() + ", " + me.getY());
  // showStatus("Перемещение мыши в " + me.getX() + ", " + me.getY());
}
// Отобразить msg в окне аплета в текущей позиции X,Y.
public void paint(Graphics g) {
  g.drawString(msg, mouseX, mouseY);
}
}
```

Пример работы этого аплета показан на рис. 22.1.

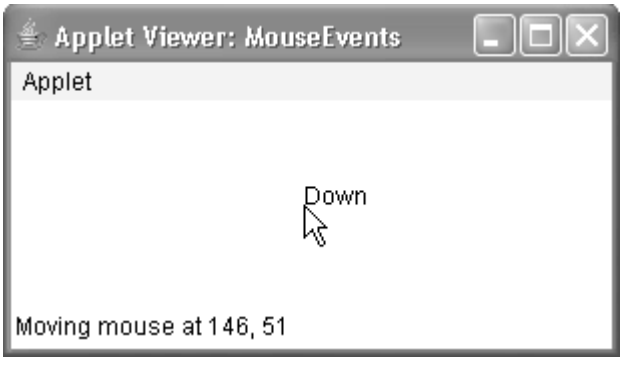

Рис. 22.1. Пример работы аплета, использующего обработчики событий мыши

Давайте рассмотрим этот пример внимательней. Класс `MouseEvents` расширяет `Applet` и реализует интерфейсы `MouseListener` и `MouseMotionListener`. Эти два интерфейса содержат методы, принимающие и обрабатывающие различные типы событий мыши. Обратите внимание, что аплет одновременно является и источником, и слушателем этих событий. Это работает благодаря тому, что `Component`, применяющий методы `addMouseListener()` и `addMouseMotionListener()`, является суперклассом `Applet`. Использование одного и того же объекта в качестве источника и слушателя событий типично для аплетов.

Внутри `init()` аплет регистрирует самого себя в качестве слушателя событий мыши. Это делается методами `addMouseListener()` и `addMouseMotionListener()`, которые, как уже упоминалось, являются членами `Component`. Эти методы показаны ниже:

```
void addMouseListener(MouseListener ml)
void addMouseMotionListener(MouseMotionListener mml)
```

Здесь *ml* — ссылка на объект, принимающий события мыши, а *mml* — ссылка на объект, принимающий события перемещения мыши. В данной программе в обоих методах используется один и тот же объект.

Затем аплет реализует все методы, определенные в интерфейсах `MouseListener` и `MouseMotionListener`. Таким образом, существуют обработчики событий для всех разнообразных событий мыши. Каждый из них обрабатывает собственное событие и затем возвращает управление.

Обработка событий клавиатуры

Чтобы обработать клавиатурные события, используется та же общая архитектура, то и в примере с событиями мыши, приведенном в предыдущем разделе. Отличие, конечно, в том, что здесь вы будете реализовывать интерфейс `KeyListener`.

Прежде чем обращаться к примеру, полезно будет еще раз рассмотреть процесс генерации клавиатурных событий. При нажатии клавиши генерируется событие `KEY_PRESSED`. Это приводит к вызову обработчика событий `keyPressed()`. При отпускании клавиши генерируется событие `KEY_RELEASED` и выполняется обработчик `keyReleased()`. Если нажатие клавиши генерирует символ, то также возникает событие `KEY_TYPED` и вызывается обработчик `keyTyped()`. Таким образом, всякий раз, когда пользователь нажимает клавишу, генерируются как минимум два, а то и три события. Если вас интересуют действительные символы, введенные с клавиатуры, то вы можете игнорировать информацию о нажатии и отпускании клавиш. Однако если ваша программа должна обрабатывать специальные клавиши вроде клавиш со стрелками или функциональных клавиш, то их следует отслеживать в обработчике `keyPressed()`.

В следующем аплете демонстрируется клавиатурный ввод. Он отображает нажатые клавиши в окне аплета и показывают состояние нажата/отпущена для каждой клавиши в строке состояния.

```
// Использование обработчиков событий 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");
      // showStatus("Клавиша нажата");
   }

   public void keyReleased(KeyEvent ke) {
      showStatus("Key Up");
      // showStatus("Клавиша отпущена");
   }
```

```
    public void keyTyped(KeyEvent ke) {
      msg += ke.getKeyChar();
      repaint();
    }
    // Отображение нажатых клавишей.
    public void paint(Graphics g) {
      g.drawString(msg, X, Y);
    }
}
```

Пример работы этого аплета показан на рис. 22.2.

Рис. 22.2. Пример работы аплета, использующего обработчики событий клавиатуры

Если вы хотите обрабатывать специальные клавиши, такие как клавиши со стрелками и функциональные, то вам следует реагировать на них в обработчике `keyPressed()`. Такие клавиши в `keyTyped()` не доступны. Чтобы идентифицировать эти клавиши, можно использовать коды виртуальных клавиш. Например, следующий аплет выводит имена некоторых специальных клавиш.

```
// Использование некоторых кодов виртуальных 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");
  // showStatus("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>";
       // msg += "<Стрелка влево>";
       break;
     case KeyEvent.VK_RIGHT:
       msg += "<Right Arrow>";
       // msg += "<Стрелка вправо>";
       break;
  }
  repaint();
}
public void keyReleased(KeyEvent ke) {
  showStatus("Key Up");
  showStatus("Клавиша отпущена");
}
public void keyTyped(KeyEvent ke) {
  msg += ke.getKeyChar();
  repaint();
}
// Отобразить нажатые клавиши.
public void paint(Graphics g) {
  g.drawString(msg, X, Y);
}
}
```

Пример работы этого аплета показан на рис. 22.3.

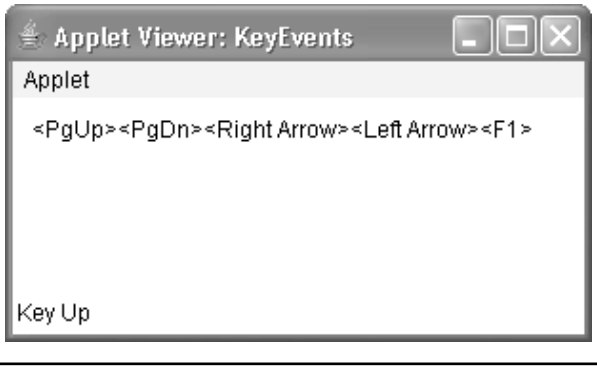

Рис. 22.3. Пример работы аплета, использующего некоторые коды виртуальных клавиш

Процедуры, показанные в приведенных примерах обработки событий мыши и клавиатуры, могут быть обобщены для обработки событий любого типа, включая события, генерируемые элементами управления. В последующих главах вы увидите множество примеров обработки событий других типов, но все они следуют одной и той же базовой структуре, что и только что описанные программы.

Классы адаптеров

В Java имеется специальное средство, называемое *классом адаптера*, который в некоторых ситуациях упрощает реализацию обработчиков событий. Класс адаптера предлагает пустую реализацию всех методов интерфейса слушателя событий. Классы адаптеров удобны, когда вы хотите принимать и обрабатывать только некоторые из событий, обрабатываемых определенным интерфейсом слушателя. Вы можете определить новый класс для использования в качестве слушателя событий, расширив один из классов адаптеров и реализовав только те события, в которых вы заинтересованы.

Например, класс `MouseMotionAdapter` имеет два метода: `mouseDragged()` и `mouseMoved()`, которые являются методами, определенными в интерфейсе `MouseMotionListener`. Если вы заинтересованы только в событиях перетаскивания мыши, можете просто расширить `MouseMotionAdapter` и переопределить `mouseDragged()`. Пустая реализация `mouseMoved()` обработает за вас события перемещения мыши.

В табл. 22.10 перечислены часто используемые классы адаптеров в `java.awt.event` и отмечены интерфейсы, реализуемые каждым из них.

Таблица 22.10. Часто используемые интерфейсы слушателей, реализуемые классами адаптеров

Класс адаптера	Интерфейс слушателя
`ComponentAdapter`	`ComponentListener`
`ContainerAdapter`	`ContainerListener`
`FocusAdapter`	`FocusListener`
`KeyAdapter`	`KeyListener`
`MouseAdapter`	`MouseListener`
`MouseMotionAdapter`	`MouseMotionListener`
`WindowAdapter`	`WindowListener`

В следующем примере демонстрируется применения адаптера. Он отображает сообщение в строке состояния средства просмотра аплетов, когда выполняется щелчок кнопкой мыши или перетаскивание. Однако все прочие события мыши молча игнорируются. Программа состоит из трех классов: `AdapterDemo` расширяет `Applet`. Его метод `init()` создает экземпляр `MyMouseAdapter` и регистрирует этот объект для получения уведомлений о событиях мыши. Также он создает экземпляр `MyMouseMotionAdapter` и регистрирует его для получения уведомлений о событиях перемещения мыши. Оба конструктора принимают ссылку на аплет в качестве аргумента.

`MyMouseAdapter` расширяет `MouseAdapter` и переопределяет метод `mouseClicked()`. Все прочие события мыши молча игнорируются кодом, унаследованным от класса `MouseAdapter`. `MyMouseMotionAdapter` расширяет `MouseMotionAdapter` и переопределяет метод `mouseDragged()`. Другое событие перемещения мыши молча игнорируется кодом, унаследованным от класса `MouseMotionAdapter`.

Обратите внимание, что оба класса слушателей событий сохраняют ссылку на аплет. Эта информация предоставляется в виде аргумента и используется позднее для вызова метода `showStatus()`.

```
// Демонстрация применения 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;
}
// Обработка щелчка мыши.
public void mouseClicked(MouseEvent me) {
  adapterDemo.showStatus("Mouse clicked");
  // adapterDemo.showStatus("Щелчок кнопкой мыши");
}
}

class MyMouseMotionAdapter extends MouseMotionAdapter {
AdapterDemo adapterDemo;
public MyMouseMotionAdapter(AdapterDemo adapterDemo) {
  this.adapterDemo = adapterDemo;
}
// Обработка перетаскивания мыши.
public void mouseDragged(MouseEvent me) {
  adapterDemo.showStatus("Mouse dragged");
  // adapterDemo.showStatus("Перетаскивание мышью");
}
}
```

Как видите, отсутствие необходимости реализовывать все методы, определенные интерфейсами `MouseMotionListener` и `MouseListener`, позволяет сэкономить ощутимое количество усилий и избавляет ваш код от перегрузки пустыми методами. В качестве упражнения вы можете попробовать переписать один из приведенных ранее примеров, обрабатывающих клавиатурный ввод, с использованием `KeyAdapter`.

Вложенные классы

В главе 7 объяснялись основы вложенных классов. Сейчас вы убедитесь, насколько они важны. Напомним, что *вложенный класс* — это класс, определенный внутри другого класса или даже внутри выражения. В настоящем разделе мы проиллюстрируем, как могут использоваться вложенные классы для упрощения кода в случае классов адаптеров событий.

Чтобы понять выгоду, которую обеспечивают вложенные классы, рассмотрим аплет, приведенный в следующем листинге. В нем *не используются* вложенные классы. Его назначение — отобразить строку "Mouse Pressed" в строке состояния средства просмотра аплетов или браузера, когда нажата кнопка мыши. В этой программе присутствует еще два класса верхнего уровня. `MousePressedDemo` расширяет `Applet`, а `MyMouseAdapter` расширяет `MouseAdapter`. Метод `init()` в `MousePressedDemo` создает экземпляр `MyMouseAdapter` и предоставляет этот объект в качестве аргумента методу `addMouseListener()`.

Обратите внимание, что ссылка на аплет выступает в качестве аргумента конструктора `MyMouseAdapter`. Эта ссылка сохраняется в переменной экземпляра для последующего использования методом `mousePressed()`. Когда нажимается кнопка мыши, вызывается метод `showStatus()` аплета через сохраненную ссылку на аплет. Другими словами, `showStatus()` вызывается относительно ссылки на аплет, сохраненной в `MyMouseAdapter`.

```
// В этом аплете НЕ используются вложенные классы.
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");
  // mousePressedDemo.showStatus("Кнопка мыши нажата");
}
}
```

В следующем листинге показано, как можно усовершенствовать предыдущую программу, используя вложенный класс. Здесь `InnerClassDemo` — класс верхнего уровня, расширяющий `Applet`. `MyMouseAdapter` — вложенный класс, расширяющий `MouseAdapter`. Поскольку `MyMouseAdapter` определен внутри области определения `InnerClassDemo`, он имеет доступ ко всем переменным и методам, находящимся в контексте этого класса. Таким образом, метод `mousePressed()` может вызывать метод `showStatus()` непосредственно. Более нет необходимости делать это через сохраненную ссылку на аплет. А потому не нужно и передавать `MyMouseAdapter()` ссылку на вызывающий объект.

```
// Демонстрация применения вложенного 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");
     // showStatus("Кнопка мыши нажата");
  }
  }
}
```

Анонимные вложенные классы

Анонимный вложенный класс — это класс, которому не назначено имя. В данном разделе мы проиллюстрируем, как анонимный вложенный класс может облегчить написание обработчиков событий. Рассмотрим аплет, показанный в следующем листинге. Как и раньше, его назначение — отобразить строку "Mouse Pressed" в строке состояния средства просмотра аплетов или браузера, когда нажата кнопка мыши.

```
// Демонстрация применения анонимного вложенного 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");
       // showStatus("Кнопка мыши нажата");
     }
  });
}
}
```

В этой программе присутствует только один класс верхнего уровня — `AnonymousInnerClassDemo`. Метод `init()` вызывает метод `addMouseListener()`. Его аргументом служит выражение, определяющее и создающее экземпляр анонимного вложенного класса. Давайте тщательно проанализируем это выражение.

Синтаксис `new MouseAdapter(){...}` указывает компилятору, что код между фигурными скобками определяет анонимный вложенный класс. Более того, этот класс расширяет `MouseAdapter`. Этот новый класс не имеет имени, но его экземпляр автоматически создается при выполнении этого выражения.

Поскольку анонимный вложенный класс определен внутри контекста `AnonymousInnerClassDemo`, он имеет доступ ко всем переменным и методам, находящимся в контексте этого класса. Поэтому он может вызывать метод `showStatus()` непосредственно.

Как видите, именованные и анонимные вложенные классы решают некоторые досадные проблемы простым и эффективным способом. Они также позволяют вам создавать более эффективный код.

23
ГЛАВА

Введение в AWT: работа с окнами, графикой и текстом

Абстрактный оконный инструментарий (Abstract Window Toolkit — AWT) уже был представлен в главе 21, где использовался в нескольких примерах аплетов. В настоящей главе мы займемся его более глубоким исследованием. AWT включает в себя многочисленные классы и методы, позволяющие создавать и управлять окнами. Кроме того, он служит фундаментом для Swing. AWT достаточно велик, и полное его описание легко бы заняло целую книгу. Поэтому невозможно описать во всех подробностях каждый метод, класс или переменную экземпляра AWT. Однако в этой и двух последующих главах мы объясним базовые приемы эффективного применения AWT для создания аплетов и независимых приложений с графическим интерфейсом пользователя. После этого вы получите возможность самостоятельно поработать и с другими частями AWT. Кроме того, вы сможете легко перейти к Swing.

В настоящей главе вы научитесь тому, как создавать и управлять окнами, шрифтами, выводом текста и графикой. В главе 24 будут описаны различные элементы управления, такие как линейки прокрутки и экранные кнопки, поддерживаемые AWT. Также там пойдет речь о дополнительных аспектах механизма обработки событий Java. Глава 25 будет посвящена подсистемам графических изображений и анимации AWT.

Хотя чаще всего AWT используется в аплетах, этот инструментарий также применяется при создании отдельных окон, выполняемых в среде графического пользовательского интерфейса (GUI), такой как Windows. По причине согласованности большинство примеров этой главы представлены в виде аплетов. Чтобы запускать их, вам понадобится средство просмотра аплетов или Java-совместимый Web-браузер. Несколько программ, однако, продемонстрируют создание самостоятельных оконных программ.

Прежде чем начать, стоит упомянуть еще об одном. В настоящее время большинство программ на Java построены на основе пользовательского интерфейса Swing. Поскольку Swing предлагает более широкие возможности, нежели AWT, в отношении создания таких распространенных компонентов GUI, как кнопки, окна списка и флажки, очень легко прийти к мысли, что AWT утратил свое значение. Однако такое заключение ошибочно.

Как уже упоминалось, Swing построен на базе AWT. А потому многие аспекты AWT также касаются и Swing. Более того, многие классы AWT используются Swing, прямо или опосредованно. И, наконец, для некоторых типов небольших программ (особенно маленьких аплетов), которые очень ограниченно используют GUI, имеет смысл применять AWT вместо Swing. Поэтому даже несмотря на то, что большинство интерфейсов в наши дни базируются на Swing, хорошее знание AWT все еще востребовано. Просто примите к сведению, что без знания AWT вы не можете считать себя профессиональным программистом на Java.

На заметку! *Если вы еще не читали главу 22, сделайте это прямо сейчас. В ней представлен обзор обработки событий, которые будут использоваться во многих примерах настоящей главы.*

Классы AWT

Классы AWT содержатся в пакете `java.awt`. Это один из наиболее объемных пакетов Java. К счастью, благодаря логической организации в виде иерархии "сверху вниз", понять и использовать его гораздо легче, чем может показаться на первый взгляд. В табл. 23.1 перечислены некоторые из множества классов AWT.

Таблица 23.1. Примеры классов AWT

Класс	Описание
`AWTEvent`	Инкапсулирует события AWT.
`AWTEventMulticaster`	Доставляет события множеству слушателей.
`BorderLayout`	Граничный диспетчер раскладки. Использует пять компонентов: North, South, East, West и Center.
`Button`	Создает элемент управления = экранную кнопку.
`Canvas`	Пустое, свободное от семантики окно.
`CardLayout`	Карточный диспетчер раскладки. Эмулирует индексированные карты. Отображается только одна, находящаяся сверху.
`Checkbox`	Создает элемент управления — помечаемый флажок.
`CheckboxGroup`	Создает группу элементов управления — флажков.
`CheckboxMenuItem`	Создает включаемый (on/off) пункт меню.
`Choice`	Создает всплывающий список.
`Color`	Управляет цветами в переносимой и не зависящей от платформы манере.
`Component`	Абстрактный суперкласс для различных компонентов AWT.
`Container`	Подкласс `Component`, который может содержать другие компоненты.
`Cursor`	Инкапсулирует курсор — битовую карту.
`Dialog`	Создает диалоговое окно.
`Dimension`	Специфицирует измерения объекта. Ширина сохраняется в `width`, а высота — в `height`.
`Event`	Инкапсулирует события.

Продолжение табл. 23.1

Класс	Описание
EventQueue	Очередь событий.
FileDialog	Создает окно, в котором можно выбрать файл.
FlowLayout	Потоковый диспетчер раскладки. Располагает компоненты слева направо и сверху вниз.
Font	Инкапсулирует шрифт печати.
FontMetrics	Инкапсулирует различную информацию, относящуюся к шрифту. Эта информация помогает отображать текст в окне.
Frame	Создает стандартное окно, снабженное линейкой заголовка, элементами управления размерами и линейкой меню.
Graphics	Инкапсулирует графический контекст. Этот контекст используется разнообразными методами вывода для отображения в окне.
GraphicsDevice	Описывает графическое устройство, такое как экран или принтер.
GraphicsEnvironment	Описывает коллекцию доступных объектов Font и GraphicsDevice.
GridBagConstraints	Описывает различные ограничения, относящиеся к классу GridBagLayout.
GridBagLayout	Диспетчер сетчатой управляемой раскладки. Отображает компоненты в соответствии с ограничениями, указанными в GridBagConstraints.
GridLayout	Диспетчер сетчатой раскладки. Отображает компоненты в двумерной сетке.
Image	Инкапсулирует графические образы.
Insets	Инкапсулирует границы контейнера.
Label	Создает метку, отображающую строку.
List	Создает список, из которого пользователь может выбирать. Аналогичен стандартному окну списка Windows.
MediaTracker	Управляет медиа-объектами.
Menu	Создает выпадающее меню.
MenuBar	Создает линейку меню.
MenuComponent	Абстрактный класс, реализованный различными классами меню.
MenuItem	Создает пункт меню.
MenuShortcut	Инкапсулирует “горячую клавишу” для быстрого вызова пункта меню.
Panel	Простейший конкретный подкласс Container.
Point	Инкапсулирует пару декартовых координат, хранящихся в x и y.
Polygon	Инкапсулирует многоугольник.
PopupMenu	Создает всплывающее меню.
PrintJob	Абстрактный класс, представляющий задание печати.
Rectangle	Инкапсулирует прямоугольник.
Robot	Поддерживает автоматическое тестирование приложений на основе AWT.

Окончание табл. 23.1

Класс	Описание
`Scrollbar`	Создает элемент управления — линейку прокрутки.
`ScrollPane`	Контейнер, предоставляющий горизонтальную и вертикальную линейки прокрутки для другого компонента.
`SystemColor`	Содержит цвета для элементов (виджетов) GUI, таких как окна, линейки прокрутки, текст и прочее.
`TextArea`	Создает элемент управления — многострочный текстовый редактор.
`TextComponent`	Суперкласс для `TextArea` и `TextField`.
`TextField`	Создает элемент управления — однострочный текстовый редактор.
`Toolkit`	Абстрактный класс, реализованный AWT.
`Window`	Создает окно без рамки, без линейки меню и заголовка.

Хотя базовая структура AWT не менялась со времен Java 1.0, некоторые из оригинальных методов устарели и заменены новыми. Для обеспечения обратной совместимости Java все еще поддерживает исходные методы версии 1.0. Однако поскольку эти методы не предназначены для использования в новом коде, мы не станем их описывать в настоящей книге.

Основы окон

AWT определяет окна в соответствии с иерархией классов, которые добавляют функциональность и специфичность на каждом уровне. Два наиболее часто используемых типа окон наследуются либо от `Panel`, что используется в аплетах, либо от класса `Frame`, который создает стандартное окно приложения. Большую часть своей функциональности эти окна наследуют от своих родительских классов. Поэтому описание иерархии классов, относящихся к этим двум классам, является основополагающим для их понимания. На рис. 23.1 показана иерархия классов для `Panel` и `Frame`. Рассмотрим каждый из этих классов.

Component

В вершине иерархии находится класс `Component`. `Component` — это абстрактный класс, инкапсулирующий все атрибуты визуального компонента. Все элементы пользовательского интерфейса являются подклассами `Component`. Он определяет свыше сотни общедоступных методов, отвечающих за управление событиями, такими как клавишный и мышиный ввод, перемещение и изменение размеров окна, а также перерисовка. (Вы уже использовали многие из этих методов, когда создавали аплеты в главах 21 и 22.) Объект `Component` отвечает за запоминание текущих цветов фона и переднего плана, а также текущего выбранного шрифта для печати.

Container

Класс `Container` — подкласс `Component`. Он имеет дополнительные методы, позволяющие вкладывать в него другие объекты `Component`.Другие объекты `Container` также могут быть находиться внутри `Container` (поскольку они сами являются экземплярами `Component`). Это позволяет строить многоуровневые системы вложенности. Контейнер отвечает за раскладку (то есть, расположение) любых компонентов, которые он содержит.

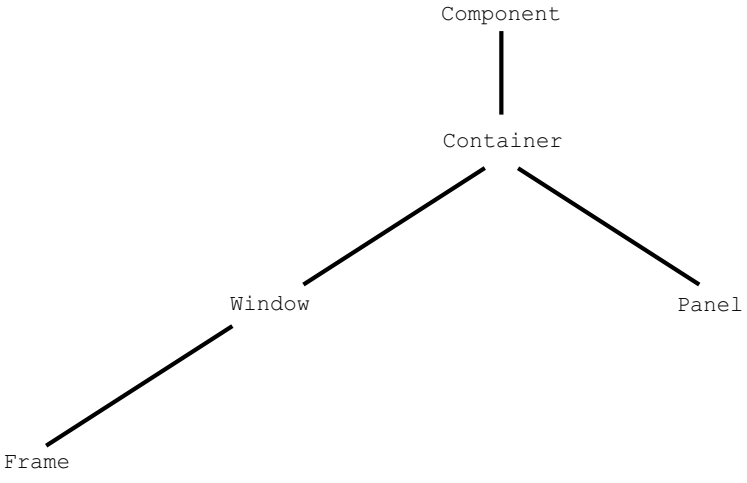

Рис. 23.1. Иерархия классов для `Panel` и `Frame`

Это достигается через использование различных диспетчеров раскладки, о которых вы узнаете в главе 24.

Panel

Класс `Panel` — конкретный подкласс `Container`. Он не добавляет никаких новых методов, а просто реализует `Container`. Таким образом, `Panel` может обеспечивать рекурсивную вложенность и представляет собой конкретный экранный компонент. `Panel` — суперкласс для `Applet`. Когда экранный вывод направляется в аплет, он рисуется на поверхности объекта `Panel`. По сути, `Panel` — это окно, лишенное линейки заголовка, линейки меню и рамки. И поэтому вы не можете видеть упомянутых элементов, когда аплет запущен внутри браузера. Когда же вы запускаете аплет в средстве просмотра аплетов, то заголовок и рамку предоставляет это средство.

Другие компоненты могут быть добавлены к объекту `Panel` с помощью его метода `add()` (унаследованного от `Container`). Как только эти компоненты добавлены, вы можете позиционировать их и изменять размеры вручную, используя методы `setLocation()`, `setSize()`, `setPreferredSize()` или `setBounds()`, определенные в `Component`.

Window

Класс `Window` создает окно верхнего уровня. Окно верхнего уровня не содержится внутри другого объекта; оно располагается непосредственно на рабочем столе. Обычно вам не придется создавать объекты `Window` непосредственно. Вместо этого вы будете работать с подклассом `Window`, называемым `Frame`, который описан ниже.

Frame

Класс `Frame` инкапсулирует то, что обычно воспринимается как “окно”. Это подкласс `Window`, имеющий линейку заголовка, линейку меню, границы и элементы управления размером. Если вы создаете объект `Frame` внутри аплета, он будет содержать предупреждающее сообщение, такое как “Java Applet Window”, предназначенное пользователю, что-

бы известить его о том, что окно аплета было создано. Это окно предупредит пользователя о том, что окно, которое он видит, запущено аплетом, а не программным обеспечением, выполняющимся на самом компьютере. (Аплет, маскирующийся под приложение, выполняющееся на хосте, может быть использовано для получения паролей и другой важной информации без ведома пользователя.) Когда окно `Frame` создается отдельно стоящим приложением вместо аплета, то тем самым создается нормальное окно.

Canvas

Не являющийся частью иерархии аплетов или рамочных окон, класс `Canvas`, возможно, не покажется вам особенно полезным. `Canvas` инкапсулирует пустое окно, в котором вы можете рисовать. Пример применения `Canvas` будет приведен далее в настоящей книге.

Работа с рамочными окнами

После аплета тип окон, которые вам придется создавать чаще всего, унаследован от `Frame`. Вы будете создавать дочерние окна внутри аплетов и окна верхнего уровня или дочерние — для отдельно выполняемых приложений. Как упоминалось, это создает окна в стандартном стиле.

Вот два конструктора класса `Frame`:

```
Frame()
Frame(String title)
```

Первая форма создает стандартное окно без заголовка. Вторая форма создает окно с указанным в *`title`* заголовком. Обратите внимание, что вы не можете специфицировать размеры окна. Вместо этого должны устанавливать размер окна после его создания.

Существует несколько ключевых методов, которые вы будете использовать при работе с окнами `Frame`. Сейчас мы их и рассмотрим.

Установка размеров окна

Метод `setSize()` используется для установки размеров окна. Его сигнатура показана ниже:

```
void setSize(int newWidth, int newHeight)
void setSize(Dimension newSize)
```

Новый размер окна указан в *`newWidth`* и *`newHeight`* или же в полях `width` и `height` передаваемого объекта `Dimension`, переданного в *`newSize`*. Размеры задаются в пикселях.

Метод `getSize()` применяется для получения текущего размера окна. Его сигнатура выглядит следующим образом:

```
Dimension getSize()
```

Этот метод возвращает текущий размер окна в полях `width` и `height` объекта `Dimension`.

Сокрытие и отображение окна

После того как рамочное окно создано, оно остается невидимым до тех пор, пока вы не вызовите `setVisble()`. Его сигнатура такова:

```
void setVisible(boolean visibleFlag)
```

Компонент видим, если этому методу передается аргумент `true`. В противном случае он скрыт.

Установка заголовка окна

Вы можете изменить заголовок рамочного окна, используя метод `setTitle()`, который имеет следующую общую форму:

```
void setTitle(String newTitle)
```

Здесь *`newTitle`* — новый заголовок окна.

Закрытие рамочного окна

При использовании рамочного окна ваша программа должна удалять окно с экрана после его закрытия, вызывая `setVisible(false)`. Чтобы перехватить событие закрытия окна, вы должны реализовать метод `windowClosing()` интерфейса `WindowListener`. Внутри `windowClosing()` вы должны удалить окно с экрана. Пример, приведенный в следующем разделе, иллюстрирует этот прием.

Создание рамочного окна в аплете

Хотя можно создавать окно просто созданием экземпляра `Frame`, вам редко доведется поступать так, поскольку с таким окном не так много можно делать. Например, вы не сможет принимать или обрабатывать события, которые происходят в нем, или просто выводить в него информацию. В основном вам придется создавать подклассы `Frame`. Это позволит переопределять методы `Frame` и обеспечивать обработку событий.

Создать новое рамочное окно из аплета достаточно просто. Для начала создается подкласс `Frame`. Затем переопределяется любой из стандартных методов аплета, таких как `init()`, `start()` и `stop()`, чтобы отображать или скрывать фрейм по необходимости. И, наконец, реализуется `windowClosing()`.

Как только вы определите подкласс `Frame`, то сможете создать объект этого класса. Это приведет к появлению рамочного окна, которое, однако, изначально будет невидимым. Вы делаете его видимым посредством вызова `setVisible()`. При создании окно получает высоту и ширину по умолчанию. Вы можете установить размеры окна явно с помощью метода `setSize()`.

Приведенный ниже аплет создает подкласс `Frame` по имени `SampleFrame`. Экземпляр окна этого класса создается внутри метода `init()` класса `AppletFrame`. Обратите внимание на то, что `SampleFrame` вызывает конструктор `Frame`. Это позволяет создать стандартное рамочное окно с заголовком, переданным в `title`. Этот пример переопределяет методы аплета `start()` и `stop()` так, что они, соответственно, отображают и скрывают дочернее окно. Это позволяет автоматически удалить окно, когда вы прерываете работу аплета, когда закрываете окно, или, при использовании браузера — когда переходите на другую страницу. Это также вызывает отображение дочернего окна, когда браузер возвращается к аплету.

```
// Создание дочернего окна из аплета.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
```

```
/*
  <applet code="AppletFrame" width=300 height=50>
  </applet>
*/

// Создание подкласса Frame.
class SampleFrame extends Frame {
SampleFrame(String title) {
  super(title);

  // создание объекта для обработки событий окна
  MyWindowAdapter adapter = new MyWindowAdapter(this);

  // регистрация его для получения событий
  addWindowListener(adapter);
}

public void paint(Graphics g) {
  g.drawString("This is in frame window", 10, 40);
}
}

class MyWindowAdapter extends WindowAdapter {
SampleFrame sampleFrame;

public MyWindowAdapter(SampleFrame sampleFrame) {
  this.sampleFrame = sampleFrame;
}

public void windowClosing(WindowEvent we) {
  sampleFrame.setVisible(false);
}
}

// Создание рамочного окна.
public class AppletFrame extends Applet {
Frame f;

public void init() {
  f = new SampleFrame("A Frame Window");
  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("Это окно аплета", 10, 20);
}
}
```

Результат выполнения этого аплета показан на рис. 23.2.

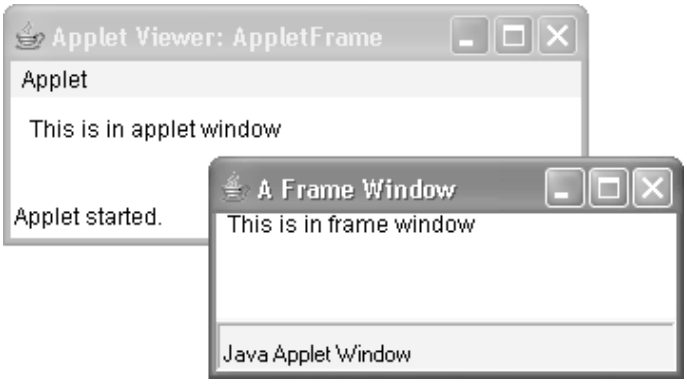

Рис. 23.2. Создание дочернего окна из аплета

Обработка событий в рамочном окне

Поскольку `Frame` — подкласс `Component`, он наследует все, что определено в `Component`. Это значит, что вы можете использовать и управлять рамочным окном точно так же, как делаете это с главным окном аплета. Например, вы можете переопределить `paint()` для отображения вывода, вызвать `repaint()`, когда вам нужно восстановить окно, и добавить обработчики событий. Всякий раз, когда возникают события в окне, вызываются обработчики, определенные этим окном. Каждое окно обрабатывает свои собственные события. Например, следующая программа создает окно, реагирующее на события мыши. Главное окно аплета также реагирует на события мыши. Если вы поэкспериментируете с этой программой, то увидите, что события мыши посылаются окну, в котором они происходят.

```
// Обработка событий мыши в дочернем окне и окне аплета.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="WindowEvents" width=300 height=50>
</applet>
*/

// Создание подкласса 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 объект для получения его собственных событий мыши
   addMouseListener(this);
   addMouseMotionListener(this);
   // создать объект для обработки событий окна
   MyWindowAdapter adapter = new MyWindowAdapter(this);
   // зарегистрировать для получения этих событий
   addWindowListener(adapter);
}
```

```
// Обработка щелчка мыши.
public void mouseClicked(MouseEvent me) {
}
// Обработка входа курсора мыши.
public void mouseEntered(MouseEvent evtObj) {
   // сохранить координаты
   mouseX = 10;
   mouseY = 54;
   msg = "Mouse just entered child.";
   // msg = "Курсор мыши только что вошел в дочернее окно.";
   repaint();
}
// Обработка выхода курсора мыши.
public void mouseExited(MouseEvent evtObj) {
   // сохранить координаты
   mouseX = 10;
   mouseY = 54;
   msg = "Mouse just left child window.";
   // msg = "Курсор мыши только что покинул дочернее окно.";
   repaint();
}
// Обработка нажатия кнопки мыши.
public void mousePressed(MouseEvent me) {
   // сохранить координаты
   mouseX = me.getX();
   mouseY = me.getY();
   msg = "Down";
   repaint();
}
// Обработка отпускания кнопки мыши.
public void mouseReleased(MouseEvent me) {
   // сохранить координаты
   mouseX = me.getX();
   mouseY = me.getY();
   msg = "Up";
   repaint();
}
// Обработка перетаскивания мышью.
public void mouseDragged(MouseEvent me) {
   // сохранить координаты
   mouseX = me.getX();
   mouseY = me.getY();
   movX = me.getX();
   movY = me.getY();
   msg = "*";
   repaint();
}
// Обработка 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);
   // g.drawString("Курсор мыши в " + movX + ", " + movY, 10, 40);
}
}

class MyWindowAdapter extends WindowAdapter {
SampleFrame sampleFrame;
public MyWindowAdapter(SampleFrame sampleFrame) {
  this.sampleFrame = sampleFrame;
}
public void windowClosing(WindowEvent we) {
  sampleFrame.setVisible(false);
}
}

// Окно аплета.
public class WindowEvents extends Applet
  implements MouseListener, MouseMotionListener {
SampleFrame f;
String msg = "";
int mouseX=0, mouseY=10;
int movX=0, movY=0;

// Создание рамочного окна.
public void init() {
  f = new SampleFrame("Handle Mouse Events");
  f.setSize(300, 200);
  f.setVisible(true);
  // зарегистрировать объект для получения его собственных событий мыши
  addMouseListener(this);
  addMouseMotionListener(this);
}

// Удаление рамочного окна при останове аплета.
public void stop() {
  f.setVisible(false);
}

// Показать рамочное окно при запуске аплета.
public void start() {
  f.setVisible(true);
}

// Обработка щелчка мыши.
public void mouseClicked(MouseEvent me) {
}

// Обработка входа мыши.
public void mouseEntered(MouseEvent me) {
  // сохранить координаты
  mouseX = 0;
  mouseY = 24;
  msg = "Mouse just entered applet window.";
  // msg = "Курсор мыши только что вошел в окно аплета.";
  repaint();
}
```

```
// Обработка выхода мыши.
public void mouseExited(MouseEvent me) {
   // сохранить координаты
   mouseX = 0;
   mouseY = 24;
   msg = "Mouse just left applet window.";
  // msg = "Курсор мыши только что покинул окно аплета.";
   repaint();
}

// Обработка нажатия кнопки мыши.
public void mousePressed(MouseEvent me) {
   // сохранить координаты
   mouseX = me.getX();
   mouseY = me.getY();
   msg = "Down";
   repaint();
}

// Обработка отпускания кнопки мыши.
public void mouseReleased(MouseEvent me) {
   // сохранить координаты
   mouseX = me.getX();
   mouseY = me.getY();
   msg = "Up";
   repaint();
}

// Обработка перетаскивания мыши.
public void mouseDragged(MouseEvent me) {
   // сохранить координаты
   mouseX = me.getX();
   mouseY = me.getY();
   movX = me.getX();
   movY = me.getY();
   msg = "*";
   repaint();
}

// Обработка перемещения мыши.
public void mouseMoved(MouseEvent me) {
   // сохранить координаты
   movX = me.getX();
   movY = me.getY();
   repaint(0, 0, 100, 20);
}

// Отображение сообщения в окне аплета
public void paint(Graphics g) {
   g.drawString(msg, mouseX, mouseY);
   g.drawString("Mouse at " + movX + ", " + movY, 0, 10);
   // g.drawString("Курсор мыши в " + movX + ", " + movY, 0, 10);
}
}
```

Результат выполнения этого аплета показан на рис. 23.3.

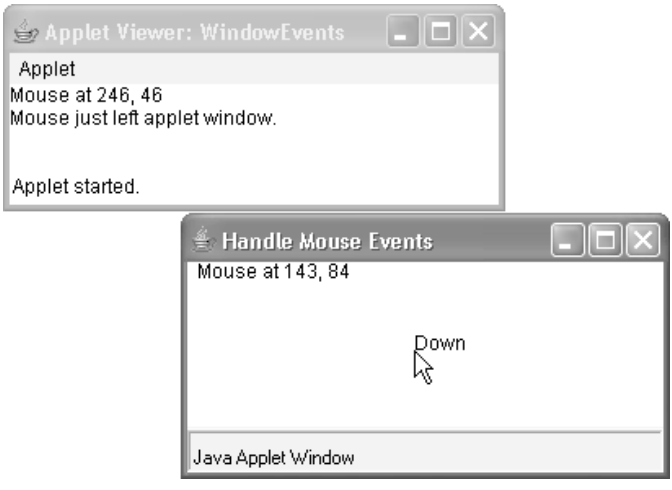

Рис. 23.3. Обработка событий мыши в дочернем окне и окне аплета

Создание оконной программы

Хотя создание аплетов — привычная задача для применения Java AWT, на основе AWT также возможно создавать отдельно стоящие приложения. Чтобы сделать это, просто создавайте экземпляр нужного окна или окон внутри `main()`. Например, следующая программа создает рамочное окно, реагирующее на щелчки мыши и нажатия клавиш.

```
// Создание приложения на базе AWT.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
// Создание рамочного окна.
public class AppWindow extends Frame {
String keymsg = "This is a test.";
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);
}
// Создание окна.
public static void main(String args[]) {
  AppWindow appwin = new AppWindow();
  appwin.setSize(new Dimension(300, 200));
  appwin.setTitle("An AWT-Based Application");
  // appwin.setTitle("AWT-приложение");
  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.mousemsg = "Клавиша мыши нажата в " + appWindow.mouseX +
                      ", " + appWindow.mouseY;
  appWindow.repaint();
}
}

class MyWindowAdapter extends WindowAdapter {
public void windowClosing(WindowEvent we) {
  System.exit(0);
}
}
```

Результат выполнения этого аплета показан на рис. 23.4.

Однажды созданное, рамочное окно начинает “жить собственной жизнью”. Обратите внимание, что `main()` завершается вызовом `appwin.setVisible(true)`. Однако программа продолжает выполняться до тех пор, пока вы не закроете окно. По сути, при создании оконного приложения вы используете `main()` для запуска окна верхнего уровня. После этого ваша программа функционирует как приложение на основе GUI, а не как консольное приложение вроде показанных ранее.

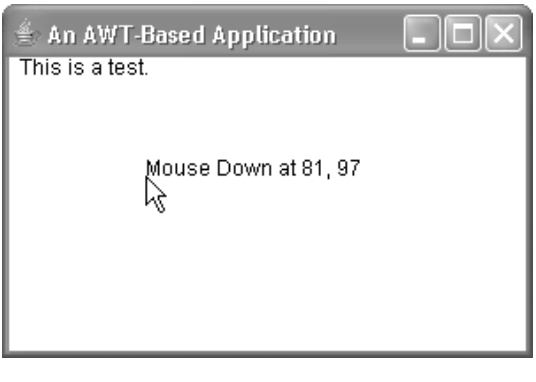

Рис. 23.4. Пример AWT-приложения

Отображение информации внутри окна

В наиболее общем случае окно является контейнером для информации. Хотя в предыдущих примерах мы уже выводили небольшие объемы текста в окно, все же до сих пор мы не начали использовать возможности окон представлять высококачественный текст и графику. На самом деле большая часть мощи AWT проявляется именно благодаря поддержке упомянутых вещей. По этой причине остаток настоящей главы мы посвятим обсуждению возможностей Java по выводу текста, графики и управлению шрифтами. Как вы убедитесь, они обеспечивают как необходимую мощность, так и гибкость.

Работа с графикой

AWT поддерживает богатый ассортимент графических методов. Вся графика рисуется относительно окна. Это может быть главное окно аплета, дочернее окно аплета или же окно автономного приложения. Начальная точка каждого окна находится в верхнем левом углу и имеет координаты 0,0. Координаты указываются в пикселях. Весь вывод в окно выполняется в конкретном графическом контексте. *Графический контекст*, инкапсулированный классом `Graphics`, получается двумя путями:

- передается в аплет, когда вызывается один из его методов, таких как `paint()` или `update()`;
- возвращается методом `getGraphics()` класса `Component`.

Для обеспечения согласованности остаток примеров настоящей главы продемонстрирует графику в главном окне аплета. Однако те же приемы применимы к любому окну.

Класс `Graphics` определяет множество функций рисования. Любая фигура может быть нарисована только контуром либо залита. Объекты рисуются и заливаются текущим выбранным цветом графики, которым по умолчанию является черный. Когда рисуется графический объект, выходящий за пределы окна, его вывод автоматически усекается. Давайте рассмотрим несколько методов рисования.

Рисование линий

Линии рисуются с помощью метода `drawLine()`, показанного ниже:

```
void drawLine(int startX, int startY, int endX, int endY)
```

`drawLine()` отображает линию в текущем цвете рисования, начинающуюся с точки *`startX`*,*`startY`* и заканчивающуюся в *`endX`*,*`endY`*.

Следующий аплет рисует несколько линий.

```
// Рисование линий
import java.awt.*;
import java.applet.*;
/*
<applet code="Lines" width=300 height=200>
</applet>
*/
public class Lines extends Applet {
public void paint(Graphics g) {
  g.drawLine(0, 0, 100, 100);
  g.drawLine(0, 100, 100, 0);
```

```
    g.drawLine(40, 25, 250, 180);
    g.drawLine(75, 90, 400, 400);
    g.drawLine(20, 150, 400, 40);
    g.drawLine(5, 290, 80, 19);
  }
  }
```

Результат выполнения этого аплета показан на рис. 23.5.

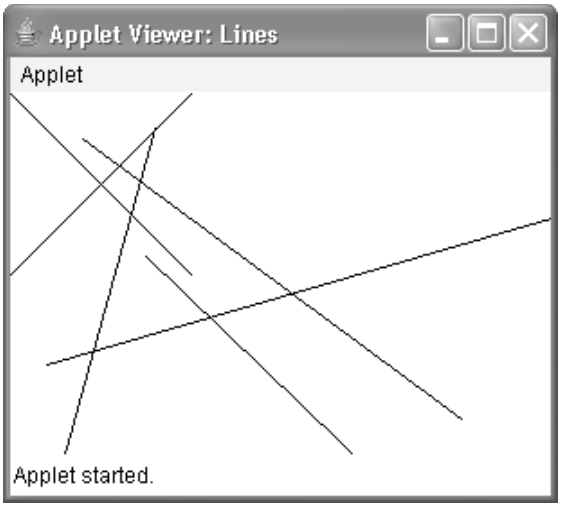

Рис. 23.5. Пример рисования линий

Рисование прямоугольников

Методы `drawRect()` и `fillRect()` отображают, соответственно, контурный и закрашенный прямоугольники.

Выглядят они так:

```
void drawRect(int top, int left, int width, int height)
void fillRect(int top, int left, int width, int height)
```

Левый верхний угол прямоугольника расположен в *top*,*left*. Размеры прямоугольника задаются *width* и *height*.

Чтобы нарисовать прямоугольник со скругленными углами, используйте методы `drawRoundRect()` и `fillRoundRect()`, показанные ниже:

```
void drawRoundRect(int top, int left, int width, int height,
                   int xDiam, int yDiam)
void fillRoundRect(int top, int left, int width, int height,
                   int xDiam, int yDiam)
```

Нарисованные этими методами прямоугольники будут иметь скругленные углы. Диаметр скругления по оси X указывается в *xDiam*. Диаметр скругления дуги по оси Y задается в *yDiam*.

Приведенный ниже аплет рисует несколько прямоугольников.

```
// Рисование прямоугольников
import java.awt.*;
import java.applet.*;
/*
<applet code="Rectangles" width=300 height=200>
</applet>
*/
public class Rectangles extends Applet {
public void paint(Graphics g) {
  g.drawRect(10, 10, 60, 50);
  g.fillRect(100, 10, 60, 50);
  g.drawRoundRect(190, 10, 60, 50, 15, 15);
  g.fillRoundRect(70, 90, 140, 100, 30, 40);
}
}
```

Результат выполнения этого аплета показан на рис. 23.6.

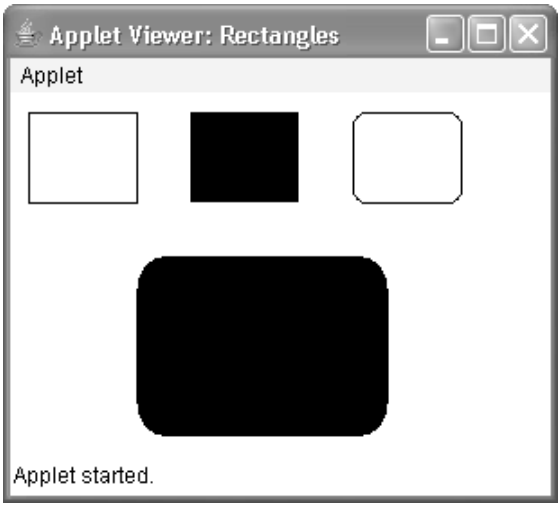

Рис. 23.6. Пример рисования прямоугольников

Рисование эллипсов и окружностей

Чтобы нарисовать эллипс, используйте `drawOval()`. Для заливки эллипса применяйте `fillOval()`. Эти методы выглядят следующим образом:

```
void drawOval(int top, int left, int width, int height)
void fillOval(int top, int left, int width, int height)
```

Эллипс рисуется внутри описанного прямоугольника, чей верхний левый угол имеет координаты *top*,*left*, а ширина и высота указаны в *width* и *height*. Чтобы нарисовать круг, в качестве описанного прямоугольника задавайте квадрат.

Следующая программа рисует несколько эллипсов:

```
// Рисование эллипсов.
import java.awt.*;
import java.applet.*;
/*
<applet code="Ellipses" width=300 height=200>
</applet>
*/
public class Ellipses extends Applet {
public void paint(Graphics g) {
  g.drawOval(10, 10, 50, 50);
  g.fillOval(100, 10, 75, 50);
  g.drawOval(190, 10, 90, 30);
  g.fillOval(70, 90, 140, 100);
}
}
```

Результат выполнения этого аплета показан на рис. 23.7.

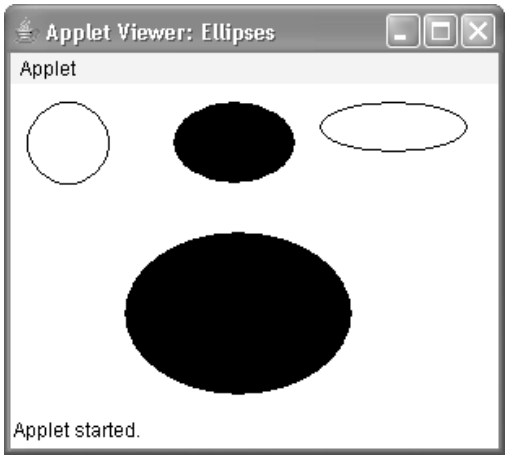

Рис. 23.7. Пример рисования эллипсов

Рисование дуг

Дуги могут быть нарисованы методами `drawArc()` и `fillArc()`, показанными ниже:

```
void drawArc(int top, int left, int width, int height, int startAngle,
             int sweepAngle)
void fillArc(int top, int left, int width, int height, int startAngle,
             int sweepAngle)
```

Дуга ограничена прямоугольником, чей верхний левый угол находится в точке с координатами *top*,*left*, а ширина и высота указаны в *width* и *height*. Дуга рисуется, начиная с угла *startAngle*, и продолжается на величину угла *sweepAngle*. Углы указываются в градусах. Ноль градусов соответствует горизонтали, направленной по часовой стрелке, показывающей три часа. Дуга рисуется в направлении против часовой стрелки, если *sweepAngle* положительно, и по часовой, если *sweepAngle* отрицательно. Таким образом, чтобы нарисовать дугу, начинающуюся с направления 12 часов по часам, и до 6 часов, нужно указать в качестве начального угла 90 градусов, а угла поворота — 180 градусов.

```
// Рисование дуг окружностей.
import java.awt.*;
import java.applet.*;
/*
<applet code="Arcs" width=300 height=200>
</applet>
*/
public class Arcs extends Applet {
public void paint(Graphics g) {
  g.drawArc(10, 40, 70, 70, 0, 75);
  g.fillArc(100, 40, 70, 70, 0, 75);
  g.drawArc(10, 100, 70, 80, 0, 175);
  g.fillArc(100, 100, 70, 90, 0, 270);
  g.drawArc(200, 80, 80, 80, 0, 180);
}
}
```

Результат выполнения этого аплета показан на рис. 23.8.

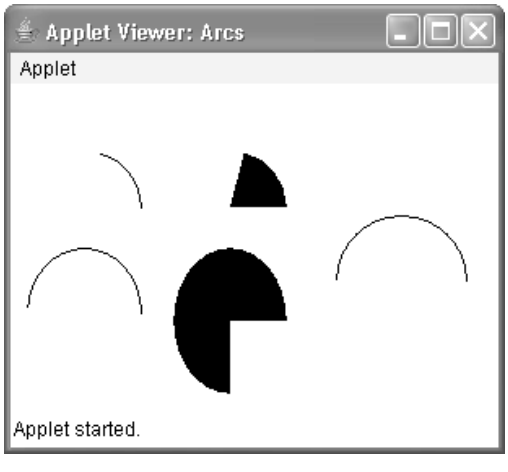

Рис. 23.8. Пример рисования дуг окружностей

Рисование многоугольников

Можно рисовать фигуры произвольной формы, используя методы `drawPolygon()` и `fillPolygon()`, показанные ниже:

```
void drawPolygon(int x[ ], int y[ ], int numPoints)
void fillPolygon(int x[ ], int y[ ], int numPoints)
```

Конечные точки многоугольника указываются парами координат в массивах *x* и *y*. Количество точек, определенных в *x* и *y*, указывается в *numPonts*. Существуют альтернативных формы этих методов, в которых многоугольник специфицируется объектом `Polygon`.

Следующий аплет рисует форму песочных часов:

```
// Рисование многоугольника.
import java.awt.*;
import java.applet.*;
/*
<applet code="HourGlass" width=230 height=210>
</applet>
*/
public class HourGlass extends Applet {
public void paint(Graphics g) {
  int xpoints[] = {30, 200, 30, 200, 30};
  int ypoints[] = {30, 30, 200, 200, 30};
  int num = 5;
  g.drawPolygon(xpoints, ypoints, num);
}
}
```

Результат выполнения этого аплета показан на рис. 23.9.

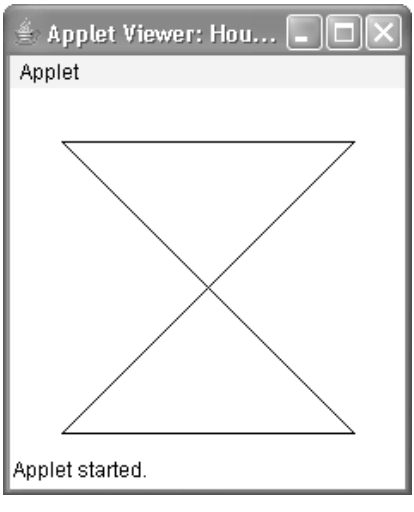

Рис. 23.9. Пример рисования многоугольника

Установка размеров графики

Часто возникает необходимость установить размер графического объекта таким, чтобы он занял текущий размер окна, в котором он нарисован. Чтобы добиться этого, получите текущие размеры окна вызовом `getSize()` для оконного объекта. Этот метод вернет размеры окна, инкапсулируя их в объекте `Dimension`. Получив текущий размер окна, вы можете соответствующим образом масштабировать графику.

Для демонстрации этого приема рассмотрим аплет, который, начиная с квадрата размером 200×200 пикселей, будет увеличивать его с каждым щелчком мыши на 25 пикселей до тех пор, пока тот не получит размер 500×500 пикселей. В этой точке следующий щелчок вернет его к размеру 200×200 пикселей, после чего процесс начнется сначала.

Внутри окна прямоугольник рисуется вдоль внутренних границ окна; внутри прямоугольника рисуется X, чтобы заполнить его. Этот аплет работает в `appletviewer`, но может не работать в окне браузера.

```
// Изменение размеров для заполнения текущего размера 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

Java поддерживает цвета в переносимой, независимой от устройства манере. Система цветов AWT позволяет специфицировать любой цвет по вашему желанию. Затем она находит наилучшее соответствие этому цвету, учитывая ограничения оборудования дисплея, на котором выполняется ваш аплет или программа. Таким образом, код не должен заботиться о различиях в поддержке цвета различными аппаратными устройствами. Цвет инкапсулирован в классе `Color`.

Как вы уже видели в главе 21, `Color` определяет несколько констант (вроде `Color.black`) для описания множества наиболее часто используемых цветов. Вы можете также создавать свои собственные цвета, используя один из доступных конструкторов цвета. Ниже приведены три наиболее часто используемых формы таких конструкторов:

```
Color(int red, int green, int blue)
Color(int rgbValue)
Color(float red, float green, float blue)
```

Первый конструктор принимает три ингредиента, задающие цвет как смесь красной, зеленой и синей составляющих. Эти значения должны находиться в пределах от 0 до 255, как в следующем примере:

```
new Color(255, 100, 100); // светло-красный
```

Второй конструктор цвета принимает единственный целочисленный аргумент, содержащий смесь интенсивностей красного, зеленого и синего, упакованную в одно целое число. Это целое число организовано так, что красная составляющая упакована в биты с 16 по 23, зеленая — с 8 по 15 и синяя — с 0 по 7. Вот пример применения этого конструктора:

```
int newRed = (0xff000000 | (0xc0 << 16) | (0x00 << 8) | 0x00);
Color darkRed = new Color(newRed);
```

И последний конструктор, `Color(float, float, float)`, принимает три значения с плавающей точкой (в диапазоне от 0,0 до 1,0), указывающие относительные значения смеси красного, зеленого и синего.

Создав подобным образом цвет, вы можете использовать его для установки цвета переднего плана и/или фона, используя методы `setForeground()` и `setBackground()`, описанные в главе 21. Также вы можете выбрать его в качестве текущего цвета рисования.

Методы `Color`

Класс `Color` определяет несколько методов, помогающих манипулировать цветами. Рассмотрим их.

Использование цвета, насыщенности и яркости

Цветовая модель "*цвет-насыщенность-яркость*" (hue-saturation-brightness — HSB) представляет собой альтернативу модели "красный-зеленый-синий" (red-green-blue — RGB) для указания определенного цвета. Образно говоря, *цвет* в модели HSB представляет собой цветовой круг. Он задается числом в диапазоне от 0,0 до 1,0 (приближенно образуя радугу, состоящую из красного, оранжевого, желтого, зеленого, синего, индиго и фиолетового). *Насыщенность* — вторая шкала в диапазоне от 0,0 до 1,0, представляет примеси для интенсификации цвета. Значения *яркости* также лежат в диапазоне от 0,0 до 1,0, где 1 — ярко-белый, а 0 — черный. `Color` поддерживает два метода, позволяющие выполнять преобразования между RGB и HSB; методы показаны ниже.

```
static int HSBtoRGB(float hue, float saturation, float brightness)
static float[ ] RGBtoHSB(int red, int green, int blue, float values[ ])
```

`HSBtoRGB()` возвращает упакованное значение RGB, совместимое с конструктором `Color(int)`. `RGBtoHSB()` возвращает массив чисел с плавающей точкой, представляющих значения HSB, соответствующие составляющим RGB. Если *`values`* не равно `null`, то этот массив представляет заданные значения HSB и возвращается. В противном случае создается новый массив, и значения HSB возвращаются в нем. В любом случае массив содержит цвет в элементе с индексом 0, насыщенность в элементе 1 и яркость — в элементе 2.

`getRed()`, `getGreen()`, `getBlue()`

Вы можете получить красную, зеленую и синюю составляющие цвета независимо, используя для этого методы `getRed()`, `getGreen()` и `getBlue()`, показанные ниже:

```
int getRed()
int getGreen()
int getBlue()
```

Каждый из этих методов возвращает компонент цвета RGB, извлеченный из вызывающего объекта `Color` в нижних 8 битах целого числа.

getRgb()

Чтобы получить упакованное RGB-представление цвета, предусмотрен метод `getRed()`, показанный ниже:

```
int getRGB()
```

Возвращаемое значение организовано, как было описано выше.

Установка текущего цвета графики

По умолчанию графические объекты рисуются текущим цветом переднего плана. Вы можете изменить этот цвет вызовом метода `setColor()` класса `Graphics`:

```
void setColor(Color newColor)
```

Здесь *newColor* специфицирует новый цвет рисования.

Вы можете получить текущий установленный цвет, вызвав метод `getColor()`, показанный ниже:

```
Color getColor()
```

Аплет, демонстрирующий цвета

Следующий аплет конструирует несколько цветов и рисует различные объекты, используя эти цвета:

```
// Демонстрация цветов.
import java.awt.*;
import java.applet.*;
/*
<applet code="ColorDemo" width=300 height=200>
</applet>
*/
public class ColorDemo extends Applet {
// рисование линий
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 (paint mode) определяет то, как рисуются объекты в окне. По умолчанию новый вывод в окно перекрывает любое существующее содержимое. Однако можно иметь новые объекты, объединенные операцией XOR с предыдущим содержимым окна, используя `setXORMode()`, как показано ниже:

```
void setXORMode(Color xorColor)
```

Здесь *xorColor* специфицирует цвет, который будет соединен операцией XOR с содержимым окна во время рисования. Преимущество режима XOR в том, что новый объект всегда будет гарантированно видимым, независимо от того, какого цвета был объект, нарисованный ранее.

Чтобы вернуться к методу рисования с перекрытием, вызовите метод `setrPaintMode()`, показанный ниже:

```
void setPaintMode()
```

Обычно метод перекрытия используется для нормального вывода, а режим XOR — для специальных целей. Например, следующая программа отображает крестообразный "прицел", отлеживающий курсор мыши. Крест отображается рисованием в режиме XOR в поле окна, и потому всегда видим, независимо от цвета фона.

```
// Демонстрация применения режима XOR.
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() {
  addMouseMotionListener(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);

    // xor крестообразный "прицел"
    g.setXORMode(Color.black);
    g.drawLine(chsX-10, chsY, chsX+10, chsY);
    g.drawLine(chsX, chsY-10, chsX, chsY+10);
    g.setPaintMode();
  }
}
```

Пример вывода этого аплета показан на рис. 23.10.

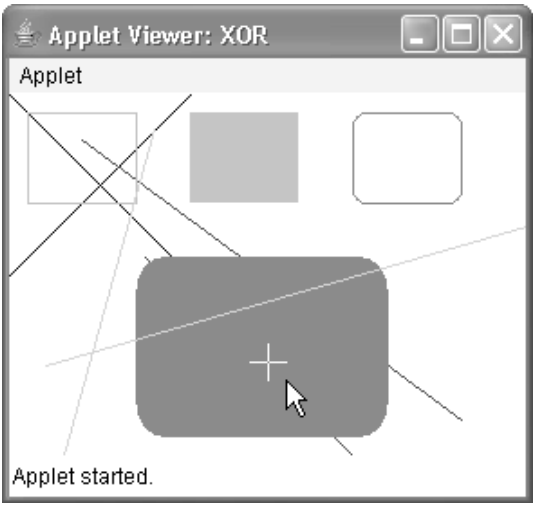

Рис. 23.10. Пример применения режима XOR

Работа со шрифтами

AWT поддерживает множество типов шрифтов. В прошлые годы шрифты, пришедшие из области традиционной печати, стали важной частью генерируемых компьютерами документов и дисплеев. Гибкость AWT обеспечивается за счет абстрагирования операций манипулирования шрифтами и возможности динамического выбора шрифтов.

Шрифты имеют имя семейства, логическое имя шрифта и название гарнитуры. *Имя семейства* — обобщенное имя шрифта, такое как Courier. *Логическое имя* специфицирует категорию шрифтов вроде Monospaced. *Гарнитура* специфицирует конкретный шрифт, такой как Courier Italic.

Шрифты инкапсулируются классом `Font`. Несколько методов, определенных в классе `Font`, перечислено в табл. 23.2.

Таблица 23.2. Методы, определенные в классе `Font`

Метод	Описание
`static Font decode(String str)`	Возвращает шрифт по заданному имени.
`boolean equals(Object FontObj)`	Возвращает `true`, если вызывающий объект содержит тот же шрифт, что и указанный в `FontObj`.
`String getFamily()`	Возвращает имя семейства шрифтов, к которому относится вызывающий шрифт.
`static Font getFont(String property)`	Возвращает шрифт, ассоциированный с системным свойством, указанным в `property`. Если свойство `property` не существует, возвращается `null`.
`static Font getFont(String property, Font defaultFont)`	Возвращает шрифт, ассоциированный с системным свойством, указанным в `property`. Если свойство `property` не существует, возвращается `defaultFont`.
`String getFontName()`	Возвращает название гарнитуры вызывающего шрифта.
`String getName()`	Возвращает логическое имя вызывающего шрифта.
`int getSize()`	Возвращает размер в точках вызывающего шрифта.
`int getStyle()`	Возвращает значения стиля вызывающего шрифта.
`int hashCode()`	Возвращает хеш-код, ассоциированный с вызывающим объектом.
`boolean isBold()`	Возвращает `true`, если шрифт включает значение стиля `BOLD`. В противном случае возвращается `false`.
`boolean isItalic()`	Возвращает `true`, если шрифт включает значение стиля `ITALIC`. В противном случае возвращается `false`.
`boolean isPlain()`	Возвращает `true`, если шрифт включает значение стиля `PLAIN`. В противном случае возвращается `false`.
`String toString()`	Возвращает строковый эквивалент вызывающего шрифта.

В классе `Font` определены переменные, которые описаны в табл. 23.3.

Таблица 23.3. Переменные, определенные в классе `Font`

Переменная	Значение
`String name`	Имя шрифта.
`float pointSize`	Размер шрифта в точках.
`int size`	Размер шрифта в точках.
`int style`	Стиль шрифта.

Определение доступных шрифтов

Работая со шрифтами, часто нужно знать, какие шрифты присутствуют на конкретной машине. Для получения этой информации служит метод `getAvailableFontFamilyNames()`, определенный в классе `GraphicsEnvironment`. Выглядит он так:

```
String[] getAvailableFontFamilyNames()
```

Этот метод возвращает массив строк, содержащих имена доступных семейств шрифтов. В дополнение к нему в классе `GraphicsEnvironment` определен метод `getAllFonts()`, показанный ниже:

```
Font[] getAllFonts()
```

Этот метод возвращает массив объектов `Font`, описывающих все доступные шрифты.

Поскольку эти методы являются членами `GraphicsEnvironment`, для их вызова вам понадобится ссылка на объект этого класса. Вы можете получить эту ссылку, используя статический метод `getLocalGraphicsEnvironment()`, который определен в `GraphicsEnvironment`. Выглядит он так:

```
static GraphicsEnvironment getLocalGraphicsEnvironment()
```

Рассмотрим пример аплета, который показывает, как получить имена всех доступных семейств шрифтов.

```
// Отображение шрифтов.
/*
<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);
}
}
```

Примерный вывод этого аплета показан на рис. 23.11. Однако список шрифтов у вас может отличаться от показанного на рисунке.

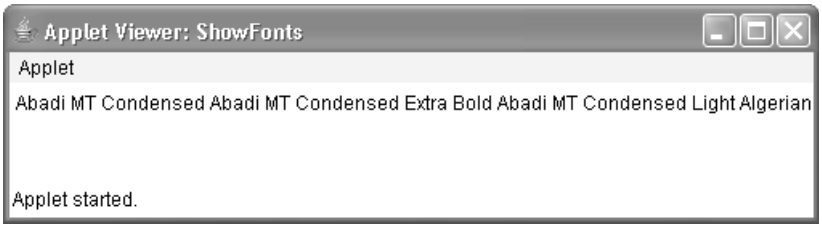

Рис. 23.11. Пример получения списка доступных шрифтов

Создание и выбор шрифта

Чтобы выбрать новый шрифт, вы должны сначала сконструировать объект `Font`, описывающий шрифт. Один конструктор `Font` имеет следующую общую форму:

```
Font(String fontName, int fontStyle, int pointSize)
```

Здесь *fontName* специфицирует имя желаемого шрифта. Имя может быть специфицировано с использованием либо логического имени, либо названия гарнитуры. Все среды Java поддерживают следующие шрифты: Dialog, DialogInput, Sans Serif и Monospaced. Dialog — шрифт, используемый в диалоговых окнах вашей системы. Dialog также является шрифтом по умолчанию, когда никакой другой не установлен явно. Также вы можете использовать любой другой шрифт, который поддерживает ваша конкретная среда, однако будьте осторожны — эти другие шрифты могут быть не всегда доступны.

Стиль шрифта, указан в *fontStyle*. Он может состоять из одной или более следующих констант: `Font.PLAIN`, `Font.BOLD` и `Font.ITALIC`. Чтобы скомбинировать стили, объединяйте их с помощью операции "ИЛИ". Например, `Font.BOLD` и `Font.ITALIC` специфицируют полужирный курсив.

Размер шрифта в точках указывается в *pointSize*.

Чтобы использовать только что созданный шрифт, вы должны выбрать его с помощью `setFont()` — метода, определенного в `Component`. Вот его общая форма:

```
void setFont(Font fontObj)
```

Здесь *fontObj* — объект, содержащий желаемый шрифт.

Следующая программа выводит примеры каждого из стандартных шрифтов. Всякий раз, когда вы щелкаете кнопкой мыши внутри окна, выбирается новый шрифт и отображается его имя.

```
// Отображение шрифтов.
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) {
  // Переключать шрифты с каждым щелчком кнопкой мыши.
  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 вывода этого аплета показан на рис. 23.12.

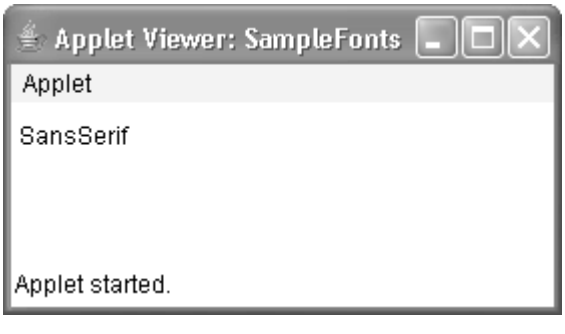

Рис. 23.12. Пример отображения шрифтов

Получение информации о шрифте

Предположим, что вы хотите получить информацию о текущем выбранном шрифте. Для этого вы должны сначала получить текущий шрифт вызовом `getFont()`. Этот метод определен в классе `Graphics`, как показано ниже:

```
Font getFont()
```

Получив текущий выбранный шрифт, вы можете извлечь информацию о нем с использованием различных методов, определенных в `Font`. Например, следующий аплет отображает имя, семейство, размер и стиль текущего выбранного шрифта.

```
// Отображение информации 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;
  // msg += ", Шрифт: " + fontFamily;

  msg += ", Size: " + fontSize + ", Style: ";
  // msg += ", Размер: " + fontSize + ", Style: ";

  if((fontStyle & Font.BOLD) == Font.BOLD)
     msg += "Bold ";
     // msg += "Полужирный ";

  if((fontStyle & Font.ITALIC) == Font.ITALIC)
     msg += "Italic ";
     // msg += "Курсив ";

  if((fontStyle & Font.PLAIN) == Font.PLAIN)
     msg += "Plain ";
     // msg += "Обычный ";

  g.drawString(msg, 4, 16);

}
}
```

Управление выводом текста с использованием класса FontMetrics

Как только что было сказано, Java поддерживает множество шрифтов. Для большинства их них символы не имеют одинакового размера — большинство шрифтов пропорциональны. Высота каждого символа, длина *подстрочных элементов*, таких как нижняя часть некоторых букв, например, *y*, а также промежуток между горизонтальными линиями варьируются от шрифта к шрифту. Более того, размер шрифта в точках также может изменяться. Все эти (и другие) атрибуты являются переменными и не представляют особого интереса для вас, как программиста, поскольку Java не требует ручного управления почти всем текстовым выводом.

Учитывая, что размер каждого шрифта может отличаться, и что шрифты могут изменяться в процессе выполнения вашей 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. Чтобы позволить это, AWT предусматривает класс `FontMetrics`, инкапсулирующий разнообразную информацию о шрифте. Начнем с определения общей терминологии, используемой при описании шрифтов (табл. 23.4).

Таблица 23.4. Общая терминология, используемая при описании шрифтов

Высота (height)	Размер строки текста сверху вниз.
Базовая линия (baseline)	Строка, по которой выровнены нижние грани символов (за исключением подстрочных элементов).
Возвышение (ascent)	Расстояние от базовой линии до вершины символов.
Спуск (descent)	Расстояние от базовой линии до нижней точки символов.
Межстрочный интервал (leading)	Расстояние между нижней точкой строки текста и ее верхней точкой.

Как вы наверняка заметили, метод `drawString()` использовался во многих предыдущих примерах. Он рисует строку в текущем цвете и шрифте, начиная с указанного местоположения. Однако это местоположение находится на левой грани базовой линии символов, а не в верхней левой точке, как это принято в других методах рисования. Часто допускается ошибка, заключающаяся в попытке рисовать строку по тем же координатам, где вы рисовали бы рамку. Например, если вам нужно нарисовать прямоугольник, начиная с координаты (0,0), то вы увидите полный прямоугольник. Если же вы попытаетесь вывести строку "Typesetting", начиная с координаты (0,0), то увидите только подстрочные элементы букв *y*, *p* и *g*. Как вы убедитесь далее, используя метрики шрифта, можно определить правильное местоположение каждой строки, подлежащей отображению.

`FontMetrics` определяет несколько методов, которые помогают вам управлять текстовым выводом. Несколько наиболее часто используемых перечислены в табл. 23.5. Эти методы помогают правильно отобразить текст в окне. Рассмотрим некоторые примеры.

Таблица 23.5. Выборка методов, определенных в `FontMetrics`

Метод	Описание
`int bytesWidth(byte b[], int start, int numBytes)`	Возвращает ширину *numBytes* символов, содержащихся в массиве *b*, начиная со *start*.
`int charWidth(char c[], int start, int numChars)`	Возвращает ширину *numChars* символов, содержащихся в массиве *c*, начиная со *start*.
`int charWidth(char c)`	Возвращает ширину *c*.
`int charWidth(int c)`	Возвращает ширину *c*.
`int getAscent()`	Возвращает возвышение шрифта.
`int getDescent()`	Возвращает спуск шрифта.
`Font getFont()`	Возвращает текущий шрифт.
`int getHeight()`	Возвращает высоту строки текста. Это значение может быть использовано для вывода множества строк текста в окно.
`int getLeading()`	Возвращает пробел между строками текста.
`int getMaxAdvance()`	Возвращает ширину наиболее широкого символа. Если это значение недоступно, возвращается `-1`.
`int getMaxAscent()`	Возвращает максимальное возвышение.
`int getMaxDescent()`	Возвращает максимальный спуск.
`int[] getWidths()`	Возвращает ширину первых 256 символов.
`int stringWidth(String str)`	Возвращает ширину строки, указанной в *str*.
`String toString()`	Возвращает строковый эквивалент вызывающего объекта.

Отображение множества строк текста

Возможно, наиболее часто `FontMetrics` используется для определения расстояния между строками текста. Вторым по частоте применением является определение длины строки, которую нужно отобразить. Ниже вы увидите, как решаются эти задачи.

В общем случае для того, чтобы отобразить многострочный текст, ваша программа должна отслеживать текущую позицию вывода. Всякий раз, когда встречается символ новой строки, координата Y должна быть перенесена в начало следующей строки. Всякий раз, когда отображается строка, координата X должна быть установлена в точку, где эта строка заканчивается. Это позволит начать вывод следующей строки так, чтобы она начиналась сразу после окончания предыдущей.

Чтобы определить расстояние между строками, вы можете использовать значение, возвращенное `getLeading()`. Чтобы определить общую высоту шрифта, добавьте значение, возвращенное `getAscent()`, к значению, возвращенному `getDescent()`. Затем вы можете использовать эти значения для позиционирования каждой строки текста, подлежащего выводу. Однако во многих случаях вам не понадобятся эти значения по отдельности. Часто все, что вам нужно знать — это общая высота строки, представляющая сумму межстрочного интервала со значениями подъема и спуска шрифта. Самый простой путь получения этого значения — вызвать `getHeight()`. Просто увеличивайте координату Y на эту величину всякий раз, когда хотите перейти на следующую строку выводимого текста.

Чтобы начать вывод с конца предыдущего вывода в той же строке, вы должны знать длину в пикселях каждой строки, которую отображаете. Чтобы получить это значение, вызывайте `stringWidth()`. Вы можете использовать это значение для вычисления координаты X всякий раз, когда выводите строку.

В следующем аплете показано, как вывести многострочный текст в окно. Он также отображает несколько предложений на одной и той же строке. Обратите внимание на переменные `curX` и `curY`. Они отслеживают текущую позицию вывода текста.

```
// Демонстрация многострочного вывода.
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);
  nextLine("This is on line two.", g);
  sameLine(" This is on same line.", g);
  sameLine(" This, too.", g);

  nextLine("This is on line three.", g);
  // nextLine("Это в строке один.", g);
  // nextLine("Это в строке два.", g);
  // sameLine(" Это в той же строке.", g);
  // sameLine(" И это тоже.", g);
  // nextLine("Это в строке три.", g);
}

// Переход на следующую строку.
void nextLine(String s, Graphics g) {
  FontMetrics fm = g.getFontMetrics();
  curY += fm.getHeight(); // переход на следующую строку
  curX = 0;
  g.drawString(s, curX, curY);
  curX = fm.stringWidth(s); // переход к концу строки
}

// Отображение в той же строке.
void sameLine(String s, Graphics g) {
  FontMetrics fm = g.getFontMetrics();
  g.drawString(s, curX, curY);
  curX += fm.stringWidth(s); // переход к концу строки
}
}
```

Примерный вывод этого аплета показан на рис. 23.13.

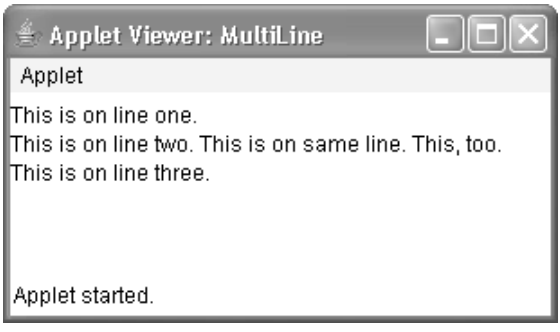

Рис. 23.13. Пример многострочного вывода

Центрирование текста

Рассмотрим пример, центрирующий текст в окне слева направо и сверху вниз. Он получает возвышение, спуск и ширину строки и вычисляет позицию, в которой он должен быть отображен для центрирования.

```
// Центрирование 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);
  // drawCenteredString("Центрировано.", d.width, d.height, g);
  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);
}
}
```

На рис. 23.14 показан пример вывода этого аплета.

Рис. 23.14. Пример центрирования текста

Выравнивание многострочного текста

Когда вы используете текстовый процессор, то обычно видите текст выровненным таким образом, что одна или обе его стороны образуют ровную вертикальную линию. Например, большинство текстовых процессоров могут выравнивать текст по левому краю и/или по правому краю. Большинство также могут центрировать текст. В следующей программе вы увидите, как добиться этого эффекта.

В этой программе строки, подлежащие выравниванию, разбиваются на слова. Для каждого слова программа отслеживает его длину в текущем шрифте и автоматически переходит на следующую строку, когда слово не умещается в текущей. Каждая завершенная строка отображается в окне с текущим выбранным стилем выравнивания. Всякий раз, когда вы щелкаете кнопкой мыши в окне аплета, стиль выравнивания изменяется. Пример вывода этого аплета показан на рис. 23.15.

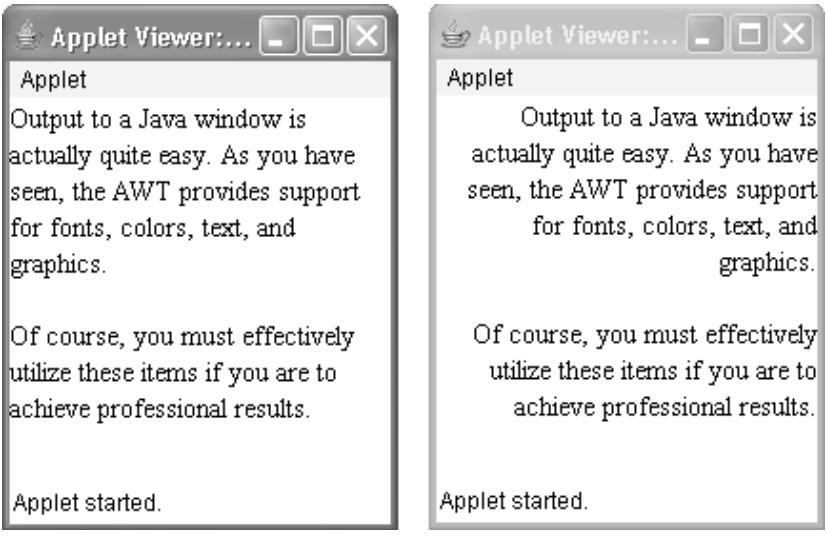

Рис. 23.15. Пример выравнивания текста

```
// Демонстрация выравнивания 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(align) {
     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 + 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();
}
}
```

Давайте разберемся, как работает этот аплет. Сначала в нем создаются несколько констант, которые будут использоваться для определения стиля выравнивания, затем объявляются несколько переменных. Метод `init()` получает текст, подлежащий отображению. Далее он инициализирует размер шрифта в блоке `try-catch`, который затем установит размер шрифта равным 14, если параметр `fontSize` опущен в HTML-дескрипторе. Параметр `text` представляет собой длинную строку текста с HTML-дескриптором `<P>` в качестве разделителя абзаца.

Метод `update()` — "двигатель" нашего примера. Он устанавливает шрифт и получает базовую линию и высоту из объекта метрик шрифта. Далее он создает `StringTokenizer` и использует его для извлечения следующей лексемы (строки, отделенной пробелами) из строки, специфицированной в `text`. Если следующей лексемой является `<P>`, осуществляется вертикальная прокрутка. В противном случае `update()` проверяет, вписывается ли длина лексемы в текущем шрифте в ширину колонки. Если строка уже заполнена текстом или лексем для вывода более не осталось, строка выводится специальной версией `drawString()`. Первые три случая `drawString()` просты. Каждый выравнивает строку, переданную в `line`, по левому или правому краю или же по центру колонки, в зависимости от текущего стиля выравнивания. В случае `LEFTRIGHT` выравниваются обе стороны строки. Это значит, что необходимо вычислить оставшееся пространство (как разницу между шириной строки и шириной колонки) и распределить его между словами. Последний метод класса переключает стиль выравнивания при каждом щелчке кнопкой мыши в окне аплета.

24
ГЛАВА

Использование элементов управления, диспетчеров компоновки и меню AWT

В этой главе исследование AWT продолжается. Сначала в ней будут рассмотрены стандартные элементы управления и диспетчеры компоновки. Затем речь пойдет о меню и линейке меню. Мы поговорим о двух компонентах верхнего уровня: диалоговом окне и диалоговом окне выбора файла. В конце главы будет предложен другой взгляд на вопрос обработки событий.

Элементами управления называются компоненты, которые позволяют пользователю взаимодействовать с вашим приложением различными способами — например, наиболее распространенным элементом управления является экранная кнопка. *Диспетчер компоновки* автоматически позиционирует компоненты внутри контейнера. Поэтому внешний вид окна зависит как от элементов управления, которые оно содержит, так и от диспетчера компоновки, используемого для их позиционирования.

Кроме элементов управления, обрамляющее окно может также включать *линейку меню* стандартного стиля. Каждый пункт в линейке меню раскрывает меню, в котором пользователь может выбрать необходимую ему команду. Линейка меню всегда располагается в верхней части окна. Несмотря на различие во внешнем виде, линейки меню обрабатываются почти так же, как и другие элементы управления.

Несмотря на то что позиционировать компоненты в окне можно вручную, сделать это обычно бывает непросто. Диспетчер компоновки выполняет эту задачу автоматически. В первой части этой главы, в которой рассматриваются различные элементы управления, используется диспетчер компоновки, принятый по умолчанию. Он отображает компоненты в контейнере, размещая их слева направо и сверху вниз. После того как мы рассмотрим элементы управления, мы поговорим о диспетчерах компоновки. Вы узнаете, как лучше всего управлять позиционированием элементов управления.

Основы элементов управления

AWT поддерживает следующие типы элементов управления:

- метки;
- экранные кнопки;
- флажки;
- списки выбора;
- списки;
- линейки прокрутки;
- элементы редактирования текста.

Все эти элементы управления относятся к подклассам класса `Component`.

Добавление и удаление элементов управления

Чтобы включить элемент управления в окно, его нужно сначала добавить к нему. Для этого необходимо создать экземпляр требуемого элемента управления, после чего добавить его в окно с помощью метода `add()`, который определен классом `Container`. Метод `add()` имеет несколько форм. В первой части этой главы используется следующая форма:

```
Component add(Component compObj)
```

Здесь параметр *compObj* представляет экземпляр элемента управления, который вы хотите добавить. Метод возвращает ссылку на *compObj*. После того как элемент управления будет добавлен, он будет отображаться при отображении его родительского окна.

Иногда бывает необходимо удалить элементы управления из окна. Для этого служит метод `remove()`. Он тоже определен в классе `Container`. Метод имеет следующую общую форму:

```
void remove(Component obj)
```

Здесь параметр *obj* представляет ссылку на элемент управления, который вы хотите удалить. Можно удалить все элементы управления, если вызвать метод `removeAll()`.

Реагирование на действия, производимые над элементами управления

За исключением меток, которые являются пассивными элементами, каждый элемент управления генерирует событие в тот момент, когда к нему обращается пользователь. Например, когда пользователь щелкает на экранной кнопке, посылается событие, идентифицирующее эту кнопку. В общем случае ваша программа просто реализует соответствующий интерфейс, после чего регистрирует слушателя события для каждого элемента управления, за которым вы хотите вести наблюдение. Как было сказано в главе 22, если установить слушатель, то события будут посылаться ему автоматически. В последующих разделах для каждого элемента управления будет описан соответствующий интерфейс.

HeadlessException

Большинство элементов управления AWT, рассматриваемых в этой главе, теперь имеют конструкторы, способные генерировать исключение `HeadlessException` при попытке создать экземпляр компонента GUI в неинтерактивной среде (т.е. в среде, в которой, например, нет монитора, мыши или клавиатуры). Исключение `HeadlessException` было добавлено в Java 1.4. С помощью этого исключения можно написать код, который будет адаптироваться к неинтерактивным средам. (Естественно, делать это можно не всегда.) Это исключение не обрабатывается программами в этой главе, так как для иллюстрации элементов управления AWT нужна интерактивная среда.

Метки

Самым простым элементом управления является метка. *Метка* представляет собой объект типа `Label` и содержит строку, которую она отображает. Метки являются пассивными элементами управления, не поддерживающими никакого взаимодействия с пользователем. `Label` определяет следующие конструкторы:

```
Label() throws HeadlessException
Label(String str) throws HeadlessException
Label(String str, int how) throws HeadlessException
```

В первом варианте создается пустая метка. Во втором варианте создается метка, содержащая строку *`str`*. Эта строка выравнивается по левому краю. В третьем варианте создается метка, содержащая строку *`str`*, выравнивание которой определяется с помощью параметра *`how`*. Параметр *`how`* должен принимать одну из следующих трех констант: `Label.LEFT`, `Label.RIGHT` или `Label.CENTER`.

Изменить состояние текста в метке можно с помощью метода `setText()`. Получить текущую метку можно с помощью метода `getText()`. Эти методы показаны далее:

```
void setText(String str)
String getText()
```

В методе `setText()` параметр *`str`* определяет новую метку. Метод `getText()` возвращает текущую метку.

Выровнять строку в метке можно с помощью метода `setAlignment()`. Чтобы получить текущее выравнивание, используется метод `getAlignment()`. Эти методы показаны ниже:

```
void setAlignment(int how)
int getAlignment()
```

Здесь параметр *`how`* должен представлять одну из трех приведенных выше констант.

В следующем примере создаются три метки, которые затем добавляются в аплет.

```
// Демонстрация 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");

    // добавление меток в окно аплета
    add(one);
    add(two);
    add(three);
  }
}
```

На рис. 24.1 показано окно, созданное аплетом `LabelDemo`. Обратите внимание на то, что упорядочение меток в окне выполнено посредством диспетчера компоновки, используемого по умолчанию. Позже вы увидите, каким образом можно более точно управлять размещением меток.

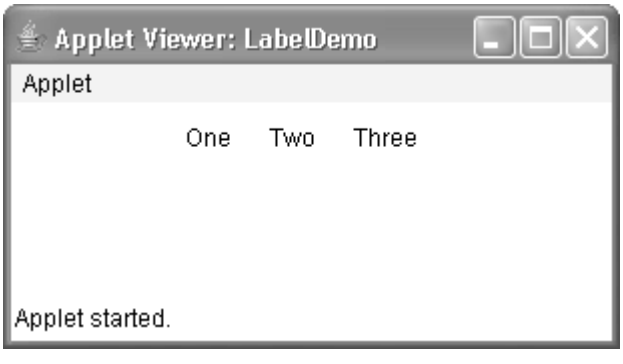

Рис. 24.1. Окно аплета `LabelDemo`

Использование кнопок

Пожалуй, наиболее широко используемым элементом управления является экранная кнопка. *Экранная кнопка* — это компонент, который содержит метку и генерирует событие, когда пользователь щелкает на нем кнопкой мыши. Экранные кнопки являются объектами типа `Button`. `Button` определяет следующие конструкторы:

```
Button() throws HeadlessException
Button(String str) throws HeadlessException
```

В первом варианте создается пустая кнопка. Во втором варианте создается кнопка с меткой *`str`*.

После того как кнопка будет создана, с помощью метода `setLabel()` можно определить ее метку. Получить метку можно с помощью метода `getLabel()`. Эти методы показаны ниже:

```
void setLabel(String str)
String getLabel()
```

Здесь параметр *`str`* представляет новую метку для кнопки.

Обработка кнопок

Когда пользователь щелкает на кнопке, возникает событие действия. Оно посылается любому слушателю, который предварительно зарегистрировался на получение уведомлений о событиях действия от данного компонента. Каждый слушатель реализует интерфейс `ActionListener`. Этот интерфейс определяет метод `actionPerformed()`, который вызывается во время события. В качестве параметра для этого метода указывается объект `ActionEvent`. Он содержит ссылку на кнопку, сгенерировавшую событие, и ссылку на *командную строку действия*, связанную с этой кнопкой. По умолчанию командной строкой действия является метка кнопки. Как правило, для идентификации кнопки используется или ссылка на кнопку, или командная строка действия. (Скоро вы увидите примеры каждого из этих подходов.)

Ниже показан пример создания трех кнопок с метками “Yes” (Да), “No” (Нет) и “Undecided” (Нет решения). Когда пользователь щелкает на одной из кнопок, отображается сообщение, свидетельствующее о том, какая кнопка была нажата. В этом варианте команда действия кнопки (которая по умолчанию является ее меткой) используется для определения, какая из кнопок была нажата. Получение метки производится посредством вызова метода `getActionCommand()` в объекте `ActionEvent`, который передается методу `actionPerformed()`.

```
// Пример 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.";
      // msg = "Нажата кнопка Yes.";
    }
    else if(str.equals("No")) {
      msg = "You pressed No.";
      // msg = "Нажата кнопка No.";
    }
```

```
    else {
      msg = "You pressed Undecided.";
      // msg = "Нажата кнопка Undecided.";
    }
    repaint();
  }
  public void paint(Graphics g) {
    g.drawString(msg, 6, 100);
  }
}
```

Окно функционирующего аплета `ButtonDemo` можно видеть на рис. 24.2.

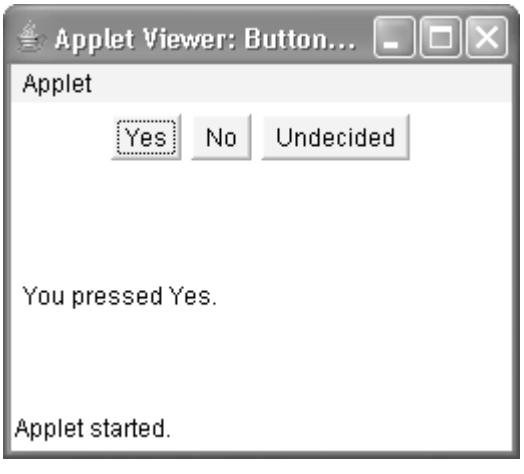

Рис. 24.2. Окно аплета `ButtonDemo`

Как уже было сказано, помимо сравнения командных строк действия кнопок можно также определить, какая из кнопок была нажата, если сравнить объект, полученный из метода `getSource()`, с объектами кнопок, которые вы добавляете в окно. Для этого вы должны вести список добавляемых объектов. Этот подход отражен в следующем аплете.

```
// Распознавание объектов 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");
```

```
    // сохранение ссылок на кнопки при добавлении
    bList[0] = (Button) add(yes);
    bList[1] = (Button) add(no);
    bList[2] = (Button) add(maybe);

    // регистрация 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();
        // msg = "Нажата кнопка " + bList[i].getLabel();
      }
    }
    repaint();
  }

  public void paint(Graphics g) {
    g.drawString(msg, 6, 100);
  }
}
```

В этом варианте при добавлении кнопок в окно аплета программа записывает в массив ссылку на каждую кнопку. (Вспомните, что метод `add()` возвращает ссылку на кнопку при ее добавлении.) Впоследствии этот массив используется в методе `actionPerformed()`, чтобы определить, какая кнопка была нажата.

В простых аплетах распознать кнопки по их меткам обычно бывает несложно. Однако если вы будете изменять метку внутри кнопки во время выполнения или использовать кнопки с одинаковыми метками, то определить, какая кнопка была нажата, можно будет очень легко, если воспользоваться ее объектной ссылкой. Можно также присвоить строке команды действия, связанной с кнопкой, что-нибудь отличное от ее метки, вызвав метод `setActionCommand()`. Он изменяет строку команды действия, не влияя на строку, используемую в качестве метки кнопки. Таким образом, если задать строку команды действия, можно отделить друг от друга команду действия и метку кнопки.

Использование флажков

Флажок представляет собой элемент управления, который служит для включения или отключения опции. Он состоит из небольшого окна, которое может либо содержать отметку (в виде “галочки”), либо быть пустым. У каждого флажка есть метка, описывающая опцию, которую представляет флажок. При щелчке на флажке можно изменить состояние флажка. Флажки могут использоваться как индивидуально, так и в качестве части группы. Флажки являются объектами класса `Checkbox`.

Класс `Checkbox` поддерживает следующие конструкторы:

```
Checkbox() throws HeadlessException
Checkbox(String str) throws HeadlessException
Checkbox(String str, boolean on) throws HeadlessException
Checkbox(String str, boolean on, CheckboxGroup cbGroup) throws HeadlessException
Checkbox(String str, CheckboxGroup cbGroup, boolean on) throws HeadlessException
```

Первая форма конструктора создает флажок, метка которого изначально является пустой. Флажок находится в неотмеченном состоянии. Вторая форма создает флажок, чья метка определяется параметром *str*. Флажок находится в неотмеченном состоянии. Третья форма позволяет установить исходное состояние флажка. Если параметр *on* равен `true`, то флажок изначально будет отмечен, в противном случае — нет. Четвертая и пятая формы создают флажок, метка которого определяется параметром *str*, а группа — параметром *cbGroup*. Если флажок не является частью группы, то параметр *cbGroup* должен иметь значение `null`. (Группы флажков описываются в следующем разделе.) Значение параметра *on* определяет исходное состояние флажка.

Чтобы получить текущее состояние флажка, используется метод `getState()`. Для определения его состояния используется метод `setState()`. С помощью метода `getLabel()` можно получить текущую метку, связанную с флажком. Чтобы задать метку, нужно вызвать метод `setLabel()`. Эти методы показаны ниже:

```
boolean getState()
void setState(boolean on)
String getLabel()
void setLabel(String str)
```

Здесь если параметр *on* равен `true`, то флажок будет помечен. Если он равен `false`, флажок не будет помечен. Строка, передаваемая в параметре *str*, становится новой меткой, связанной с вызываемым флажком.

Обработка флажков

Каждый раз, когда флажок отмечается или когда отметка снимается, генерируется событие элемента. Оно посылается любому слушателю, который ранее зарегистрировался на получение уведомлений о событиях элемента от данного компонента. Каждый слушатель реализует интерфейс `ItemListener`. Этот интерфейс определяет метод `itemStateChanged()`. Объект `ItemEvent` задается в качестве параметра для данного метода. Он содержит информацию о событии (например, выбор или отмена выбора).

В следующей программе создаются четыре флажка. Первый флажок изначально отмечен. Состояние каждого флажка отображается. Каждый раз при изменении состояния флажка производится обновление отображаемого состояния.

```
// Демонстрация применения флажков.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="CheckboxDemo" width=250 height=200>
  </applet>
*/

public class CheckboxDemo extends Applet implements ItemListener {
  String msg = "";
  Checkbox winXP, winVista, solaris, mac;

  public void init() {
    winXP = new Checkbox("Windows XP", null, true);
    winVista = new Checkbox("Windows Vista");
    solaris = new Checkbox("Solaris");
    mac = new Checkbox("Mac OS");
```

```
    add(winXP);
    add(winVista);
    add(solaris);
    add(mac);

    winXP.addItemListener(this);
    winVista.addItemListener(this);
    solaris.addItemListener(this);
    mac.addItemListener(this);
  }

  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }

  // Отображение текущего состояния флажков.
  public void paint(Graphics g) {
    msg = "Current state: ";
    // msg = "Текущее состояние: ";
    g.drawString(msg, 6, 80);
    msg = " Windows XP: " + winXP.getState();
    g.drawString(msg, 6, 100);
    msg = " Windows Vista: " + winVista.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);
  }
}
```

Окно функционирующего аплета показано на рис. 24.3.

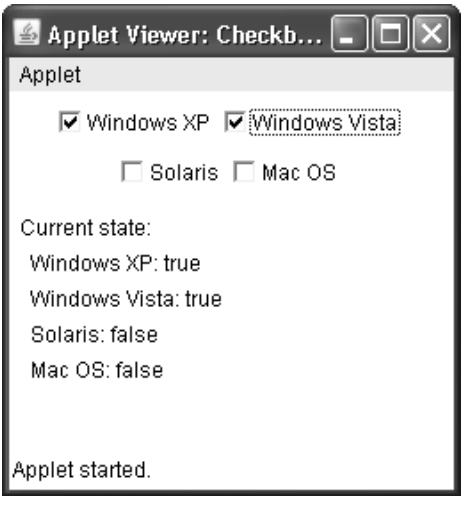

Рис. 24.3. Окно аплета `CheckboxDemo`

CheckboxGroup

Допускается создание набора взаимно исключающих флажков, в котором одновременно может быть отмечен один и только один флажок. Эти флажки часто называются *переключателями*, потому что они работают подобно переключателю каналов в автомобильном радиоприемнике — одновременно можно выбрать только одну радиостанцию. Чтобы создать набор взаимоисключающих кнопок, вы должны вначале определить группу, к которой они принадлежат, а затем определить эту группу при конструировании флажков. Группы флажков представляют собой объекты типа `CheckboxGroup`. Определен только конструктор по умолчанию, создающий пустую группу.

Чтобы узнать, какой флажок в группе отмечен на данный момент, вызовите метод `getSelectedCheckBox()`.

Отметить флажок можно с помощью метода `setSelectedCheckbox()`. Эти методы показаны далее:

```
Checkbox getSelectedCheckbox()
void setSelectedCheckbox(Checkbox which)
```

Здесь параметр *which* представляет флажок, который вы хотите выбрать. Ранее выбранный флажок будет отключен.

Ниже показан пример программы, в которой используются флажки, являющиеся частью группы.

```
// Демонстрация применения группы флажков.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="CBGroup" width=250 height=200>
  </applet>
*/

public class CBGroup extends Applet implements ItemListener {
  String msg = "";
  Checkbox winXP, winVista, solaris, mac;
  CheckboxGroup cbg;

  public void init() {
    cbg = new CheckboxGroup();
    winXP = new Checkbox("Windows XP", cbg, true);
    winVista = new Checkbox("Windows Vista", cbg, false);
    solaris = new Checkbox("Solaris", cbg, false);
    mac = new Checkbox("Mac OS", cbg, false);

    add(winXP);
    add(winVista);
    add(solaris);
    add(mac);

    winXP.addItemListener(this);
    winVista.addItemListener(this);
    solaris.addItemListener(this);
    mac.addItemListener(this);
  }

  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }
```

```
  // Отображение текущего состояния флажков.
  public void paint(Graphics g) {
    msg = "Current selection: ";
    // msg = "Текущий выбор: ";
    msg += cbg.getSelectedCheckbox().getLabel();
    g.drawString(msg, 6, 100);
  }
}
```

Окно аплета `CBGroup` во время выполнения показано на рис. 24.4. Обратите внимание на то, что в данном случае флажки имеют форму окружности.

Рис. 24.4. Окно аплета `CBGroup`

Элементы управления выбором

Класс `Choice` используется для создания *всплывающего списка* элементов, в котором пользователь может делать свой выбор. Поэтому элемент управления `Choice` является разновидностью меню. Будучи неактивным, компонент `Choice` занимает ровно столько пространства, сколько необходимо для отображения выбранного на данный момент элемента. Когда пользователь щелкает на нем, раскрывается весь список, в котором можно выбрать новый элемент. Каждый элемент в списке представляет собой строку, которая появляется в виде метки с выравниванием влево в том порядке, в каком он добавляется в объект `Choice`. `Choice` определяет только конструктор по умолчанию, который создает пустой список.

Чтобы добавить элемент выбора в список, нужно вызвать метод `add()`. Он имеет следующую общую форму:

```
void add(String name)
```

Здесь параметр *`name`* представляет добавляемый элемент. Добавление элементов в список производится в том порядке, в котором производятся вызовы метода `add()`.

Чтобы узнать, какой из элементов является выбранным на данный момент, можно вызвать один из двух методов: `getSelectedItem()` или `getSelectedIndex()`. Они показаны ниже:

```
String getSelectedItem()
int getSelectedIndex()
```

Метод `getSelectedItem()` возвращает строку, содержащую имя элемента. Метод `getSelectedIndex()` возвращает индекс элемента. Первый элемент имеет индекс `0`. По умолчанию выбранным является первый элемент, добавленный в список.

Чтобы узнать, сколько элементов содержится в списке, нужно вызвать метод `getItemCount()`. Указать, какой элемент будет выбран на данный момент, можно с помощью метода `select()`, используя целочисленный индекс, начинающийся с нуля, или строку, совпадающую с именем элемента в списке. Эти методы показаны ниже:

```
int getItemCount()
void select(int index)
void select(String name)
```

Для заданного индекса можно получить имя, связанное с элементом в этом индексе, если вызвать метод `getItem()`, который имеет следующую общую форму:

```
String getItem(int index)
```

Здесь параметр *index* представляет индекс требуемого элемента.

Обработка списков выбора

Каждый раз при выборе определенного элемента генерируется событие. Оно посылается всем слушателям, которые предварительно зарегистрировались на получение уведомлений о событиях элемента от данного компонента. Каждый слушатель реализует интерфейс `ItemListener`. Этот интерфейс определяет метод `itemStateChanged()`. В качестве параметра для этого метода задается объект `ItemEvent`.

Ниже показан пример создания двух меню `Choice`. В одном из них производится выбор операционной системы, в другом — браузера.

```
// Демонстрация применения списков выбора.
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();

    // добавление элементов в список os
    os.add("Windows XP");
    os.add("Windows Vista");
    os.add("Solaris");
    os.add("Mac OS");
```

```
    // добавление элементов в список browser
    browser.add("Internet Explorer");
    browser.add("Firefox");
    browser.add("Opera");

    // добавление списков выбора в окно
    add(os);
    add(browser);

    // регистрация на получение уведомлений о событиях
    os.addItemListener(this);
    browser.addItemListener(this);
  }

  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }

  // Отображение текущего выбора.
  public void paint(Graphics g) {
    msg = "Current OS: ";
    // msg = "Текущая ОС: ";
    msg += os.getSelectedItem();
    g.drawString(msg, 6, 120);
    msg = "Current Browser: ";
    // msg = "Текущий браузер: ";
    msg += browser.getSelectedItem();
    g.drawString(msg, 6, 140);
  }
}
```

Окно аплета во время выполнения показано на рис. 24.5.

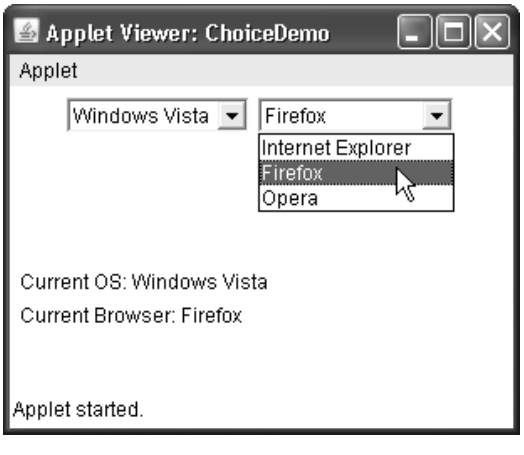

Рис. 24.5. Окно аплета `ChoiceDemo`

Использование списков

Класс `List` предлагает компактный прокручиваемый список, в котором можно выбирать множество элементов. В отличие от объекта `Choice`, который отображает только один выбранный элемент меню, объект `List` можно создать таким образом, чтобы он показывал любое количество элементов выбора в видимом окне. Его можно создать так, чтобы в нем можно было выбирать несколько элементов. Класс `List` предлагает следующие конструкторы:

```
List() throws HeadlessException
List(int numRows) throws HeadlessException
List(int numRows, boolean multipleSelect) throws HeadlessException
```

Первый вариант создает элемент управления `List`, который позволяет выбрать одновременно только один элемент. Во втором варианте значение параметра *`numRows`* определяет количество пунктов в списке, которые всегда будут видимыми (остальные пункты можно просмотреть, используя прокрутку). В третьем варианте, если параметр *`multipleSelect`* равен `true`, то пользователь может выбрать одновременно два или более пункта. Если же он будет равен `false`, то выбрать можно будет только один пункт.

Для добавления пункта в список вызовите метод `add()`. Он имеет два варианта:

```
void add(String name)
void add(String name, int index)
```

Здесь параметр *`name`* определяет имя элемента, добавленного в список. В первом случае добавляются пункты в конец списка. Во втором случае добавляется пункт в индекс, определяемый параметром *`index`*. Индексация начинается с нуля. Чтобы добавить пункт в конец списка, необходимо указать –1.

Для тех списков, в которых можно выбрать только один пункт, узнать, какой из пунктов выбран на данный момент, можно с помощью метода `getSelectedItem()` или `getSelectedIndex()`. Эти методы показаны ниже:

```
String getSelectedItem()
int getSelectedIndex()
```

Метод `getSelectedItem()` возвращает строку, содержащую имя пункта. Если выбрано несколько пунктов или если еще не был выбран ни один пункт, возвращается `null`. Метод `getSelectedIndex()` возвращает индекс пункта. Первый пункт имеет индекс 0. Если выбрано несколько пунктов или если еще не был выбран ни один пункт, возвращается `-1`.

Для списков, в которых можно выбирать несколько пунктов, узнать выбранные на данный момент пункты можно с помощью методов `getSelectedItems()` или `getSelectedIndexes()`, которые показаны ниже:

```
String[] getSelectedItems()
int[] getSelectedIndexes()
```

Метод `getSelectedItems()` возвращает массив, содержащий имена выбранных на данный момент пунктов. Метод `getSelectedIndexes()` возвращает массив, содержащий индексы выбранных на данный момент пунктов.

Чтобы узнать, сколько пунктов содержится в списке, вызовите метод `getItemCount()`. С помощью метода `select()` можно определить, какой пункт будет выбран на данный момент; для этих целей используется целочисленный индекс, начинающийся с нуля. Эти методы показаны ниже:

```
int getItemCount()
void select(int index)
```

Для данного индекса можно получить имя, связанное с пунктом в этом индексе, если вызвать метод `getItem()`, который имеет следующую общую форму:

```
String getItem(int index)
```

Здесь параметр *index* представляет индекс требуемого пункта.

Обработка списков

Для обработки событий списков необходимо реализовать интерфейс `ActionListener`. Каждый раз при двойном щелчке на пункте `List` генерируется объект `ActionEvent`. Его метод `getActionCommand()` можно использовать для получения имени нового выбранного пункта. Также каждый раз при выборе или отмене выбора пункта посредством одиночного щелчка генерируется объект `ItemEvent`. Его метод `getStateChanged()` можно использовать, чтобы узнать, чем было вызвано данное событие: выбором пункта или отменой его выбора. Метод `getItemSelectable()` возвращает ссылку на объект, инициировавший данное событие.

Ниже показан пример, который преобразовывает элементы управления `Choice` из предыдущего раздела в компоненты `List`, один для множественного выбора, а другой — для одиночного выбора:

```
// Демонстрация применения 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);

    // добавление элементов в список os
    os.add("Windows XP");
    os.add("Windows Vista");
    os.add("Solaris");
    os.add("Mac OS");

    // добавление элементов в список browser
    browser.add("Internet Explorer");
    browser.add("Firefox");
    browser.add("Opera");
    browser.select(1);

    // добавление списков в окно
    add(os);
    add(browser);

    // регистрация на получение уведомлений о событиях действия
    os.addActionListener(this);
```

```
    browser.addActionListener(this);
  }

  public void actionPerformed(ActionEvent ae) {
    repaint();
  }

  // Отображение текущего выделения.
  public void paint(Graphics g) {
    int idx[];

    msg = "Current OS: ";
    idx = os.getSelectedIndexes();
    for(int i=0; i<idx.length; i++)
      msg += os.getItem(idx[i]) + " ";
    g.drawString(msg, 6, 120);
    msg = "Current Browser: ";
    msg += browser.getSelectedItem();
    g.drawString(msg, 6, 140);
  }
}
```

Аплет `ListDemo` во время выполнения показан на рис. 24.6.

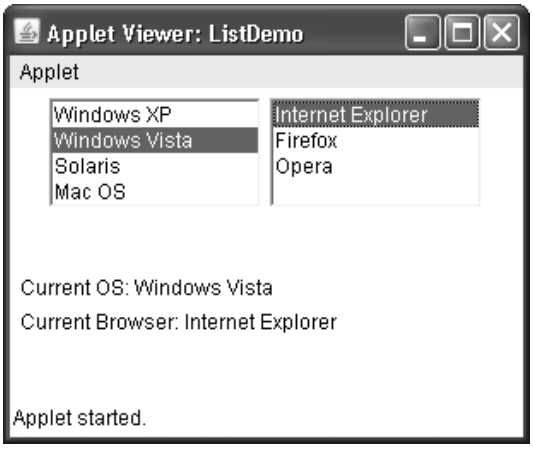

Рис. 24.6. Окно аплета `ListDemo`

Управление линейками прокрутки

Линейки прокрутки используются для выбора непрерывных значений между заданным минимальным и максимальным значениями. Линейки прокрутки могут быть ориентированы вертикально и горизонтально. В действительности линейка прокрутки состоит из нескольких отдельных частей. На обоих концах линейки находится стрелка, щелчок на которой вызывает перемещение текущего значения линейки прокрутки на одну единицу в направлении, указываемом стрелкой. Текущее значение линейки прокрутки по отношению к ее минимальным и максимальным значениям обозначается посредством *ползунка*

линейки прокрутки. Пользователь может перетащить ползунок в другую позицию, после чего линейка прокрутки отобразит новое значение. В фоновой части линейки по обе стороны ползунка пользователь может щелкнуть кнопкой мыши, чтобы осуществить переход в этом направлении с некоторым приращением больше 1. Обычно это действие переводится в некоторую разновидность перелистывания страниц вверх и вниз. Линейки прокрутки инкапсулируются классом `Scrollbar`.

Класс `Scrollbar` определяет следующие конструкторы:

```
Scrollbar()
Scrollbar(int style)
Scrollbar(int style, int initialValue, int thumbSize, int min, int max)
```

Первый вариант создает вертикальную линейку прокрутки. Во втором и третьем вариантах можно задавать ориентацию линейки прокрутки. Если параметр *`style`* равен `Scrollbar.VERTICAL`, создается вертикальная линейка прокрутки. Если параметр *`style`* равен `Scrollbar.HORIZONTAL`, конструируется горизонтальная линейка прокрутки. В третьем варианте конструктора исходное состояние линейки прокрутки определяется параметром *`initialValue`*. Количество единиц, представляемых высотой ползунка, задается параметром *`thumbSize`*. Минимальные и максимальные значения линейки прокрутки передаются в параметрах *`min`* и *`max`*.

Если вы создаете линейку прокрутки с помощью одного из первых двух конструкторов, то прежде чем ее можно будет использовать, вам нужно определить параметры линейки прокрутки с помощью метода `setValues()`, показанного ниже:

```
void setValues(int initialValue, int thumbSize, int min, int max)
```

Параметры имеют то же значение, что и в третьем только что описанном конструкторе.

Чтобы узнать текущее значение линейки прокрутки, вызовите метод `getValue()`. Этот метод возвращает текущую настройку. Чтобы настроить текущее значение, вызовите метод `setValue()`. Эти методы показаны ниже:

```
int getValue()
void setValue(int newValue)
```

Здесь параметр *`newValue`* определяет новое значение для линейки прокрутки. Когда вы присваиваете значение, ползунок внутри линейки прокрутки будет позиционирован так, чтобы соответствовать новому значению.

Методы `getMinimum()` и `getMaximum()`, показанные ниже, позволяют получить минимальные и максимальные значения:

```
int getMinimum()
int getMaximum()
```

Они возвращают требуемую величину.

По умолчанию прокрутка вверх или вниз на одну строку производится с приращением 1, которое добавляется или вычитается из линейки прокрутки. Изменить приращение можно с помощью метода `setUnitIncrement()`.

По умолчанию приращением для перелистывания страниц вверх и вниз является 10. Это значение можно изменить методом `setBlockIncrement()`. Последние два метода показаны далее:

```
void setUnitIncrement(int newIncr)
void setBlockIncrement(int newIncr)
```

Обработка линеек прокрутки

Для обработки событий линейки прокрутки необходимо реализовать интерфейс `AdjustmentListener`. Как только пользователь начинает работать с линейкой прокрутки, генерируется объект `AdjustmentEvent`. Его метод `getAdjustmentType()` служит для определения типа настройки. В табл. 24.1 перечислены типы событий настройки.

Таблица 24.1. События настройки

Событие	Описание
`BLOCK_DECREMENT`	Сгенерировано событие перелистывания страницы вниз.
`BLOCK_INCREMENT`	Сгенерировано событие перелистывания страницы вверх.
`TRACK`	Сгенерировано событие абсолютного перемещения.
`UNIT_DECREMENT`	Нажата кнопка перевода страницы на одну строку вниз.
`UNIT_INCREMENT`	Нажата кнопка перевода страницы на одну строку вверх.

Следующий код создает вертикальную и горизонтальную линейки прокрутки. Отображаются текущие настройки линеек прокрутки. При перемещении курсора мыши в окне координаты каждого события перемещения будут использоваться для обновления линеек прокрутки. Звездочка отображается в текущей позиции при перетаскивании.

```
// Демонстрация применения линеек 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);
    horzSB = new Scrollbar(Scrollbar.HORIZONTAL,
                           0, 1, 0, width);
    add(vertSB);
    add(horzSB);

    // регистрация на получение уведомлений о событиях
    vertSB.addAdjustmentListener(this);
    horzSB.addAdjustmentListener(this);

    addMouseMotionListener(this);
  }
```

```
  public void adjustmentValueChanged(AdjustmentEvent ae) {
    repaint();
  }

  // Обновление линеек прокрутки в ответ на перетаскивание с помощью мыши.
  public void mouseDragged(MouseEvent me) {
    int x = me.getX();
    int y = me.getY();
    vertSB.setValue(y);
    horzSB.setValue(x);
    repaint();
  }

  // Это нужно для MouseMotionListener
  public void mouseMoved(MouseEvent me) {
  }

  // Отображение текущего значения линеек прокрутки.
  public void paint(Graphics g) {
    msg = "Vertical: " + vertSB.getValue();
    // msg = "Вертикальная: " + vertSB.getValue();
    msg += ", Horizontal: " + horzSB.getValue();
    // msg += ", Горизонтальная: " + horzSB.getValue();
    g.drawString(msg, 6, 160);

    // показываем текущую позицию при перетаскивании с помощью мыши
    g.drawString("*", horzSB.getValue(),
                 vertSB.getValue());
  }
}
```

Аплет `SBDemo` во время выполнения показан на рис. 24.7.

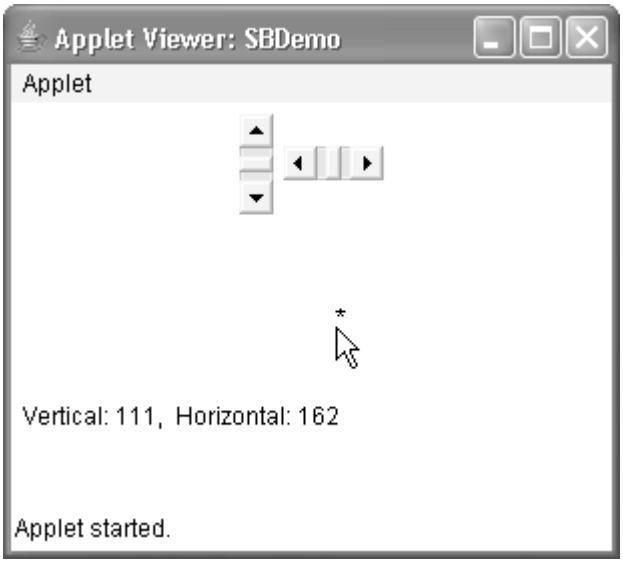

Рис. 24.7. Окно аплета `SBDemo`

Использование класса TextField

Класс TextField реализует однострочную область для ввода текста, которая называется *элементом управления для редактирования*. В текстовых полях пользователь может вводить строки и редактировать текст с помощью клавиш с изображением стрелок, клавиш вырезания и вставки, а также с помощью мыши. Класс TextField является подклассом TextComponent. Класс TextField определяет следующие конструкторы:

```
TextField() throws HeadlessException
TextField(int numChars) throws HeadlessException
TextField(String str) throws HeadlessException
TextField(String str, int numChars) throws HeadlessException
```

Первый вариант создает текстовое поле. Второй вариант создает текстовое поле, ширина которого в символах определяется параметром *numChars*. Третий вариант конструктора инициализирует текстовое поле и устанавливает его ширину.

Класс TextField (и его суперкласс TextComponent) предлагают несколько методов, с помощью которых можно работать с текстовым полем. Чтобы узнать, какая строка содержится на данный момент в текстовом поле, вызовите метод getText(). Чтобы настроить текст, вызовите метод setText(). Эти методы показаны ниже:

```
String getText()
void setText(String str)
```

Здесь параметр *str* определяет новую строку.

Пользователь может выделить часть текста в текстовом поле. Часть текста также можно выделить программно с помощью метода select(). Посредством метода getSelectedText() программа сможет узнать, какой текст выделен в данный момент. Упомянутые методы показаны ниже:

```
String getSelectedText()
void select(int startIndex, int endIndex)
```

Метод getSelectedText() возвращает выделенный текст. Метод select() выделяет символы, начиная со значения *startIndex* и заканчивая значением *endIndex*-1.

С помощью метода setEditable() можно разрешить пользователю изменять содержимое текстового поля. С помощью метода isEditable() можно позволить пользователю редактировать текст. Эти методы показаны ниже:

```
boolean isEditable()
void setEditable(boolean canEdit)
```

Метод isEditable() возвращает true, если текст можно изменять, и false, если текст изменять нельзя. В методе setEditable(), если параметр *canEdit* равен true, то текст можно изменять. Если же этот параметр равен false, текст изменять нельзя.

Иногда нужно сделать так, чтобы текст, вводимый пользователем, был невидимым (в частности, при вводе пароля). С помощью метода setEchoChar() можно отключить отображение вводимых символов. Этот метод задает одиночный символ, который TextField будет отображать при вводе символов (и, следовательно, реальные символы отображаться не будут). С помощью метода getEchoChar() можно узнать, включен ли этот режим для текстового поля. Получить символ отображения можно с помощью метода getEchoChar(). Эти методы показаны ниже:

```
void setEchoChar(char ch)
boolean echoCharIsSet()
char getEchoChar()
```

Здесь параметр *ch* определяет отображаемый символ.

Обработка TextField

Поскольку текстовые поля выполняют свои собственные функции редактирования, ваша программа вообще не будет реагировать на события отдельных клавиш, возникающие в текстовом поле. Однако вам, возможно, нужно будет сделать так, чтобы пользователь мог нажимать клавишу <Enter>. При нажатии на эту клавишу будет генерироваться событие действия.

Ниже показан пример, в котором создается классический экран для ввода имени пользователя и его пароля.

```
// Демонстрация применения 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);
    // Label namep = new Label("Имя: ", Label.RIGHT);
    Label passp = new Label("Password: ", Label.RIGHT);
    // Label passp = new Label("Пароль: ", Label.RIGHT);
    name = new TextField(12);
    pass = new TextField(8);
    pass.setEchoChar('?');

    add(namep);
    add(name);
    add(passp);
    add(pass);

    // регистрация на получение уведомлений о событиях действия
    name.addActionListener(this);
    pass.addActionListener(this);
  }

  // Пользователь нажал <Enter>.
  public void actionPerformed(ActionEvent ae) {
    repaint();
  }

  public void paint(Graphics g) {
    g.drawString("Name: " + name.getText(), 6, 60);
    g.drawString("Selected text in name: "
                + name.getSelectedText(), 6, 80);
    g.drawString("Password: " + pass.getText(), 6, 100);
```

```
    // g.drawString("Имя: " + name.getText(), 6, 60);
    // g.drawString("Выделенный текст в имени: "
    //            + name.getSelectedText(), 6, 80);
    // g.drawString("Пароль: " + pass.getText(), 6, 100);
  }
}
```

Аплет `TextFieldDemo` во время выполнения показан на рис. 24.8.

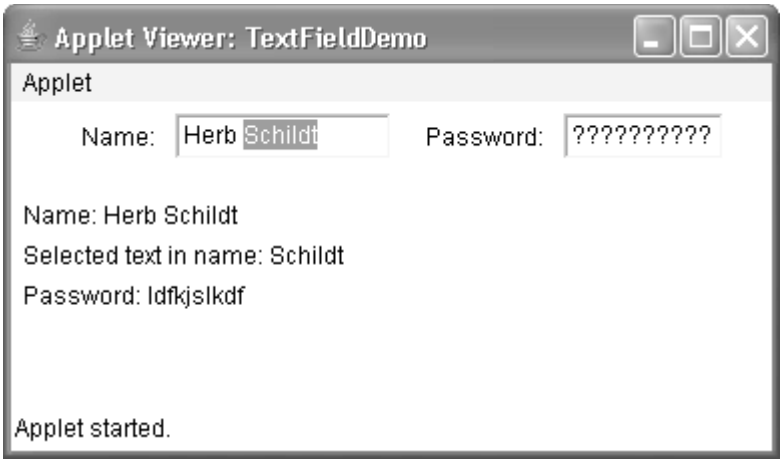

Рис. 24.8. Окно аплета `TextFieldDemo`

Использование `TextArea`

Иногда бывает так, что одной строки для ввода текста недостаточно. На этот случай AWT предлагает простой многострочный редактор `TextArea`. Ниже показаны конструкторы `TextArea`.

```
TextArea() throws HeadlessException
TextArea(int numLines, int numChars) throws HeadlessException
TextArea(String str) throws HeadlessException
TextArea(String str, int numLines, int numChars) throws HeadlessException
TextArea(String str, int numLines, int numChars, int sBars) throws HeadlessException
```

Здесь параметр *numLines* определяет высоту текстовой области, измеряемой в строках, а параметр *numChars* задает ее ширину, измеряемую в символах. Исходный текст можно определить посредством параметра *str*. В пятом варианте можно конструировать линейки прокрутки, которые могут понадобиться для работы с этой областью. Параметр *sBars* должен принимать одно из следующих значений:

- `SCROLLBARS_BOTH`
- `SCROLLBARS_NONE`
- `SCROLLBARS_HORIZONTAL_ONLY`
- `SCROLLBARS_VERTICAL_ONLY`

Класс `TextArea` является подклассом класса `TextComponent`. Следовательно, он поддерживает методы `getText()`, `setText()`, `getSelectedText()`, `select()`, `isEditable()` и `setEditable()`, описанные в предыдущем разделе.

`TextArea` включает следующие дополнительные методы:

```
void append(String str)
void insert(String str, int index)
void replaceRange(String str, int startIndex, int endIndex)
```

Метод `append()` добавляет строку, определяемую посредством параметра *`str`*, в конец текущего текста. Метод `insert()` вставляет строку, указанную в параметре *`str`*, в определенный индекс. Чтобы заменить текст, вызовите метод `replaceRange()`. Он заменяет символы, начиная с индекса, указанного в параметре *`startIndex`*, и заканчивая индексом, указанным в параметре *`endIndex`*`-1`, на текст, передаваемый в параметре *`str`*.

Текстовые области являются практически автономными элементами управления. Вашей программе не грозят издержки, связанные с управлением областями текста. Области текста генерируют только события получения и потери фокуса. Обычно программа просто получает текущий текст, если в этом есть необходимость.

В следующей программе создается элемент управления `TextArea`.

```
// Демонстрация применения TextArea.
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 SE 6 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);
  }
}
```

Аплет `TextAreaDemo` во время выполнения показан на рис. 24.9.

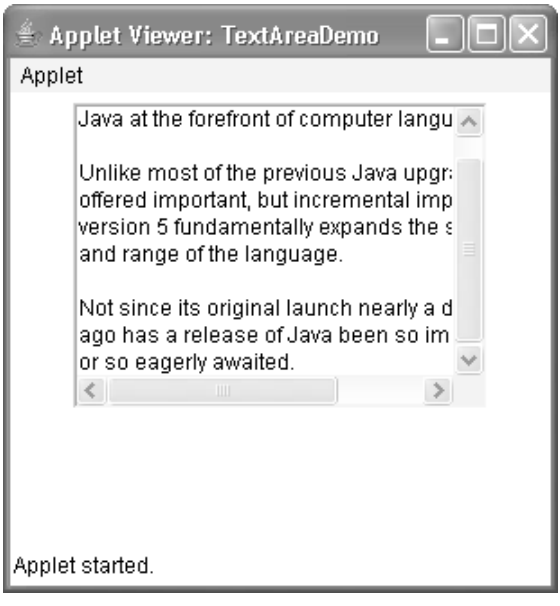

Рис. 24.9. Окно аплета `TextAreaDemo`

Диспетчеры компоновки

Каждый рассмотренный до настоящего времени компонент позиционировался диспетчером компоновки, используемым по умолчанию. Как было замечено в начале этой главы, диспетчер компоновки автоматически размещает ваши элементы управления внутри окна с применением некоторого алгоритма. Если вам приходилось писать программы для других сред с графическим пользовательским интерфейсом (например, для Windows), то вы, наверное, привыкли размещать свои средства управления вручную. Хотя элементы управления, созданные с помощью Java, тоже можно размещать вручную, вы вряд ли этим будете заниматься. На это есть две главных причины. Во-первых, размещать вручную большое количество компонентов очень утомительно. Во-вторых, в тот момент, когда вам нужно расположить какой-нибудь элемент управления, вы можете не знать о том, какую он имеет высоту и ширину, поскольку на этот момент еще не будут готовы собственные компоненты инструментария. Эта ситуация подобна загадке о происхождении курицы и яйца; трудно понять, когда можно использовать размеры данного компонента для его позиционирования относительно другого компонента.

У каждого объекта `Container` имеется свой диспетчер компоновки. Диспетчер компоновки представляет собой экземпляр любого класса, реализующего интерфейс `LayoutManager`. Диспетчер компоновки устанавливается посредством метода `setLayout()`. Если вызов метода `setLayout()` не осуществляется, используется диспетчер компоновки, принятый по умолчанию. Всякий раз при изменении размеров контейнера (или при начальном определении размеров) диспетчер компоновки используется для позиционирования каждого компонента внутри контейнера.

Метод setLayout() имеет следующую общую форму:

```
void setLayout(LayoutManager LayoutObj)
```

Здесь параметр *LayoutObj* представляет ссылку на требуемый диспетчер компоновки. Если вы хотите отменить использование диспетчера компоновки и позиционировать компоненты вручную, присвойте параметру *layoutObj* значение null. Если вы сделаете это, вам нужно будет определить форму и позицию каждого компонента вручную с помощью метода setBounds(), определенного в классе Component. Обычно вы будете работать с диспетчером компоновки.

Каждый диспетчер компоновки следит за списком компонентов, хранящихся под их именами. Диспетчер компоновки получает уведомление каждый раз, когда вы добавляете компонент в контейнер. Всякий раз, когда нужно изменить размеры контейнера, диспетчер компоновки использует для этого свои методы minimumLayoutSize() и preferredLayoutSize(). Каждый компонент, который находится под управлением диспетчера компоновки, содержит методы getPreferredSize() и getMinimumSize(). Они возвращают предпочтительный и минимальный размеры, которые необходимы для отображения каждого компонента. Диспетчер компоновки будет учитывать их, если это вообще будет возможно, и поддерживать непротиворечивую политику размещения. Можно переопределить эти методы для элементов управления, для которых вы создаете подклассы. Иначе будут применяться значения по умолчанию.

Java имеет несколько предварительно определенных классов LayoutManager, часть из которых будет описана далее. Вы можете использовать такой диспетчер компоновки, который наилучшим образом подходит для вашего приложения.

FlowLayout

Диспетчер компоновки FlowLayout используется по умолчанию. Именно этот диспетчер компоновки применялся в предыдущих примерах. Диспетчер FlowLayout реализует простой стиль компоновки, который подобен тому, как следуют друг за другом слова в текстовом редакторе. Направление размещения определяется свойством ориентации компонента контейнера, которое по умолчанию задает направление слева направо и сверху вниз. Поэтому по умолчанию компоненты размещаются построчно, начиная с левого верхнего угла. В любом случае, если строка больше не может уместить компонент, он появится в следующей строке. Между каждым компонентом ставится небольшой промежуток: сверху и снизу, а также справа и слева. Ниже показаны конструкторы FlowLayout:

```
FlowLayout()
FlowLayout(int how)
FlowLayout(int how, int horz, int vert)
```

Первый вариант размещает элементы по схеме, принятой по умолчанию: компоненты размещаются по центру, а между каждым компонентом остается промежуток, равный пяти пикселям. Второй вариант позволяет определить способ расположения каждой строки. Допустимыми значениями параметра *how* являются следующие:

```
FlowLayout.LEFT
FlowLayout.CENTER
FlowLayout.RIGHT
FlowLayout.LEADING
FlowLayout.TRAILING
```

Эти значения определяют выравнивание по левому краю, по центру, по правому краю, по переднему краю и по заднему краю, соответственно. Третий конструктор позволяет определить промежутки по горизонтали и вертикали между компонентами посредством параметров *horz* и *vert*. Ниже показан вариант уже знакомого вам аплета `CheckboxDemo`, видоизмененного таким образом, что в нем используется последовательное размещение с выравниванием влево.

```
// Компоновка с выравниванием влево.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="FlowLayoutDemo" width=250 height=200>
  </applet>
*/
public class FlowLayoutDemo extends Applet
  implements ItemListener {
  String msg = "";
  Checkbox winXP, winVista, solaris, mac;
  public void init() {
    // Задаем компоновку с выравниванием влево.
    setLayout(new FlowLayout(FlowLayout.LEFT));
    winXP = new Checkbox("Windows XP", null, true);
    winVista = new Checkbox("Windows Vista");
    solaris = new Checkbox("Solaris");
    mac = new Checkbox("Mac OS");
    add(winXP);
    add(winVista);
    add(solaris);
    add(mac);
    // Регистрирация на получение уведомлений  о событиях.
    winXP.addItemListener(this);
    winVista.addItemListener(this);
    solaris.addItemListener(this);
    mac.addItemListener(this);
  }
  // Перерисовываем в случае изменения состояния флажка.
  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }
  // Отображаем текущее состояние флажков.
  public void paint(Graphics g) {
    msg = "Current state: ";
    // msg = "Текущее состояние: ";
    g.drawString(msg, 6, 80);
    msg = " Windows XP: " + winXP.getState();
    g.drawString(msg, 6, 100);
    msg = " Windows Vista: " + winVista.getState();
    g.drawString(msg, 6, 120);
    msg = " Solaris: " + solaris.getState();
    g.drawString(msg, 6, 140);
    msg = " Mac: " + mac.getState();
    g.drawString(msg, 6, 160);
  }
}
```

Аплет `FlowLayoutDemo` во время выполнения показан на рис. 24.10. Сравните его с результатом, который был получен при выполнении аплета `CheckboxDemo` (см. рис. 24.3).

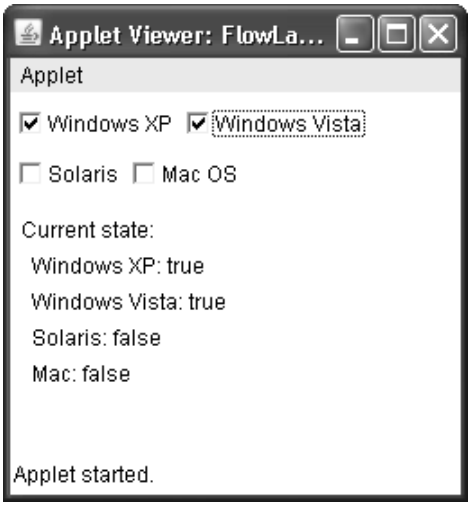

Рис. 24.10. Окно аплета `FlowLayoutDemo`

BorderLayout

Класс `BorderLayout` реализует общий стиль компоновки для окон переднего плана. Он имеет четыре узких компонента с фиксированной шириной по краям и одну большую область в центре. Четыре стороны именуют по сторонам света: север, юг, запад и восток. Область посередине именуется центром. Ниже показаны конструкторы, определяемые классом `BorderLayout`:

```
BorderLayout()
BorderLayout(int horz, int vert)
```

Первый вариант создает компоновку границы по умолчанию. Второй вариант устанавливает горизонтальный и вертикальный промежутки между компонентами посредством параметров *horz* и *vert*.

`BorderLayout` определяет следующие константы, с помощью которых указываются области:

- `BorderLayout.CENTER`
- `BorderLayout.SOUTH`
- `BorderLayout.EAST`
- `BorderLayout.WEST`
- `BorderLayout.NORTH`

При добавлении компонентов вы будете использовать эти константы в следующей форме метода `add()`, который определяется в классе `Container`:

```
void add(Component compObj, Object region)
```

Здесь параметр *compObj* представляет добавляемый компонент, а параметр *region* указывает, где будет добавлен компонент.

Ниже показан пример BorderLayout с компонентом в каждой области размещения.

```
// Демонстрация применения BorderLayout.
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("Кнопка вверху."), BorderLayout.NORTH);
    add(new Label("Сюда можно поместить сообщение нижнего колонтитула."),
        BorderLayout.SOUTH);
    add(new Button("Справа"), BorderLayout.EAST);
    add(new Button("Слева"), 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);
  }
}
```

Аплет BorderLayoutDemo во время выполнения показан на рис. 24.11.

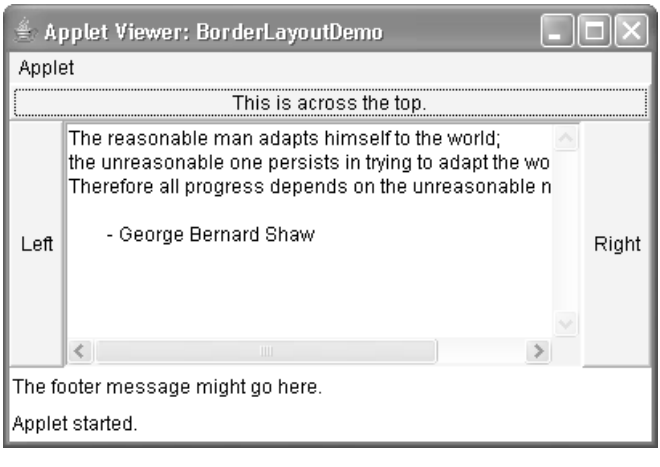

Рис. 24.11. Окно аплета BorderLayoutDemo

Использование Insets

Иногда нужно сделать так, чтобы между контейнером, в котором содержатся ваши компоненты, и окном, где он находится, оставался небольшой промежуток. Для этого необходимо переопределить метод getInsets(), определяемый классом Container. Эта

функция возвращает объект Insets, который содержит верхнюю, нижнюю, левую и правую вставки (inset), используемые при отображении объекта. Эти значения диспетчер компоновки применяет для вставки компонентов во время компоновки окна. Ниже показан конструктор Insets:

```
Insets(int top, int left, int bottom, int right)
```

Значения, передаваемые в параметрах *top*, *left*, *bottom* и *right*, определяют промежуток между контейнером и окном, в котором он заключен.

Метод getInsets() имеет следующую общую форму:

```
Insets getInsets()
```

При переопределении одного из этих методов вы должны вернуть новый объект Insets, который будет содержать требуемую промежуточную вставку.

Ниже показан измененный вариант предыдущего примера применения диспетчера BorderLayout. Здесь компоненты расставляются с отступом в десять пикселей. Для фона был выбран голубой цвет, чтобы можно было отличить вставки.

```
// Демонстрация применения BorderLayout и вставок.
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() {
    // задаем цвет фона, чтобы без труда различать вставки
    setBackground(Color.cyan);

    setLayout(new BorderLayout());

    add(new Button("This is across the top."),
        BorderLayout.NORTH);
    // add(new Button("Кнопка вверху."),
    //    BorderLayout.NORTH);
    add(new Label("The footer message might go here."),
    // add(new Label("Сюда можно поместить сообщение нижнего колонтитула."),
        BorderLayout.SOUTH);
    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);
  }
  // добавляем вставки
  public Insets getInsets() {
    return new Insets(10, 10, 10, 10);
  }
}
```

Аплет InsetsDemo во время выполнения показан на рис. 24.12.

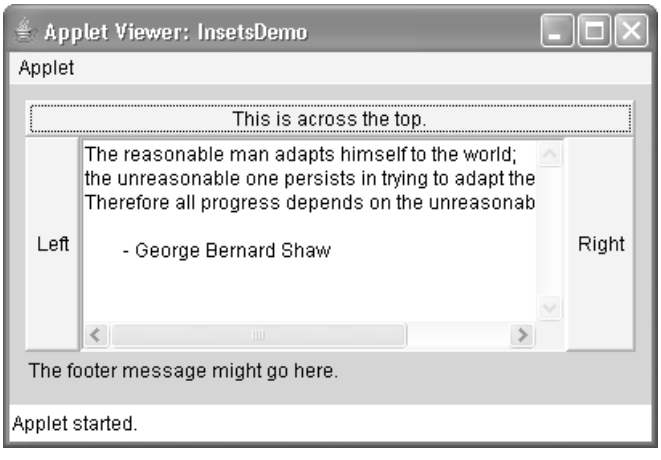

Рис. 24.12. Окно аплета InsetsDemo

GridLayout

GridLayout размещает компоненты в двухмерной сетке. Когда вы реализуете GridLayout, вы определяете количество строк и столбцов. Ниже показаны конструкторы, поддерживаемые GridLayout:

```
GridLayout()
GridLayout(int numRows, int numColumns)
GridLayout(int numRows, int numColumns, int horz, int vert)
```

Первый вариант создает сеточную компоновку с одним столбцом. Второй вариант порождает сеточную компоновку с заданным количеством строк и столбцов. Третья форма позволяет определить горизонтальный и вертикальный промежутки между компонентами посредством параметров *horz* и *vert*. Один из параметров *numRows* и *numColumns* может принимать нулевое значение. Если параметру *numRows* присвоить нулевое значение, то столбцы не будут иметь ограничения по длине. Если нулевое значение присвоить параметру *numColumns*, строки не будут иметь ограничения по длине.

Ниже показан пример программы, которая создает сетку 4×4 и заполняет ее 15 кнопками, каждая из которых обозначается своим индексом.

```
// Демонстрация применения GridLayout
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));
      }
    }
  }
}
```

Аплет `GridlayoutDemo` во время выполнения показан на рис. 24.13.

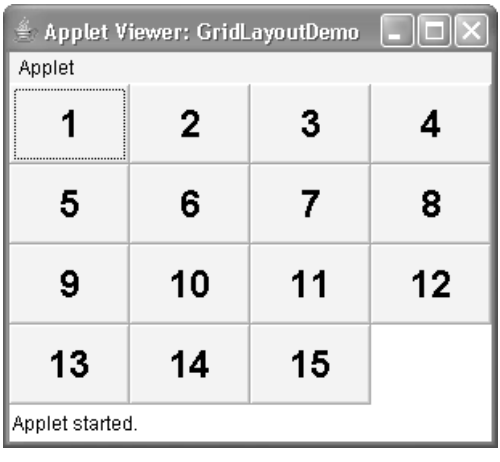

Рис. 24.13. Окно аплета `GridlayoutDemo`

Совет. *Этот пример можно взять за основу для написания программы игры в "пятнадцать".*

CardLayout

Класс `CardLayout` является уникальным среди остальных диспетчеров компоновки в том плане, что он хранит несколько различных компоновок. Каждую компоновку можно представить в виде отдельной карточки из картотеки; эти карточки можно перетасовывать как угодно, и в любой момент в начале картотеки будет находиться какая-нибудь карточка. Такой вариант может быть полезен для пользовательских интерфейсов с необязательными компонентами, которые можно динамически включать и отключать в зависимости от вводимых пользователем данных. Вы можете подготовить другие схемы компоновки и сделать их скрытыми, готовыми к активизации в случае необходимости.

`CardLayout` предлагает следующие два конструктора:

```
CardLayout()
CardLayout(int horz, int vert)
```

Первый вариант создает карточную компоновку по умолчанию. Второй вариант позволяет указать горизонтальный и вертикальный промежутки между компонентами посредством параметров *`horz`* и *`vert`*.

Использование карточной компоновки требует большего объема работы, чем другие схемы компоновки. Карточки обычно хранятся в объекте типа `Panel`. Для этой панели необходимо выбрать диспетчер компоновки `CardLayout`. Карточки, образующие картотеку, обычно тоже являются объектами типа `Panel`. Следовательно, понадобиться создать панель, которая будет содержать картотеку, и панель для каждой карточки из этой картотеки. Затем нужно добавить в соответствующую панель компоненты, формирующие каждую карточку. После этого нужно добавить эту панель на главную панель аплета. Когда это будет сделано, необходимо будет подумать о том, как пользователь сможет производить свой выбор в картотеке.

При добавлении панели карточки в панель ей обычно присваивается имя. Поэтому в большинстве случаев вы будете использовать следующую форму метода `add()` для добавления карточек на панель:

```
void add(Component panelObj, Object name)
```

Здесь параметр *`name`* представляет имя карточки, панель которой определяется посредством параметра *`panelObj`*.

После того как вы создадите картотеку, программа будет активизировать карточку с помощью одного из следующих методов, определяемых классом `CardLayout`:

```
void first(Container deck)
void last(Container deck)
void next(Container deck)
void previous(Container deck)
void show(Container deck, String cardName)
```

Здесь параметр *`deck`* представляет ссылку на контейнер (обычно панель), который хранит карточки, а параметр *`cardName`* представляет имя карточки. В результате вызова метода `first()` будет отображена первая карточка в картотеке. Чтобы показать последнюю карточку, вызовите метод `last()`. Для отображения следующей карточки вызовите метод `next()`. Чтобы показать предыдущую карточку, вызовите метод `previous()`. Методы `next()` и `previous()` автоматически циклически переходят по картотеке вверх или вниз соответственно. Метод `show()` отображает карточку, имя которой передается в параметре *`cardName`*.

В следующем коде создается двухуровневая картотека, в которой пользователь может выбрать операционную систему. Операционные системы семейства Windows отображаются на одной карточке, операционные системы Macintosh и Solaris — на другой.

```
// Демонстрация применения CardLayout.
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 winXP, winVista, 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); // присваиваем компоновке карточки компоновку панели
    winXP = new Checkbox("Windows XP", null, true);
    winVista = new Checkbox("Windows Vista");
    solaris = new Checkbox("Solaris");
    mac = new Checkbox("Mac OS");

    // добавляем на панель флажки Windows
    Panel winPan = new Panel();
    winPan.add(winXP);
    winPan.add(winVista);

    // добавляем на панель флажки других ОС
    Panel otherPan = new Panel();
    otherPan.add(solaris);
    otherPan.add(mac);

    // добавляем панели в карточную панель
    osCards.add(winPan, "Windows");
    osCards.add(otherPan, "Other");

    // добавляем карточки в главную панель аплета
    add(osCards);

    // регистрация на получение уведомлений о событиях
    Win.addActionListener(this);
    Other.addActionListener(this);

    // регистрируем события мыши
    addMouseListener(this);
  }

  // Циклический перебор панелей.
  public void mousePressed(MouseEvent me) {
    cardLO.next(osCards);
  }

  // Создаем пустые заготовки для остальных методов 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");
    }
  }
}
```

На рис. 24.14 показан аплет `CardLayoutDemo` во время выполнения. Каждая карточка активизируется после щелчка на ее кнопке. Можно циклически переходить от одной карточки к другой, щелкая кнопкой мыши.

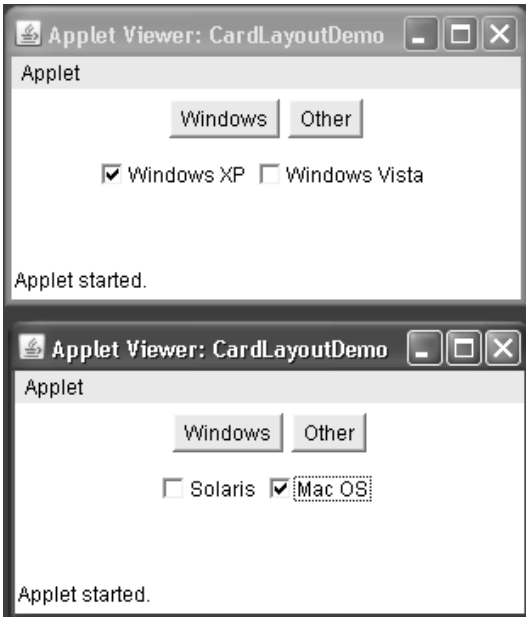

Рис. 24.14. Аплет `CardLayoutDemo` во время выполнения

GridBagLayout

Хотя предыдущие схемы компоновки отлично подходят для использования во многих аплетах, для некоторых аплетов все же необходим детальный контроль расположения компонентов в его окне. Для этого удобно применять сеточную компоновку (grid bag), которая определена в классе `GridBagLayout`. Компоновка "grid bag" обладает полезным свойством, которое позволяет задавать относительное расположение компонентов, указывая их позиции в ячейках сетки. Ключевой особенностью компоновки "grid bag" является то, что каждый компонент может иметь различные размеры, а каждая строка в сетке может иметь различное количество столбцов. Этим и объясняется название этой схемы компоновки — *grid bag* ("сетка-мешок"). Это коллекция небольших сеток, соединенных вместе.

Местонахождение и размеры каждого компонента в сеточной компоновке определяются набором ограничений, связанных с ним. Ограничения содержатся в объекте типа `GridBagConstraints`. Ограничения включают высоту и ширину ячейки, расположение компонента, его выравнивание и точку связывания внутри ячейки.

Общая процедура использования сеточной компоновки выглядит так. Сначала создается новый объект `GridBagLayout`, который определяется как текущий диспетчер компоновки. Затем выбираются ограничения для каждого компонента, добавляемого в сетку. После всего этого компоненты добавляются к диспетчеру компоновки. Хотя `GridBagLayout` является более сложным по сравнению с другими диспетчерами компоновки, использовать его будет очень легко, если вы хорошенько разберетесь с принципом его работы.

`GridBagLayout` определяет только один конструктор:

```
GridBagLayout()
```

`GridBagLayout` определяет несколько методов, многие из которых являются защищенными и не предназначены для общего использования. Однако есть один метод, который вы должны использовать: `setConstraints()`. Он показан ниже:

```
void setConstraints(Component comp, GridBagConstraints cons)
```

Здесь параметр *comp* представляет компонент, к которому применяются ограничения, определяемые с помощью параметра *cons*. Этот метод описывает ограничения, применяемые к каждому компоненту в сетке.

Залогом успешного использования `GridBagLayout` является тщательная настройка его ограничений, которые хранятся в объекте `GridBagConstraints`. `GridBagConstraints` определяет несколько полей, которые вы можете настроить для управления размерами, размещением и промежутками компонента. Эти поля перечислены в табл. 24.2. Некоторые из них будут описаны подробно немного позже.

Таблица 24.2. Ограничительные поля, определяемые в `GridBagConstraints`

Поле	Назначение
`int anchor`	Задает местоположение компонента внутри ячейки. По умолчанию — `GridBagConstraints.CENTER`.
`int fill`	Задает способ изменения размеров компонента, если размеры компонента меньше размеров его ячейки. Допустимыми являются значения `GridBagConstraints.NONE` (по умолчанию), `GridBagConstraints.HORIZONTAL`, `GridBagConstraints.VERTICAL`, `GridBagConstraints.BOTH`.
`int gridheight`	Задает высоту компонента в пересчете на ячейки. По умолчанию равно 1.
`int gridwidth`	Задает ширину компонента в пересчете на ячейки. По умолчанию равно 1.
`int gridx`	Задает координату X ячейки, в которую будет добавлен компонент. Значением по умолчанию является `GridBagConstraints.RELATIVE`.
`int gridy`	Задает координату Y ячейки, в которую будет добавлен компонент. Значением по умолчанию является `GridBagConstraints.RELATIVE`.
`Insets insets`	Задает вставки. По умолчанию все вставки являются нулевыми.
`int ipadx`	Задает дополнительный горизонтальный промежуток, окружающий компонент внутри ячейки. Значением по умолчанию является 0.
`int ipady`	Задает дополнительный вертикальный промежуток, окружающий компонент внутри ячейки. Значением по умолчанию является 0.
`double weightx`	Задает весовое значение, которое определяет горизонтальные промежутки между ячейками и краями контейнера, в котором они содержатся. Значением по умолчанию является 0,0. Чем больше вес, тем больше будет промежуток. Если все значения будут равны 0,0, то дополнительные промежутки будут распределены равномерно между краями окна.
`double weighty`	Задает весовое значение, которое определяет вертикальные промежутки между ячейками и краями контейнера, в котором они содержатся. Значением по умолчанию является 0,0. Чем больше вес, тем больше будет промежуток. Если все значения будут равны 0,0, то дополнительные промежутки будут распределены равномерно между краями окна.

GridBagConstraints определяет также несколько статических полей, которые содержат стандартные значения ограничений, такие как GridBagConstraints.CENTER и GridBagConstraints.VERTICAL.

Если размеры компонента меньше размеров его ячейки, можно воспользоваться полем anchor, чтобы определить, где внутри ячейки будет располагаться верхний левый угол компонента. Существуют три типа значений, которые можно присвоить полю anchor. Первые являются абсолютными значениями:

- GridBagConstraints.CENTER
- GridBagConstraints.SOUTH
- GridBagConstraints.EAST
- GridBagConstraints.SOUTHEAST
- GridBagConstraints.NORTH
- GridBagConstraints.SOUTHWEST
- GridBagConstraints.NORTHEAST
- GridBagConstraints.WEST
- GridBagConstraints.NORTHWEST

Как можно судить по именам этих значений, они определяют расположение компонента в определенных местах.

Второй тип значений, которые можно присвоить полю anchor, является относительным, а это означает, что значения являются относительными ориентации контейнера, которая в восточных языках может быть другой. Относительные значения перечислены ниже:

- GridBagConstraints.FIRST_LINE_END
- GridBagConstraints.LINE_END
- GridBagConstraints.FIRST_LINE_START
- GridBagConstraints.LINE_START
- GridBagConstraints.LAST_LINE_END
- GridBagConstraints.PAGE_END
- GridBagConstraints.LAST_LINE_START
- GridBagConstraints.PAGE_START

Их имена описывают расположение.

Третий тип значений, которые могут быть присвоены полю anchor, появился в Java SE 6. С их помощью можно позиционировать компоненты вертикально по отношению к базовой линии строки. Они перечислены ниже:

- GridBagConstraints.BASELINE
- GridBagConstraints.BASELINE_LEADING
- GridBagConstraints.BASELINE_TRAILING
- GridBagConstraints.ABOVE_BASELINE
- GridBagConstraints.ABOVE_BASELINE_LEADING
- GridBagConstraints.ABOVE_BASELINE_TRAILING
- GridBagConstraints.BELOW_BASELINE
- GridBagConstraints.BELOW_BASELINE_LEADING
- GridBagConstraints.BELOW_BASELINE_TRAILING

При горизонтальном положении центрирование может осуществляться или по отношению к переднему краю (LEADING), или по отношению к последнему краю (TRAILING).

Поля weightx и weighty являются достаточно важными и на первый взгляд довольно-таки запутанными. Их значения определяют, сколько дополнительного пространства будет выделено внутри контейнера для каждой строки и каждого столбца. По умолчанию оба поля имеют нулевые значения. Если все значения в столбце и строке будут нулевыми, то дополнительный промежуток будет распределен равномерно между краями окна. Увеличивая вес, вы увеличите это распределение пространства строки или столбца пропорционально остальным строкам или столбцам. Самый лучший способ понять работу этих значений — поэкспериментировать с ними на конкретном примере.

С помощью переменной gridwidth можно задать ширину ячейки в пересчете на единицы ячейки. По умолчанию эта переменная имеет значение 1.

Чтобы компонент мог использовать свободное пространство в строке, применяйте GridBagConstraints.REMAINDER. Чтобы компонент мог использовать предпоследнюю ячейку в строке, применяйте GridBagConstraints.RELATIVE. Ограничение gridheight работает точно также, но в вертикальном направлении.

Можно определить значение заполнения, которое будет использоваться для увеличения минимального размера ячейки. Для горизонтального заполнения присвойте значение переменной ipadx. Для вертикального заполнения присвойте значение переменной idpay.

Ниже показан пример, в котором GridBagLayout служит для демонстрации только что рассмотренного материала.

```
// Демонстрация применения 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 winXP, winVista, solaris, mac;

  public void init() {
    GridBagLayout gbag = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(gbag);

    // Определяем флажки.
    winXP = new Checkbox("Windows XP", null, true);
    winVista = new Checkbox("Windows Vista");
    solaris = new Checkbox("Solaris");
    mac = new Checkbox("Mac OS");

    // Определяем сетку.

    // Первой строке задаем нулевой вес, используемый по умолчанию.
    gbc.weightx = 1.0;    // используем вес столбца 1
    gbc.ipadx = 200;      // заполняем на 200 единиц
    gbc.insets = new Insets(4, 4, 0, 0); // вставка чуть в стороне
                                         // от левого верхнего угла
```

```
    gbc.anchor = GridBagConstraints.NORTHEAST;
    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(winXP, gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(winVista, gbc);
    // Задаем второй строке вес 1.
    gbc.weighty = 1.0;
    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(solaris, gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(mac, gbc);
    // Добавляем компоненты.
    add(winXP);
    add(winVista);
    add(solaris);
    add(mac);
    // Регистрация на получение уведомлений о событиях элемента.
    winXP.addItemListener(this);
    winVista.addItemListener(this);
    solaris.addItemListener(this);
    mac.addItemListener(this);
  }
  // Перерисовка в случае изменения состояния флажка.
  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }
  // Отображаем текущее состояние флажков.
  public void paint(Graphics g) {
    msg = "Current state: ";
    // msg = "Текущее состояние: ";
    g.drawString(msg, 6, 80);
    msg = " Windows XP: " + winXP.getState();
    g.drawString(msg, 6, 100);
    msg = " Windows Vista: " + winVista.getState();
    g.drawString(msg, 6, 120);
    msg = " Solaris: " + solaris.getState();
    g.drawString(msg, 6, 140);
    msg = " Mac: " + mac.getState();
    g.drawString(msg, 6, 160);
  }
}
```

Аплет `GridBagDemo` во время выполнения показан на рис. 24.15.

В этой схеме компоновки флажки операционных систем размещаются в сетке 2×2. Каждая ячейка имеет заполнение 200. Каждый компонент вставляется рядом с верхним левым углом (в 4 единицах от него). Вес столбца составляет 1, благодаря чему любой дополнительный горизонтальный промежуток распределяется равномерно между столбцами. Первая строка использует вес по умолчанию, равный 0; вторая строка имеет вес 1. Это означает, что любой дополнительный вертикальный промежуток переносится во вторую строку.

`GridBagLayout` является очень мощным диспетчером компоновки. Стоит потратить некоторое время на его изучение и эксперименты с ним. После того как вы поймете, для каких целей предназначены различные настройки, вы сможете использовать `GridBagLayout` для расположения компонентов с высокой степенью точности.

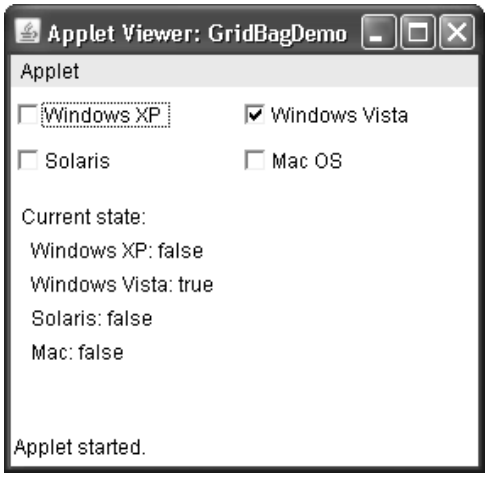

Рис. 24.15. Окно аплета `GridBagDemo`

Линейки меню и меню

Окно переднего плана может иметь связанную с ним линейку меню. Линейка меню отображает список пунктов меню верхнего уровня. Каждый пункт меню связан с выпадающим меню. Этот принцип реализован в AWT с помощью классов `MenuBar`, `Menu` и `MenuItem`. В общем случае линейка меню состоит из одного или нескольких объектов `Menu`. Каждый объект `Menu` содержит список объектов `MenuItem`. Каждый объект `MenuItem` представляет то, что может выбрать пользователь. Поскольку `Menu` является подклассом `MenuItem`, можно создать иерархию вложенных подменю. Можно также включать отмечаемые пункты меню. Они являются опциями меню типа `CheckboxMenuItem`; если пользователь щелкнет на такой опции, рядом с ней появится отметка (в виде "галочки").

Чтобы создать линейку меню, сначала создайте экземпляр `MenuBar`. Этот класс определяет только конструктор, который используется по умолчанию. Затем создайте экземпляры `Menu`, которые будут определять варианты выбора, отображаемые в строке. Ниже показаны конструкторы для `Menu`:

```
Menu() throws HeadlessException
Menu(String optionName) throws HeadlessException
Menu(String optionName, boolean removable) throws HeadlessException
```

Здесь *`optionName`* представляет имя пункта меню. Если параметр *`removable`* равен `true`, меню можно удалять и очищать. В противном случае оно останется присоединенным к линейке меню. (Удаляемые меню зависят от реализации.) Первый вариант создает пустое меню.

Отдельные пункты меню имеют тип `MenuItem`. Он определяет следующие конструкторы:

```
MenuItem() throws HeadlessException
MenuItem(String itemName) throws HeadlessException
MenuItem(String itemName, MenuShortcut keyAccel) throws HeadlessException
```

Здесь параметр *`itemName`* представляет имя, отображаемое в меню, а параметр *`keyAccel`* — клавишу быстрого доступа для данного пункта меню.

Пункт меню можно включить или отключить с помощью метода `setEnabled()`. Его форма показана далее:

```
void setEnabled(boolean enabledFlag)
```

Если параметр *`enabledFlag`* равен `true`, пункт меню будет включен, а если `false` — отключен.

Состояние пункта меню можно определить с помощью метода `isEnabled()`. Он показан ниже:

```
boolean isEnabled()
```

Метод `isEnabled()` возвращает `true`, если пункт меню, для которого он вызывается, является включенным. В противном случае он возвращает `false`.

Изменить имя пункта меню можно с помощью метода `setLabel()`. Текущее имя пункта меню можно узнать с помощью метода `getLabel()`. Эти методы показаны далее:

```
void setLabel(String newName)
String getLabel()
```

Здесь параметр *`newName`* становится новым именем вызываемого пункта меню. Метод `getLabel()` возвращает текущее имя.

Создать отмечаемый пункт меню можно с помощью подкласса `MenuItem`, называемого `CheckboxMenuItem`. Он имеет следующие конструкторы:

```
CheckboxMenuItem() throws HeadlessException
CheckboxMenuItem(String itemName) throws HeadlessException
CheckboxMenuItem(String itemName, boolean on) throws HeadlessException
```

Здесь *`itemName`* представляет имя, отображаемое в меню. Отмечаемые пункты меню работают подобно триггеру (со счетным входом). Каждый раз, когда напротив одного из пунктов ставится отметка, его состояние изменяется. В первых двух формах отмечаемое поле не имеет метки. В третьей форме, если параметр *`on`* равен `true`, отмечаемое поле имеет метку. В противном случае оно остается пустым.

Состояние отмечаемого пункта меню можно узнать с помощью метода `getState()`. Присвоить ему определенное состояние можно посредством метода `setState()`. Эти методы показаны ниже:

```
boolean getState()
void setState(boolean on)
```

Если пункт меню отмечен, метод `getState()` возвращает значение `true`. В противном случае он возвращает `false`. Чтобы отметить пункт меню, передайте значение `true` методу `setState()`. Для очистки пункта меню передайте `false`.

После того как вы создадите пункт меню, необходимо добавить его в объект `Menu` с помощью метода `add()`, который имеет следующую общую форму:

```
MenuItem add(MenuItem item)
```

Здесь *`item`* представляет добавляемый пункт меню. Добавление пунктов в меню производится в том порядке, в каком осуществлялись вызовы метода `add()`. Возвращаемым значением является *`item`*.

После того как вы добавите все пункты в объект `Menu`, его можно будет добавить в линейку меню с помощью следующей версии метода `add()`, определенного в `MenuBar`:

```
Menu add(Menu menu)
```

Здесь параметр *menu* представляет добавляемое меню. Возвращаемым значением является *menu*.

Меню генерируют события только тогда, когда производится выбор пункта типа `MenuItem` и `CheckboxMenuItem`. Они не генерируют события, когда, например, производится обращение к линейке меню с целью отображения выпадающего меню. Каждый раз при выборе пункта меню генерируется объект `ActionEvent`. По умолчанию строка команды действия представляет собой имя элемента меню. Однако вы можете определить другую строку команды действия, вызвав `setActionCommand()` в элементе меню. Каждый раз, когда отмечается или снимается отметка с пункта меню, генерируется объект `ItemEvent`. Таким образом, для обработки этих событий меню необходимо реализовать интерфейсы `ActionListener` и `ItemListener`.

Метод `getItem()` класса `ItemEvent` возвращает ссылку на пункт меню, который сгенерировал данное событие. Ниже показана общая форма этого метода:

```
Object getItem()
```

Ниже показан пример, который добавляет серию вложенных меню во всплывающее меню. В окне отображается выбранный пункт меню. Кроме этого, отображается также состояние двух пунктов меню флажков.

```
// Пример работы с меню.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="MenuDemo" width=250 height=250>
  </applet>
*/

// Создаем подкласс Frame.
class MenuFrame extends Frame {
  String msg = "";
  CheckboxMenuItem debug, test;

  MenuFrame(String title) {
    super(title);

    // Создание линейки меню и добавление ее в фрейм
    MenuBar mbar = new MenuBar();
    setMenuBar(mbar);

    // создание элеметов 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);

    // отмечаемые элементы меню
    debug = new CheckboxMenuItem("Debug");
    edit.add(debug);
    test = new CheckboxMenuItem("Testing");
    edit.add(test);

    mbar.add(edit);

    // создание объекта для обработки событий действия и элемента
    MyMenuHandler handler = new MyMenuHandler(this);
    // регистрируем объект на получение уведомлений об этих событиях
    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);

    // создание объекта для обработки событий окна
    MyWindowAdapter adapter = new MyWindowAdapter(this);
    // регистрируем объект на получение уведомлений об этих событиях
    addWindowListener(adapter);
  }

  public void paint(Graphics g) {
    g.drawString(msg, 10, 200);

   if(debug.getState())
     g.drawString("Debug is on.", 10, 220);
    else
     g.drawString("Debug is off.", 10, 220);

   if(test.getState())
     g.drawString("Testing is on.", 10, 240);
    else
     g.drawString("Testing is off.", 10, 240);
  }
}

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;
  }
  // Обработка событий действий.
  public void actionPerformed(ActionEvent ae) {
      String msg = "You selected ";
      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();
  }
  // Обработка событий элемента.
public void itemStateChanged(ItemEvent ie) {
  menuFrame.repaint();
  }
}

// Создание обрамляющего окна.
public class MenuDemo 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(new Dimension(width, height));
    f.setSize(width, height);
    f.setVisible(true);
  }
  public void start() {
    f.setVisible(true);
  }
  public void stop() {
    f.setVisible(false);
  }
}
```

Аплет `MenuDemo` во время выполнения показан на рис. 24.16.

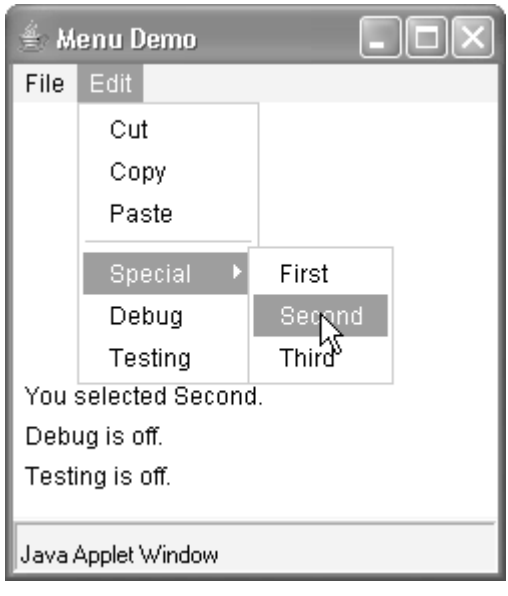

Рис. 24.16. Окно аплета `MenuDemo`

Существует еще один класс, обслуживающий меню, который вас может заинтересовать: `PopupMenu`. Он работает подобно `Menu`, но формирует меню, которое отображается в определенном месте. `PopupMenu` предлагает гибкую и полезную альтернативу некоторым типам организации меню.

Диалоговые окна

Довольно часто у вас будет возникать необходимость использовать *диалоговое окно* для хранения набора связанных между собой элементов управления. Диалоговые окна используются в первую очередь для получения данных, вводимых пользователем. Нередко они являются дочерними окнами в окне верхнего уровня. Диалоговые окна не имеют линеек меню, однако во всех других отношениях диалоговые окна функционируют по-

добно обрамляющим окнам. (Например, вы можете добавлять в них элементы управления точно так же, как вы добавляете их в обрамляющее окно.) Диалоговые окна могут быть модальными или немодальными. Когда активным является *модальное* диалоговое окно, то все вводимые данные направляются в него все время, пока оно остается открытым. Это значит, что вы не сможете обратиться к остальным частям своей программы до тех пор, пока не будет закрыто диалоговое окно. Если активным является *немодальное* диалоговое окно, то фокус ввода может быть передан другому окну вашей программы. Таким образом, остальные части вашей программы остаются активными и доступными. Диалоговые окна имеют тип `Dialog`. Ниже показаны два наиболее часто используемых конструктора:

```
Dialog(Frame parentWindow, boolean mode)
Dialog(Frame parentWindow, String title, boolean mode)
```

Здесь параметр *parentWindow* представляет "хозяина" диалогового окна. Если параметр *mode* равен `true`, диалоговое окно является модальным. В противном случае оно является немодальным. Заголовок диалогового окна можно указать в параметре *title*. В общем случае вы будете организовывать подклассы класса `Dialog`, добавляя функциональные особенности, необходимые для вашего приложения.

Ниже показан видоизмененный вариант предыдущей программы для работы с меню, в котором отображается немодальное диалоговое окно при выборе опции New (Новое). Обратите внимание, что в момент закрытия диалогового окна вызывается метод `dispose()`. Этот метод определяется классом `Window` и освобождает все системные ресурсы, задействованные диалоговым окном.

```
// Пример диалогового окна.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="DialogDemo" width=250 height=250>
  </applet>
*/

// Создание подкласса 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);
  }
}
```

```
// Создание подкласса Frame.
class MenuFrame extends Frame {
  String msg = "";
  CheckboxMenuItem debug, test;

  MenuFrame(String title) {
    super(title);

    // создание линейки меню и добавление ее во фрейм
    MenuBar mbar = new MenuBar();
    setMenuBar(mbar);

    // создание элеметов 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);

    // отмечаемые элементы меню
    debug = new CheckboxMenuItem("Debug");
    edit.add(debug);
    test = new CheckboxMenuItem("Testing");
    edit.add(test);

    mbar.add(edit);

    // создание объекта для обработки событий действия и элемента
    MyMenuHandler handler = new MyMenuHandler(this);
    // регистрация на получение уведомлений об этих событиях
    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);
```

```
  // создание объекта для обработки событий окна
  MyWindowAdapter adapter = new MyWindowAdapter(this);
   // регистрация на получение уведомлений об этих событиях
  addWindowListener(adapter);
}
public void paint(Graphics g) {
  g.drawString(msg, 10, 200);

  if(debug.getState())
    g.drawString("Debug is on.", 10, 220);
   else
    g.drawString("Debug is off.", 10, 220);

  if(test.getState())
    g.drawString("Testing is on.", 10, 240);
   else
    g.drawString("Testing is off.", 10, 240);
  }
}

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;
  }
  // Обработка событий действий.
  public void actionPerformed(ActionEvent ae) {
      String msg = "You selected ";
      String arg = ae.getActionCommand();
      // Активация диалоговго окна при выборе пункта New.
      if(arg.equals("New...")) {
         msg += "New.";
         SampleDialog d = new
           SampleDialog(menuFrame, "New Dialog Box");
         d.setVisible(true);
      }
      // Определяем остальные диалоговые окна для следующих опций.
      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();
      }
    public void itemStateChanged(ItemEvent ie) {
       menuFrame.repaint();
    }
  }
  // Создание обрамляющего окна.
  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);
    }
  }
```

Аплет `DialogDemo` во время выполнения показан на рис. 24.17.

Совет. *При желании можно попробовать определить диалоговые окна для других опций, представляемых с помощью меню.*

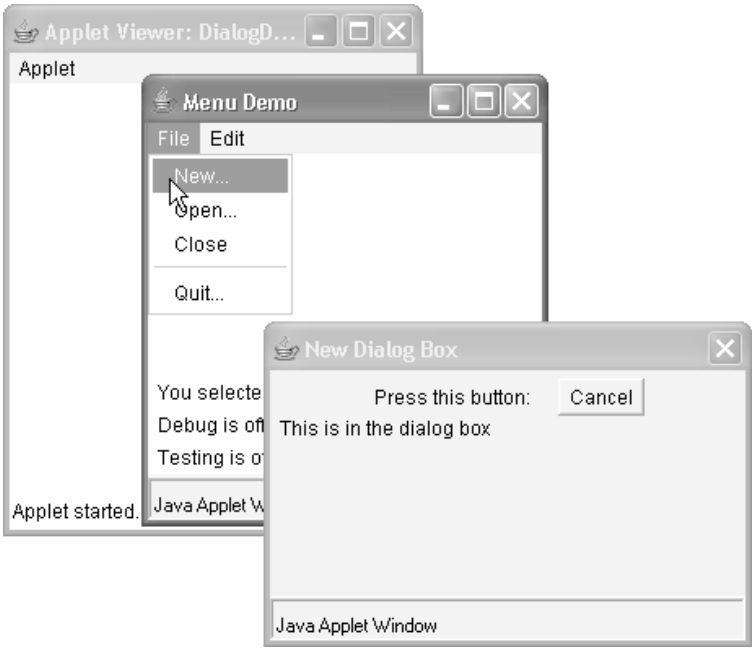

Рис. 24.17. Аплет `DialogDemo` во время выполнения

FileDialog

Java предлагает встроенное диалоговое окно, в котором пользователь может указать файл. Чтобы создать диалоговое окно выбора файла, реализуйте объект типа `FileDialog`. После этого будет отображено диалоговое окно выбора файла. Обычно оно представляет собой стандартное диалоговое окно выбора файла, используемое в операционной системе. Ниже показаны конструкторы `FileDialog`:

```
FileDialog(Frame parent, String boxName)
FileDialog(Frame parent, String boxName, int how)
FileDialog(Frame parent)
```

Здесь параметр *`parent`* представляет владельца диалогового окна, а параметр *`boxName`* задает имя, отображаемое в строке заголовка диалогового окна. Если параметр *`boxName`* опустить, заголовок диалогового окна отображаться не будет. Если параметр *`how`* будет равен `FileDialog.LOAD`, окно будет использоваться для выбора файла для чтения. Если параметр *`how`* будет равен `FileDialog.SAVE`, окно будет использоваться для выбора файла для записи. Если этот параметр опустить, окно будет использоваться для выбора файла для чтения.

`FileDialog` предлагает методы, позволяющие определять имя файла и его путь, после того как этот файл будет выбран пользователем. Ниже показано два примера этих методов:

```
String getDirectory()
String getFile()
```

Эти методы возвращают, соответственно, каталог и имя файла.

В следующей программе активизируется стандартное диалоговое окно выбора файла.

```
/* Пример диалогового окна выбора файла.
   Это приложение, а не аплет.
*/
import java.awt.*;
import java.awt.event.*;
// Создание подкласса Frame.
class SampleFrame extends Frame {
  SampleFrame(String title) {
    super(title);
    // удаление окна после его закрытия
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }
}
// Пример FileDialog.
class FileDialogDemo {
  public static void main(String args[]) {
    // создание фрейма, 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. 24.18 можно видеть результат выполнения этой программы. (Точная конфигурация диалогового окна может быть различной.)

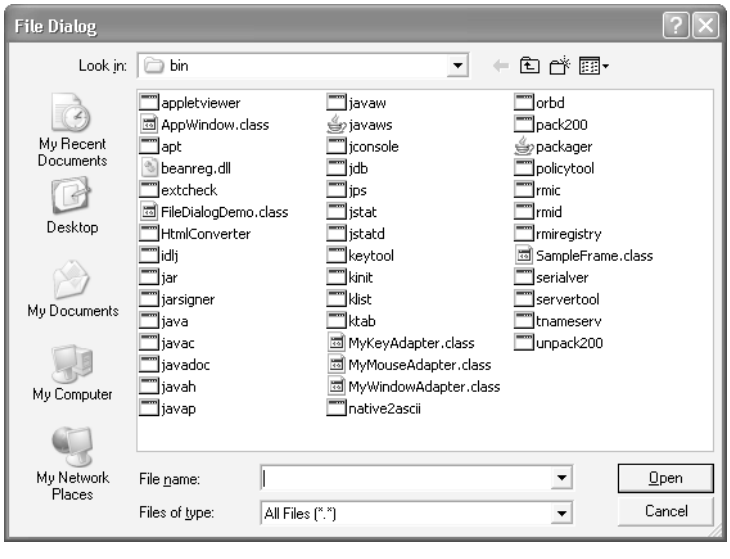

Рис. 24.18. Программа, в которой используется диалоговое окно выбора файла, во время выполнения

Обработка событий путем расширения компонентов AWT

Прежде чем завершить рассмотрение AWT, необходимо обсудить еще одну важную тему: обработку событий путем расширения компонентов AWT. Модель делегирования событий была описана в главе 22, и все предыдущие программы из этой книги использовали именно ее. Однако в Java события можно также обрабатывать за счет организации подклассов компонентов AWT. При таком подходе можно обрабатывать события почти так же, как если бы они обрабатывались под управлением исходной версии Java 1.0. Естественно, что такой подход не рекомендуется, поскольку он имеет те же недостатки, что и модель событий в Java 1.0, среди которых главным недостатком является неэффективность. Обработка событий путем расширения компонентов AWT описывается в настоящем разделе в целях полноты рассмотрения материала. Однако в остальных разделах этой книги этот подход не используется. При расширении компонента AWT необходимо вызывать метод `enableEvents()` класса `Component`. Он имеет следующую общую форму:

```
protected final void enableEvents(long eventMask)
```

Параметр *`eventMask`* представляет битовую маску, определяющую событие, которое необходимо направить этому компоненту. Для формирования маски класс `AWTEvent` определяет константы `int`. Ниже перечислены некоторые из них:

- `ACTION_EVENT_MASK`
- `KEY_EVENT_MASK`
- `ADJUSTMENT_EVENT_MASK`
- `MOUSE_EVENT_MASK`
- `COMPONENT_EVENT_MASK`
- `MOUSE_MOTION_EVENT_MASK`
- `CONTAINER_EVENT_MASK`
- `MOUSE_MOTION_EVENT_MASK`
- `FOCUS_EVENT_MASK`
- `TEXT_EVENT_MASK`
- `INPUT_METHOD_EVENT_MASK`
- `WINDOW_EVENT_MASK`
- `ITEM_EVENT_MASK`

Чтобы обработать событие, вы должны заменить соответствующий метод из одного из ваших суперклассов. В табл. 24.3 перечислены наиболее часто используемые методы и классы, которые их предлагают.

В следующих разделах будут представлены простые программы, показывающие, как осуществляется расширение компонентов AWT.

Расширение класса `Button`

В приведенной ниже программе создается аплет, который отображает кнопку с меткой “Test Button” (“Тестовая кнопка”). Если щелкнуть на этой кнопке, в строке состояния средства просмотра аплетов или браузера будет выведена строка “action event” (“событие действия”), за которой последует номер нажатой кнопки.

Таблица 24.3. Методы обработки событий

Класс	Методы обработки
Button	processActionEvent()
Checkbox	processItemEvent()
CheckboxMenuItem	processItemEvent()
Choice	processItemEvent()
Component	processComponentEvent(), processFocusEvent(), processKeyEvent(), processMouseEvent(), processMouseMotionEvent(), processMouseWheelEvent()
List	processActionEvent(), processItemEvent()
MenuItem	processActionEvent()
Scrollbar	processAdjustmentEvent()
TextComponent	processTextEvent()

Программа содержит один класс верхнего уровня ButtonDemo2, который расширяет класс Applet. В коде определяется статическая целочисленная переменная i, которой присваивается нулевое значение. Она регистрирует количество щелчков на кнопке. Метод init() реализует класс MyButton и добавляет его в аплет.

Класс MyButton является внутренним классом, расширяющим класс Button. Его конструктор использует параметр super для передачи метки кнопки конструктору суперкласса. Он вызывает метод enableEvents(), поэтому события действия могут быть получены данным объектом. Когда генерируется событие действия, вызывается метод processActionEvent(). Этот метод отображает строку в строке состояния и вызывает метод progressActionEvent() для суперкласса. Поскольку класс MyButton является внутренним, он может напрямую обращаться к методу showStatus() класса ButtonDemo2.

```
/*
* <applet code=ButtonDemo2 width=200 height=100>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class ButtonDemo2 extends Applet {
  MyButton myButton;
  static int i = 0;
  public void init() {
    myButton = new MyButton("Test Button");
    add(myButton);
  }
  class MyButton extends Button {
    public MyButton(String label) {
      super(label);
      enableEvents(AWTEvent.ACTION_EVENT_MASK);
    }
    protected void processActionEvent(ActionEvent ae) {
      showStatus("action event: " + i++);
      super.processActionEvent(ae);
    }
  }
}
```

Расширение класса Checkbox

В следующей программе создается аплет, который отображает три флажка с метками “Item 1”, “Item 2” и “Item 3”. При отметке или снятии отметки с флажка строка, содержащая имя и состояние этого флажка, будет отображаться в строке состояния средства просмотра аплетов или браузера.

Программа содержит один класс верхнего уровня `CheckboxDemo2`, который расширяет класс `Applet`. Его метод `init()` создает три экземпляра `MyCheckbox` и добавляет их в аплет. `MyCheckbox` является внутренним классом, расширяющим класс `Checkbox`. Его конструктор использует `super` для передачи метки флажка конструктору суперкласса. Он вызывает метод `enableEvents()`, так что события пунктов меню могут быть получены этим объектом. Если генерируется событие пункта, вызывается метод `processItemEvent()`. Этот метод отображает строку в строке состояния и вызывает метод `processItemEvent()` для суперкласса.

```
/*
* <applet code=CheckboxDemo2 width=300 height=100>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class CheckboxDemo2 extends Applet {
  MyCheckbox myCheckbox1, myCheckbox2, myCheckbox3;
  public void init() {
    myCheckbox1 = new MyCheckbox("Item 1");
    // myCheckbox1 = new MyCheckbox("Элемент 1");

    add(myCheckbox1);
    myCheckbox2 = new MyCheckbox("Item 2");
    // myCheckbox2 = new MyCheckbox("Элемент 2");

    add(myCheckbox2);
    myCheckbox3 = new MyCheckbox("Item 3");
    // myCheckbox3 = new MyCheckbox("Элемент 3");

    add(myCheckbox3);
  }

  class MyCheckbox extends Checkbox {
    public MyCheckbox(String label) {
      super(label);
      enableEvents(AWTEvent.ITEM_EVENT_MASK);
    }
    protected void processItemEvent(ItemEvent ie) {
      showStatus("Checkbox name/state: " + getLabel() +
                 "/" + getState());
      // showStatus("Имя/состояние флажка: " + getLabel() +
      //            "/" + getState());
      super.processItemEvent(ie);
    }
  }
}
```

Расширение группы флажков

Приведенная ниже программа является модификацией предыдущего примера с флажком. На этот раз группу флажков формируют несколько флажков. Таким образом, одновременно можно пометить только один из флажков.

```
/*
* <applet code=CheckboxGroupDemo2 width=300 height=100>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class CheckboxGroupDemo2 extends Applet {
  CheckboxGroup cbg;
  MyCheckbox myCheckbox1, myCheckbox2, myCheckbox3;
  public void init() {
    cbg = new CheckboxGroup();
    myCheckbox1 = new MyCheckbox("Item 1", cbg, true);
    add(myCheckbox1);
    myCheckbox2 = new MyCheckbox("Item 2", cbg, false);
    add(myCheckbox2);
    myCheckbox3 = new MyCheckbox("Item 3", cbg, false);
    add(myCheckbox3);
  }
  class MyCheckbox extends Checkbox {
    public MyCheckbox(String label, CheckboxGroup cbg,
                      boolean flag) {
      super(label, cbg, flag);
      enableEvents(AWTEvent.ITEM_EVENT_MASK);
    }
    protected void processItemEvent(ItemEvent ie) {
      showStatus("Checkbox name/state: " + getLabel() + "/" + getState());
      super.processItemEvent(ie);
    }
  }
}
```

Расширение класса `Choice`

В следующей программе создается аплет, который отображает список выбора с пунктами, имеющими метки “Red”, “Green” и “Blue”. При выделении пункта в строке состояния средства просмотра аплетов или браузера отображается строка, содержащая имя цвета.

Имеется один класс верхнего уровня `ChoiceDemo2`, который расширяет класс `Applet`. Его метод `init()` создает элемент выбора и добавляет его к аплету. Класс `MyChoice` является внутренним классом, который расширяет класс `Choice`. Он вызывает метод `enableEvents()`, так что этот объект может получать события пунктов. Когда генерируется событие элемента, вызывается метод `processItemEvent()`. Этот метод отображает строку в строке состояния и вызывает метод `processItemEvent()` для суперкласса.

```
/*
* <applet code=ChoiceDemo2 width=300 height=100>
* </applet>
*/
```

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class ChoiceDemo2 extends Applet {
  MyChoice choice;
  public void init() {
    choice = new MyChoice();
    choice.add("Red");
    choice.add("Green");
    choice.add("Blue");
    add(choice);
  }
  class MyChoice extends Choice {
    public MyChoice() {
      enableEvents(AWTEvent.ITEM_EVENT_MASK);
    }
    protected void processItemEvent(ItemEvent ie) {
      showStatus("Choice selection: " + getSelectedItem());
      // showStatus("Состояние выбора: " + getSelectedItem());
      super.processItemEvent(ie);
    }
  }
}
```

Расширение класса `List`

Приведенная ниже программа является модификацией предыдущего примера, но только вместо меню выбора используется список. Существует один класс верхнего уровня `ListDemo2`, который расширяет класс `Applet`. Его метод `init()` создает элемент списка и добавляет его к аплету. Класс `MyList` является внутренним классом, расширяющим класс `List`. Он вызывает метод `enableEvents()`, так что этот объект может получать события действия и элемента. Если отметить или снять отметку с элемента, будет вызван метод `processItemEvent()`. При двойном щелчке на элементе вызывается также метод `processActionEvent()`. Оба метода отображают строку, после чего передают управление суперклассу.

```
/*
* <applet code=ListDemo2 width=300 height=100>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

public class ListDemo2 extends Applet {
  MyList list;
  public void init() {
    list = new MyList();
    list.add("Red");
    list.add("Green");
    list.add("Blue");
    add(list);
  }
```

```
  class MyList extends List {
    public MyList() {
      enableEvents(AWTEvent.ITEM_EVENT_MASK | AWTEvent.ACTION_EVENT_MASK);
    }
    protected void processActionEvent(ActionEvent ae) {
      showStatus("Action event: " + ae.getActionCommand());
      // showStatus("Событие действия: " + ae.getActionCommand());
      super.processActionEvent(ae);
    }
    protected void processItemEvent(ItemEvent ie) {
      showStatus("Item event: " + getSelectedItem());
      super.processItemEvent(ie);
    }
  }
}
```

Расширение класса `Scrollbar`

В следующей программе создается аплет, который отображает линейку прокрутки. При работе с этим элементом управления в строке состояния средства просмотра аплетов или браузера отображается строка. Эта строка включает значение, представляемое линейкой прокрутки.

Имеется только один класс верхнего уровня `ScrollbarDemo2`, который расширяет класс `Applet`. Его метод `init()` создает линейку прокрутки и добавляет ее к аплету. Класс `MyScrollbar` является внутренним классом, расширяющим класс `Scrollbar`. Он вызывает метод `enableEvents()`, так что этот объект может получать события настройки. При работе с линейкой прокрутки вызывается метод `processAdjustmentEvent()`. При выборе элемента вызывается метод `processAdjustmentEvent()`. Он отображает строку, после чего передает управление суперклассу.

```
/*
* <applet code=ScrollbarDemo2 width=300 height=100>
* </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class ScrollbarDemo2 extends Applet {
  MyScrollbar myScrollbar;
  public void init() {
    myScrollbar = new MyScrollbar(Scrollbar.HORIZONTAL, 0, 1, 0, 100);
    add(myScrollbar);
  }
  class MyScrollbar extends Scrollbar {
    public MyScrollbar(int style, int initial, int thumb,
                       int min, int max) {
      super(style, initial, thumb, min, max);
      enableEvents(AWTEvent.ADJUSTMENT_EVENT_MASK);
    }
    protected void processAdjustmentEvent(AdjustmentEvent ae) {
      showStatus("Adjustment event: " + ae.getValue());
      // showStatus("Событие настройки: " + ae.getValue());
      setValue(getValue());
      super.processAdjustmentEvent(ae);
    }
  }
}
```

25 ГЛАВА

Изображения

В этой главе будет рассмотрен AWT-класс `Image` и пакет `java.awt.image`. Вместе они обеспечивают поддержку *воспроизведения изображений*, которое заключается в отображении и манипулировании графическими изображениями. *Изображением* является обычный прямоугольный графический объект. Изображения являются ключевым компонентом в Web-проектировании. Когда в 1993 году разработчики из NCSA (National Center for Supercomputer Applications — Национальный центр по применению суперкомпьютеров) решили включить дескриптор `<img>` в браузер Mosaic, это способствовало быстрому развитию системы Web. Этот дескриптор использовался для *встраивания* изображения в поток гипертекста. Java расширяет этот базовый принцип, позволяя программно управлять изображениями. Благодаря этой важной особенности, Java обеспечивает всестороннюю поддержку воспроизведения изображений.

Изображения представляют собой объекты класса `Image`, который является частью пакета `java.awt`. Манипулирование изображениями осуществляется с помощью классов пакета `java.awt.image`. Пакет `java.awt.image` содержит большое количество классов воспроизведения изображений и интерфейсов. Не имея возможности рассмотреть каждый класс и интерфейс, мы сосредоточимся на тех из них, которые участвуют в основном процессе воспроизведения изображений. Ниже перечислены классы пакета `java.awt.image`, которые будут рассмотрены в этой главе:

- `CropImageFilter`
- `MemoryImageSource`
- `FilteredImageSource`
- `PixelGrabber`
- `ImageFilter`
- `RGBImageFilter`

Мы будем применять следующие интерфейсы:

- `ImageConsumer`
- `ImageObserver`
- `ImageProducer`

Также в главе рассматривается класс `MediaTracker`, который является частью пакета `java.awt`.

Форматы файлов

Первоначально Web-изображения могли быть представлены только в формате GIF. Формат изображений GIF был разработан специалистами из CompuServe в 1987 году для того, чтобы сделать возможным просмотр изображений в оперативном режиме, поэтому его удобно было использовать и для Internet. Каждое GIF-изображение может содержать не более 256 цветов. В связи с этим ограничением в 1995 году ведущие разработчики браузеров включили поддержку JPEG-изображений. Формат JPEG был разработан группой специалистов-фотографов для хранения псевдополутоновых изображений с полным спектром цветов. Если правильно сформировать подобное изображение, то оно будет передавать более высокую степень точности, и его можно будет сжимать более компактно, чем аналогичное изображение в формате GIF. Другим форматом файла является PNG. Это тоже разновидность GIF. Как правило, в своих программах вам не придется решать, какой формат нужно использовать. Классы изображений Java абстрагируют все различия между форматами благодаря безупречному интерфейсу.

Основы работы с изображениями: создание, загрузка и отображение

В процессе работы с изображениями вы будете выполнять в основном три операции: создание изображения, его загрузка и отображение. Чтобы обратиться к изображениям, которые хранятся в памяти, а также к изображениям, которые необходимо загрузить из внешних источников данных, в Java используется класс `Image`. Таким образом, Java обеспечивает способы создания нового объекта изображения и способы его загрузки. Помимо этого, Java предлагает функциональные средства, позволяющие отображать изображения. Все это мы рассмотрим прямо сейчас.

Создание объекта `Image`

Могло показаться, что для создания изображения в памяти требуется примерно такая строка:

```
Image test = new Image(200, 100); // Ошибка — не будет работать
```

Однако это не так. Любое изображение предназначено для того, чтобы его можно было отобразить на экране монитора, а класс `Image` не располагает достаточным количеством информации об условиях, необходимых для создания подходящего формата данных для экрана. Поэтому класс `Component` пакета `java.awt` включает метод `createImage()`, используемый для создания объектов `Image`. (Имейте в виду, что все компоненты AWT являются подклассами класса `Component`, поэтому каждый из них поддерживает данный метод.)

Метод `createImage()` имеет следующие две формы:

```
Image createImage(ImageProducer imgProd)
Image createImage(int width, int height)
```

В первом случае возвращается изображение, созданное с помощью производителя (параметр *imgProd*), который представляет собой объект класса, реализующий интерфейс ImageProducer. (О производителях изображений мы поговорим чуть позже.) Во втором случае возвращается пустое изображение, имеющее определенную ширину и высоту. Ниже показан его пример:

```
Canvas c = new Canvas();
Image test = c.createImage(200, 100);
```

Здесь для формирования объекта Image будет создан экземпляр Canvas и вызван метод createImage(). Пока что изображение остается пустым. Позже вы увидите, как можно записать в него данные.

Загрузка изображения

Чтобы получить определенное изображение, его можно также загрузить. Для этого служит метод getImage(), определенный в классе Applet. Он имеет следующие формы:

```
Image getImage(URL url)
Image getImage(URL url, String imageName)
```

В первом случае будет возвращен объект Image, который инкапсулирует изображение, хранящееся по адресу, определяемому в параметре *url*. Во втором случае будет возвращен объект Image, инкапсулирующий изображение, адрес и имя которого определяются параметрами *url* и *imageName* соответственно.

Отображение изображения

После того как вы получите изображение, его можно будет отобразить с помощью метода drawImage(), который является членом класса Graphics. Этот метод имеет несколько форм. Ниже представлена форма, которая используется в настоящей книге:

```
boolean drawImage(Image imgObj, int left, int top, ImageObserver imgOb)
```

В этом случае будет отображено изображение, передаваемое посредством параметра *imgObj*; верхний левый угол изображения определяют параметры *left* и *top*. Параметр *imgOb* представляет ссылку на класс, который реализует интерфейс ImageObserver. Этот интерфейс реализуют все компоненты AWT. *Наблюдатель изображения* представляет собой объект, который может вести наблюдение за изображением во время его загрузки. ImageObserver мы рассмотрим в следующем разделе.

Загрузить и отобразить изображение с помощью методов getImage() и drawInage() несложно. Ниже показан пример аплета, который загружает и отображает отдельное изображение. В этом аплете загружается файл seattle.jpg, однако для данного примера вы можете взять любой другой файл формата GIF, JPG или PNG (при условии, что он будет находиться в том же каталоге, что и HTML-файл, содержащий аплет).

```
/*
* <applet code="SimpleImageLoad" width=248 height=146>
* <param name="img" value="seattle.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, возвращенное методом `getImage()`. Метод `getImage()` использует строку, возвращенную методом `getParameter("img")`, в качестве имени файла с изображением. Это изображение загружается из `URL`, который связан с результатом выполнения метода `getDocumentBase()`, то есть с `URL` HTML-страницы, в которой находился этот дескриптор аплета. Имя файла, возвращенное `getParameter("img")`, получено из дескриптора аплета `<param name="img" value="seattle.jpg">`. Такая схема эквивалентна применению HTML-дескриптора `<img scr="seattle.jpg" width=248 height=146>`, хотя она работает немного медленнее. На рис. 25.1 можно видеть результат выполнения этой программы.

Рис. 25.1. Результат выполнения программы `SimpleImageLoad`

Выполнение этого аплета начинается с загрузки изображения `img` в методе `init()`. На экране монитора вы будете видеть изображение в процессе его загрузки из сети, поскольку при реализации интерфейса `ImageObserver` классом `Applet` метод `paint()` вызывается каждый раз при поступлении новых данных, связанных с изображением.

Конечно, неплохо иметь возможность наблюдать за ходом загрузки изображения, однако будет лучше, если время, отводимое на загрузку изображения, вы потратите на параллельное выполнение других задач. Это позволит отобразить полностью сформированное изображение на экране монитора мгновенно сразу же после его загрузки. Интерфейс `ImageObserver`, о котором пойдет речь далее, можно применять для наблюдения за ходом загрузки изображения в момент вывода на экран монитора какой-то другой информации.

Интерфейс `ImageObserver`

Интерфейс `ImageObserver` используется для получения уведомления о формировании изображения; он определяет только один метод: `imageUpdate()`. Используя наблюдатель изображений, вы сможете выполнять ряд других действий — отображать ин-

дикатор процесса выполнения или, к примеру, переключиться на экран при получении уведомления о ходе процесса загрузки. Этот тип уведомлений полезен при загрузке изображения по сети, если разработчик содержимого не считается с теми пользователями, которые пытаются загружать аплеты с помощью низкоскоростных модемов.

Метод `imageUpdate()` имеет следующую общую форму:

```
boolean imageUpdate(Image imgObj, int flags, int left,
                    int top, int width, int height)
```

Здесь параметр *imgObj* представляет загружаемое изображение, а параметр *flags* представляет целое число, которое показывает состояние обновляемого отчета. Четыре целочисленных параметра, *left*, *top*, *width* и *height*, представляют прямоугольник, значения которого зависят от значений, передаваемых в параметре *flags*. Метод `imageUpdate()` возвращает `false`, если процесс загрузки был завершен, и `true`, если необходимо обработать еще одно изображение.

Параметр *flags* содержит один или несколько битовых флагов, определяемых как статические переменные в рамках интерфейса `ImageObserver`. Эти флаги, а также передаваемая с их помощью информация, представлены в табл. 25.1.

Класс `Applet` имеет реализацию метода `imageUpdate()` для интерфейса `ImageObserver`, который используется для перерисовки изображений во время их загрузки. Вы можете изменить это поведение, если переопределите данный метод в своем классе.

Таблица 25.1. Битовые флаги параметра *flags* метода `imageUpdate()`

Флаг	Назначение
`WIDTH`	Параметр *width* является действительным и содержит значение ширины изображения.
`HEIGHT`	Параметр *height* является действительным и содержит значение высоты изображения.
`PROPERTIES`	Свойства, связанные с изображением, можно получить с помощью метода `imgObj.getProperty()`.
`SOMEBITS`	Были получены дополнительные пиксели, необходимые для рисования изображения. Параметры *left*, *top*, *width* и *height* определяют прямоугольник, содержащий новые пиксели.
`FRAMEBITS`	Был получен весь кадр, являющийся частью ранее нарисованного многокадрового изображения. Этот кадр можно отобразить. Параметры *left*, *top*, *width* и *height* не используются.
`ALLBITS`	Изображение готово. Параметры *left*, *top*, *width* и *height* не используются.
`ERROR`	Обнаружена ошибка в изображении, за которым велось асинхронное наблюдение. Изображение не готово и не может быть отображено. Не получено никакой дополнительной информации об изображении. Будет установлен также флаг `ABORT`, чтобы показать, что процесс формирования изображения был прерван.
`ABORT`	Формирование изображения, за которым велось асинхронное наблюдение, было прервано до того, как оно было полностью готово. Однако если не было ошибки, то при попытке обращения к какой-либо части данных изображения начнется повторное формирование изображения.

Ниже показан 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.");
    // System.out.println("Изображение все еще обрабатывается.");
    return true;
  } else {
    System.out.println("Done processing the image.");
    // System.out.println("Обработка изображения завершена.");
    return false;
  }
}
```

Двойная буферизация

Изображения удобно использовать не только для хранения картинок и рисунков, как было показано только что, но и в качестве внеэкранных поверхностей рисования. С их помощью можно визуализировать любое изображение, включая текст и графику во внеэкранном буфере, содержимое которого можно будет отобразить через некоторый промежуток времени. Преимущество такого подхода заключается в том, что изображение можно будет увидеть лишь после того, как оно будет полностью готово. Для рисования сложной поверхности может потребоваться несколько миллисекунд или более, причем для пользователя этот процесс может выглядеть как серия вспышек или мерцаний.

Подобные эффекты отвлекают внимание и приводят к тому, что пользователь воспринимает визуализируемое изображение гораздо 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, а внеэкранное изображение является вторым буфером, в котором можно подготавливать пиксели для визуализации.

Вы уже видели в этой главе, как создается пустой объект `Image`. Сейчас будет показано, как осуществляется рисование изображения на самом экране. Если вы помните из предыдущих глав, для этой цели нужен объект `Graphics`, благодаря которому можно будет использовать любые методы визуализации, доступные в Java. Он удобен тем, что доступ к объекту `Graphics`, который можно применять для рисования изображения `Image`, осуществляется с помощью метода `getGraphics()`. Ниже показан фрагмент кода, в котором создается новое изображение, принимается графическое содержимое и все изображение заполняется красными пикселями.

```
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 вы создадите и заполните внеэкранное изображение, его видно не будет. Чтобы отобразить искомое изображение, вызовите метод `drawImage()`. Ниже показан пример, в котором для рисования изображения требуется большое количество времени. Вы сможете сравнить, как двойная буферизация влияет на восприятие времени рисования.

```
/*
 * <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);
    addMouseMotionListener(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);
    // g.drawString("Щелкните для включения двойной буферизации", 10, h/2);
    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 аплет имеет сложный метод `paint()`. Он окрашивает фон в синий цвет, а затем рисует поверх него красный муар. Вверху он выводит текст черного цвета и вычерчивает желтую окружность, центр которой определяется по координатам `mx`,`my`. Методы `mouseMoved()` и `mouseDragged()` заменяются для отслеживания положения курсора мыши. Эти методы идентичны друг другу, за исключением булевской переменной `flicker`. Метод `mouseMoved()` присваивает переменной `flicker` значение `true`, а метод `mouseDragger()` — значение `false`. Это равнозначно эффекту вызова метода `repaint()`, когда переменная `flicker` имеет значение `true` при перемещении курсора мыши (без щелчка ее кнопкой), и значение `false` при перемещении курсора мыши с удерживанием кнопки в нажатом состоянии.

Если метод `paint()` вызывается тогда, когда переменная `flicker` имеет значение `true`, мы будем видеть на экране монитора выполнение каждой операции рисования. Если был произведен щелчок кнопкой мыши, и метод `paint()` вызывается при том условии, что переменная `flicker` имеет значение `false`, мы будем наблюдать несколько иную картину. Метод `paint()` заменяет ссылку `g` на `Graphics` графическим содержимым внеэкранного холста, `buffer`, который мы создали с помощью метода `init()`. Затем все операции рисования становятся невидимыми. В завершающей части работы метода `paint()` мы просто вызываем метод `drawImage()` для одновременного отображения результатов всех методов рисования.

Обратите внимание, что теперь можно передавать методу `drawImage()` значение `null` в четвертом параметре. Этот параметр служит для передачи объекта `ImageObserver`, который получает уведомление о событиях изображения. Поскольку это изображение не формируется из сетевого потока, уведомления не нужны. Левый снимок экрана на рис. 25.2 показывает, как будет выглядеть аплет, если не будет произведен щелчок кнопкой мыши. Как можно видеть, снимок был получен как раз в тот момент, когда изображение было наполовину перерисованным. Правый снимок показывает, что при нажатой кнопке мыши изображение оказывается полностью сформированным благодаря использованию двойной буферизации.

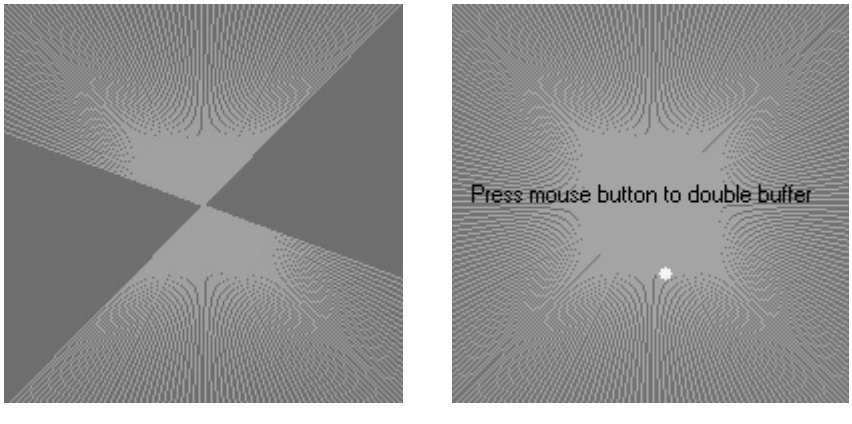

Рис. 25.2. Результат выполнения аплета `DoubleBuffer` без использования двойной буферизации (слева) и с использованием двойной буферизации (справа)

MediaTracker

Многие первые Java-разработчики считали, что интерфейс `ImageObserver` является слишком сложным, чтобы его можно было понять и работать с ним, когда стоял вопрос о загрузке нескольких изображений. Разработчикам нужно было найти более простое решение, которое позволило бы программистам загружать все свои изображения синхронно, не прибегая к методу `imageUpdate()`. В ответ на эти требования Sun Microsystems в следующем выпуске JDK добавила в пакет `java.awt` класс `MediaTracker`. `MediaTracker` представляет собой объект, который параллельно проверяет состояние произвольного количества изображений.

Чтобы использовать `MediaTracker`, нужно создать новый экземпляр и использовать его метод `addImage()` для наблюдения за состоянием загрузки изображения. Метод `addImage()` имеет следующие общие формы:

```
void addImage(Image imgObj, int imgID)
void addImage(Image imgObj, int imgID, int width, int height)
```

Здесь параметр *imgObj* представляет отслеживаемое изображение. Его идентификатор передается в параметре *imgID*. Идентификаторы не обязательно должны быть уникальными. Один и тот же идентификатор можно использовать в нескольких изображениях, обозначая их как часть группы. Во второй форме параметры *width* и *height* определяют размеры объекта при его отображении.

После того как изображение будет зарегистрировано, можно проверить, загружено ли оно, или подождать пока оно полностью не загрузится. Чтобы проверить состояние изображения, вызовите метод `checkID()`. В этой главе используется следующий вариант этого метода:

```
boolean checkID(int imgID)
```

Здесь параметр *imgID* определяет идентификационный номер изображения, которое вы хотите проверить. Метод возвращает `true`, если были загружены все изображения, имеющие заданный идентификатор (или если процесс загрузки был остановлен вследствие ошибки или был прерван пользователем). В противном случае он возвращает `false`. Вы можете использовать метод `checkAll()` для просмотра, все ли наблюдаемые изображения были загружены.

`MediaTracker` следует использовать при загрузке группы изображений. Если все интересующие вас изображения еще не загружены, можете отобразить что-нибудь, чтобы отвлечь пользователя, пока не будут полностью загружены все изображения.

Внимание! *Если `MediaTracker` использовать после вызова метода `addImage()`, то ссылка на `MediaTracker` предотвратит процесс сборки мусора в системе. Если вы хотите, чтобы система могла запускать сборщик мусора относительно отслеживаемых изображений, убедитесь в том, что он будет выполняться и в отношении экземпляра `MediaTracker`.*

Ниже показан пример, в котором загружается семь изображений и отображается привлекательная диаграмма хода выполнения 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];
  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 этом примере создается новый экземпляр `MediaTracker` в методе `init()`, после чего с помощью метода `addImage()` каждое из указанных изображений добавляется как отслеженное изображение. В методе `paint()` вызывается метод `checkID()` для каждого изображения, за которым велось наблюдение. После загрузки всех изображений они будут выведены на экран. В противном случае отображается простая диаграмма, информирующая о количестве загруженных изображений, а под ней выводятся имена полностью загруженных изображений. На рис. 25.3 показаны две сцены из этого выполняющегося аплета. На одной из них видна диаграмма, информирующая о загрузке трех изображений. Другая сцена — это автопортрет Ван Гога во время демонстрации слайда.

Интерфейс `ImageProducer`

`ImageProducer` является интерфейсом для объектов, которые должны подготовить данные для изображений. Объект, реализующий интерфейс `ImageProducer`, задает целочисленный или байтовый массив, представляющий данные изображений, и формирует объекты `Image`. Как вы могли видеть ранее, одна из форм метода `createImage()` принимает объект `ImageProducer` в качестве своего параметра. Пакет `java.awt.image` содержит два производителя изображений: `MemoryImageSource` и `FilteredImageSource`. Здесь мы рассмотрим `MemoryImageSource` и создадим новый объект `Image` на основе данных, сгенерированных в аплете.

`MemoryImageSource`

`MemoryImageSource` является классом, формирующим новое изображение `Image` на основе массива данных. Он определяет несколько конструкторов. Ниже показан один из конструкторов, который мы будем использовать:

```
MemoryImageSource(int width, int height, int pixel[], int offset,
                  int scanLineWidth)
```

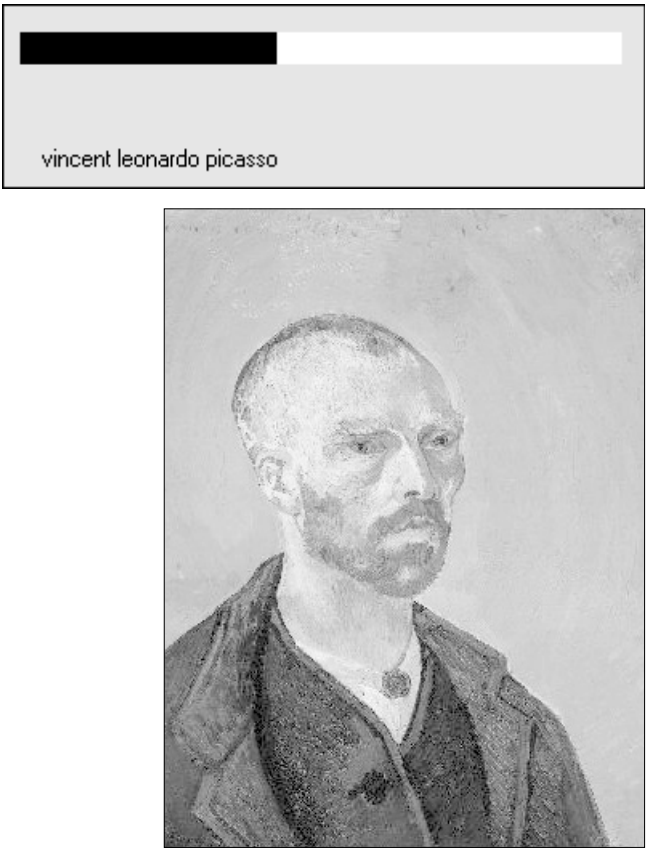

Рис. 25.3. Аплет `TrackedImageLoad` во время выполнения

Объект `MemoryImageSource` формируется на основе массива целых чисел *`pixel`* в цветовой модели RGB, которая используется по умолчанию для подготовки данных для объекта `Image`. В цветовой модели, используемой по умолчанию, пиксель представляет собой целое число со значениями альфа, красной, зеленой и синей составляющих (0xAARRGGBB). Значение альфа представляет степень прозрачности пикселя. Полностью прозрачному пикселю соответствует нулевое значение, а полностью непрозрачному — значение 255. Значения ширины и высоты готового изображения передаются в параметрах *`width`* и *`height`*. Начальная точка в массиве пикселей, с которой начнется чтение данных, определяется параметром *`offset`*. Ширина строки развертки (которая часто определяет то же, что и ширина изображения) указывается в параметре *`scanLineWidth`*.

В следующем коротком примере генерируется объект `MemoryImageSource` с помощью разновидности простого алгоритма (побитовое исключающее "ИЛИ" координат x и y каждого пикселя), взятого из книги *Beyond Photography, The Digital Darkroom* Джерарда Дж. Хольцманна (Gerard J. Holzmann, 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 `MemoryImageSource` создаются в методе `init()`. Массив целых чисел предназначен для хранения значений пикселей; данные генерируются во вложенных циклах `for`, в которых значения `r`, `g` и `b` получаются смещенными на пиксель в массиве `pixels`. В конце вызывается метод `createImage()` с новым экземпляром `MemoryImageSource`, созданным из raw-данных о пикселях в качестве его параметра. На рис. 25.4 показано изображение в момент запуска аплета. (В цвете оно выглядит гораздо привлекательнее.)

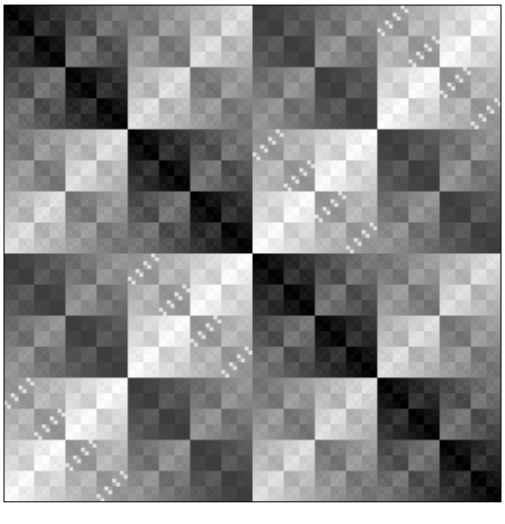

Рис. 25.4. Результат выполнения аплета `MemoryImageGenerator`

Интерфейс ImageConsumer

Интерфейс ImageConsumer является абстрактным интерфейсом для объектов, которые будут принимать данные о пикселях из изображений и задавать их в качестве другой разновидности данных. Таким образом, этот интерфейс является прямой противоположностью описанному ранее интерфейсу ImageProducer. Объект, реализующий интерфейс ImageConsumer, будет создавать массивы int или byte, представляющие пиксели из объекта Image. Мы рассмотрим класс PixelGrabber, который является простой реализацией интерфейса ImageConsumer.

Класс PixelGrabber

Класс PixelGrabber определен в пакете java.lang.image. Он является прямой противоположностью классу MemoryImageSource. Вместо того чтобы формировать изображение на основе массива значений пикселей, он принимает существующее изображение и *захватывает* в нем массив пикселей. Чтобы использовать класс PixelGrabber, вы сначала создаете массив целых чисел int, размер которого позволяет хранить данные о пикселях, после чего создаете экземпляр PixelGrabber, передавая прямоугольную область, которую необходимо захватить. Затем для этого экземпляра нужно вызывать метод grabPixels().

Ниже показан конструктор PixelGrabber, используемый в этой главе:

```
PixelGrabber(Image imgObj, int left, int top, int width, int height,
             int pixel[ ], int offset, int scanLineWidth)
```

Здесь параметр *imgObj* представляет объект, пиксели которого будут захвачены. Значения параметров *left* и *top* указывают верхний левый угол прямоугольной области, а значения параметров *width* и *height* задают размеры прямоугольной области, из которой будут получены пиксели. Пиксели будут храниться в массиве *pixel*, начиная со смещения, определенного в параметре *offset*. Ширина строки развертки (которая часто равнозначна ширине изображения) передается посредством соответствующего параметра (*scanLineWidth*).

Метод grabPixels() определяется следующим образом:

```
boolean grabPixels() throws InterruptedException
boolean grabPixels(long milliseconds) throws InterruptedException
```

Оба метода возвращают true при успешном завершении выполнения и false — в противном случае. Во второй форме метода параметр *milliseconds* определяет время, в течение которого метод будет ожидать получение пикселей. Оба метода генерируют исключение InterruptedException, если выполнение прерывается другим потоком.

Ниже показан пример, в котором производится захват пикселей в изображении с последующим построением гистограммы яркости пикселей. *Гистограмма* представляет собой простой подсчет пикселей, имеющих некоторую яркость для всех настроек яркости между 0 и 255. После того как аплет нарисует изображение, вверху выводится гистограмма.

```
/*
 * <applet code=HistoGrab.class width=341 height=400>
 * <param name=img value=vermeer.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("Interrupted");
      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;
    }
  }
}
```

На рис. 25.5 показано изображение и гистограмма знаменитой картины Вермеера (Vermeer).

Рис. 25.5. Результат выполнения аплета `HistoGrab`

Класс `ImageFilter`

При наличии пары интерфейсов `ImageProducer` и `ImageConsumer`, а также их классов `MemoryImageSource` и `PixelGrabber`, можно создать произвольный набор фильтров преобразования, которые будут принимать источники пикселей, модифицировать их и передавать некоторому потребителю. Этот механизм аналогичен механизму создания определенных классов на основе абстрактных классов ввода-вывода `InputStream`, `OutputStream`, `Reader` и `Writer` (они рассматривались в главе 19). Данная потоковая модель для изображений завершается за счет введения класса `ImageFilter`.

Пакет `java.awt.image` содержит подклассы класса `ImageFilter`, среди которых имеются `AreaAveragingScaleFilter`, `CropImageFilter`, `ReplicateScaleFilter` и `RGBImageFilter`. Существует также реализация интерфейса `ImageProducer`, `FilteredImageSource`, который принимает произвольный класс `ImageFilter` и "вкладывает в него" `ImageProducer` для фильтрации формируемых ним пикселей. Экземпляр `FilteredImageSource` можно использовать в качестве `ImageProducer` при вызовах метода `createImage()` почти так же, как и `BufferedInputStream` можно применять в качестве `InputStream`.

В этой главе мы рассмотрим два фильтра: `CropImageFilter` и `RGBImageFilter`.

Фильтр CropImageFilter

Фильтр CropImageFilter осуществляет фильтрацию источника изображения для извлечения прямоугольной области. Этот фильтр будет полезен, например, при работе с несколькими небольшими изображениями, созданными из одного крупного исходного изображения. Для загрузки двадцати изображений размером 2 Кбайт потребуется больше времени, чем для загрузки одного изображения размером 40 Кбайт, имеющего множество внедренных анимационных кадров. Если каждое составное изображение имеет один и тот же размер, то вы без особого труда сможете извлечь их с помощью фильтра CropImageFilter, расчленяя весь блок в самом начале работы аплета. Ниже показан пример формирования 16 изображений на основе одного большого изображения. Мозаичные элементы затем перетасовываются 32 раза путем замены случайной пары, взятой из 16 изображений.

```
/*
 * <applet code=TileImage.class width=288 height=399>
 * <param name=img value=picasso.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);
      }
    }
  }
}
```

На рис. 25.6 показана известная картина Пикассо, элементы которой перетасованы с помощью аплета `TileImage`.

Рис. 25.6. Результат выполнения аплета `TileImage`

Фильтр RGBImageFilter

Фильтр RGBImageFilter используется для преобразования одного изображения в другое, пиксель за пикселем, при этом осуществляя преобразование цветов. Этот фильтр можно использовать для того, чтобы осветлить изображение, увеличить его контрастность или даже для преобразования его в полутоновое изображение.

Чтобы продемонстрировать работу фильтра RGBImageFilter, мы подготовили более сложный пример, который показывает применение динамической стратегии встраивания для фильтров обработки изображений. Мы создали интерфейс для обобщенной фильтрации изображения, чтобы аплет мог просто загружать эти фильтры на основании дескрипторов <param>, не имея при этом предварительной информации о каждом фильтре ImageFilter. Пример включает главный класс аплета ImageFilterDemo, интерфейс PlugInFilter и служебный класс LoadedImage, который инкапсулирует некоторые методы MediaTracker, используемые в этой главе. Кроме того, используются три фильтра, Grayscale, Invert и Contrast, которые просто манипулируют пространством цветов исходного изображения с помощью фильтров RGBImageFilter, и два дополнительных класса, Blur и Sharpen, позволяющие реализовать более сложные фильтры свертки, изменяющие данные о пикселях на основе пикселей, окружающих их в исходных данных. Blur и Sharpen являются подклассами абстрактного вспомогательного класса Convolver. Давайте рассмотрим каждую часть из нашего примера.

ImageFilterDemo.java

Класс ImageFilterDemo является каркасом аплета для наших примеров применения фильтров изображений. Он потребляет диспетчер BorderLayout и имеет панель Panel в позиции *South* для хранения кнопок, которые будут представлять каждый фильтр. Объект Label занимает позицию *North* для информационных сообщений о ходе работы фильтра. В позиции *Center* помещается изображение (которое инкапсулируется в подклассе Canvas LoadedImage, о котором упоминалось ранее). Кнопки/фильтры дескриптора filters<param>, разделенные с помощью +, анализируются с применением StringTokenizer.

Метод actionPerformed() интересен тем, что он использует метку кнопки в качестве имени класса фильтра, который он пытается загрузить с помощью (PlugInFilter) Class.forName(a).newInstance(). Этот метод является надежным и выполняет определенное действие, если кнопка не соответствует подходящему классу, реализующему PlugInFilter.

```
/*
 * <applet code=ImageFilterDemo width=350 height=450>
 * <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);
    }
  }
}
```

На рис. 25.7 показано, как выглядит аплет, когда он впервые загружается с использованием дескриптора аплета, показанного в начале исходного файла.

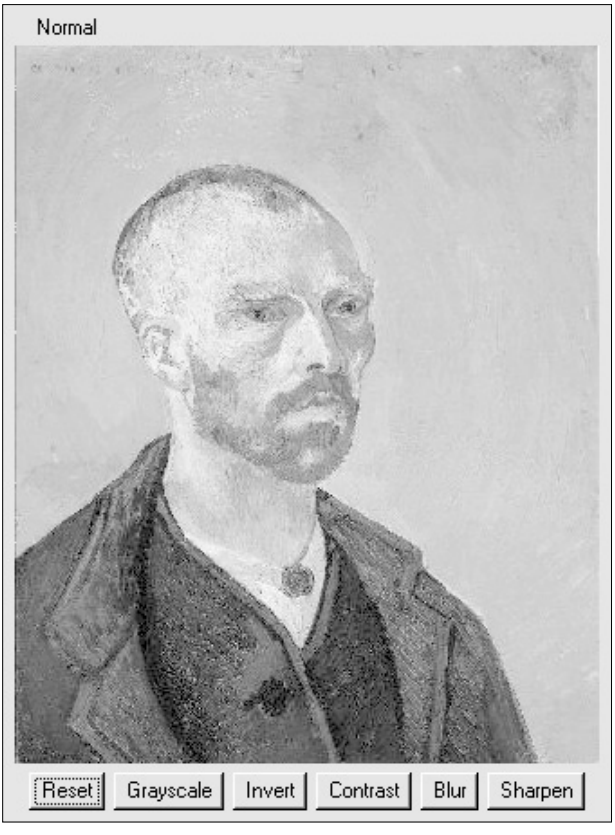

Рис. 25.7. Обычный вывод аплета `ImageFilterDemo`

PlugInFilter.java

`PlugInFilter` является простым интерфейсом, используемым для абстрактной фильтрации изображений. Он имеет только один метод, `filter()`, который принимает аплет и исходное изображение, и возвращает новое отфильтрованное изображение.

```
interface PlugInFilter {
  java.awt.Image filter(java.applet.Applet a, java.awt.Image in);
}
```

LoadedImage.java

`LoadedImage.java` является удобным классом `Canvas`, который принимает изображение во время его формирования и синхронно загружает его с помощью `MediaTracker`. В результате `LoadedImage` будет вести себя корректно внутри элемента управления `LayoutControl`, поскольку в нем переопределены методы `getPreferredSize()` и `getMinimumSize()`. Также он имеет метод `set()`, служащий для определения нового изображения `Image`, которое необходимо отобразить на данном холсте `Canvas`. Так отфильтрованное изображение отображается после завершения вставки.

```
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();
  }
}
```

Grayscale.java

Фильтр `Grayscale` является подклассом `RGBImageFilter`. Это означает, что `Grayscale` может использоваться в качестве параметра `ImageFilter` для конструктора `FilteredImageSource`. Единственное, что необходимо будет сделать — это переопределить метод `filterRGB()`, чтобы изменить входные значения цветовых составляющих. Он принимает значения красной, зеленой и синей составляющих и вычисляет яркость пикселя с помощью коэффициента преобразования цвета в яркость, утвержденного комитетом NTSC (National Television Standards Committee — Национальный комитет по телевизионным стандартам). Затем он просто возвращает серый пиксель, имеющий ту же яркость, что и источник цвета.

```
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);
  }
}
```

Invert.java

Фильтр `Invert` также является достаточно простым фильтром. Он принимает отдельно красный, зеленый и синий каналы, а затем инвертирует их, вычитая их из 255. Инвертированные значения переносятся обратно в значение пикселя и возвращаются.

```
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);
  }
}
```

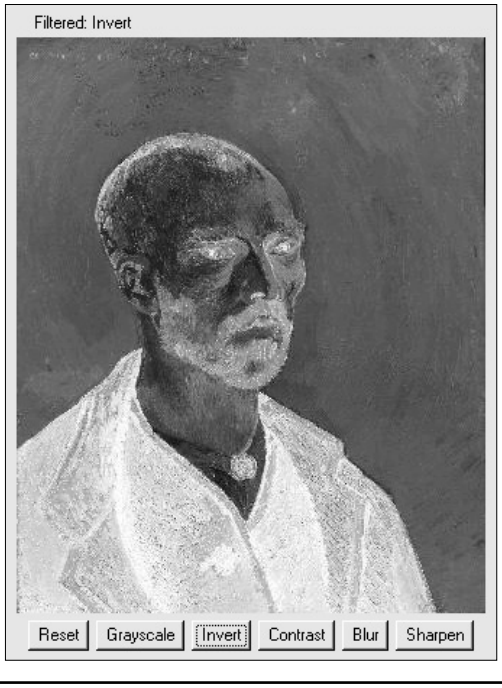

Рис. 25.8. Использование фильтра `Invert` в аплете `ImageFilterDemo`

Contrast.java

Фильтр `Contrast` очень похож на фильтр `Grayscale` за исключением того, что замена метода `filterRGB()` производится несколько сложнее. В алгоритме, который он применяет для улучшения контрастности, принимаются значения отдельно для красной, зеленой и синей составляющих, и увеличиваются в 1,2 раза, если их яркость выше 128. Если их яркость ниже 128, они делятся на 1,2. Увеличенные значения фиксируются в значении 255 с помощью метода `multiclamp()`.

```
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 multclamp(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);
  }
}
```

На рис. 25.9 показано изображение после щелчка на кнопке Contrast (Контрастность).

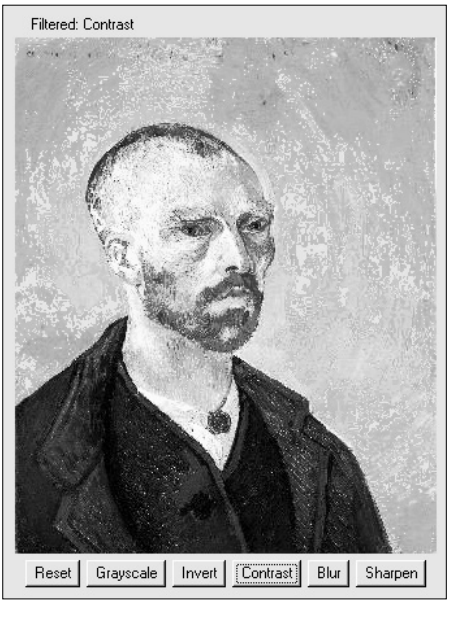

Рис. 25.9. Использование фильтра `Contrast` в аплете `ImageFilterDemo`

Convolver.java

Абстрактный класс `Convolver` обрабатывает основные операции фильтра свертки, реализуя интерфейс `ImageConsumer` для переноса исходных пикселей в массив `imgpixels`. Для отфильтрованных данных он создает второй массив `newimgpixels`. Фильтры свертки выбирают небольшие прямоугольные области пикселей вокруг каждого пикселя в изображении, которые называются *ядром свертки*. Эта область, которая в данном примере имеет размер 3×3, используется для принятия решения, каким образом будет изменяться центральный пиксель в области.

На заметку! *Фильтр не может модифицировать массив `imgpixels` по той причине, что следующий пиксель в строке развертки будет пытаться использовать исходное значение для предыдущего пикселя, которое уже может быть отфильтровано.*

Два конкретных подкласса, представленных ниже, очень просто реализуют метод `convolver()`, используя `imgpixels` для хранения исходных данных и `newimgpixels` — для результатов.

```
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 начинается фильтр...

  public Image filter(Applet a, Image in) {
    in.getSource().startProduction(this);
    imageReady = false;
    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;
    }
  }
}
```

Blur.java

Фильтр `Blur.java` является подклассом `Convolver` и работает с каждым пикселем в массиве исходного изображения `imgpixels`, вычисляя среднее по окружающей его области размером 3×3. Соответствующий выходной пиксель в `newimgpixels` является подсчитанным средним значением.

```
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);
      }
    }
  }
}
```

На рис. 25.10 показан аплет после работы фильтра `Blur`.

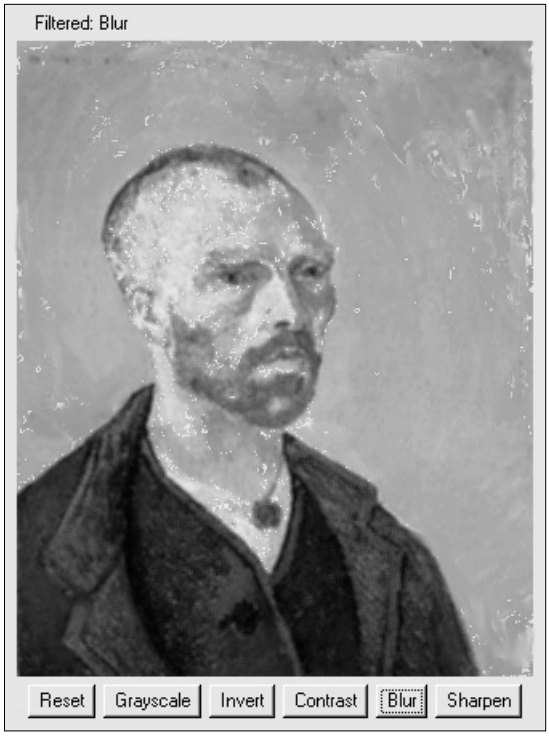

Рис. 25.10. Использование фильтра `Blur` в аплете `ImageFilterDemo`

Sharpen.java

Фильтр `Sharpen` тоже является подклассом `Convolver` и (в той или иной степени) прямой противоположностью `Blur`. Он работает с каждым пикселем в массиве исходного изображения `imgpixels` и вычисляет среднее по окружающей его области размером 3×3. Соответствующий выходной пиксель в `newimgpixels` отличается от центрального пикселя и окружающего среднего, добавленного в него. По сути, это свидетельствует о том, что если пиксель ярче на 30, чем его окружающие пиксели, то остальные пиксели станут ярче на 30. Если же он темнее на 10, то остальные станут темнее на 10. Благодаря этому будут акцентироваться края, оставляя ровные участки неизмененными.

```
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));
      }
    }
  }
}
```

На рис. 25.11 показан аплет после работы фильтра `Sharpen`.

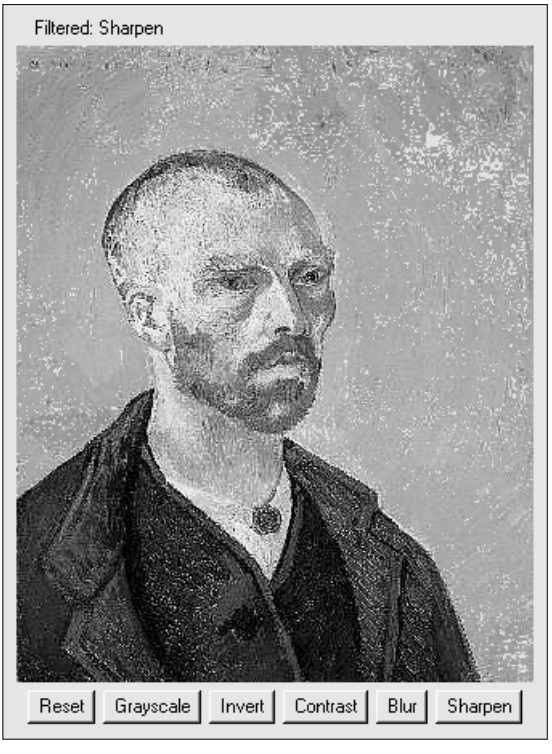

Рис. 25.11. Использование фильтра `Sharpen` в аплете `ImageFilterDemo`

Аппликационная анимация

Теперь, когда вы уже знаете об особенностях API-интерфейсов работы с изображениями, мы можем построить интересный аплет, который будет отображать последовательность анимационных ячеек. Анимационные ячейки берутся из одного изображения, которое может компоновать ячейки в сетке, определяемой параметрами `rows` и `cols` дескриптора `<param>`. Каждая ячейка в изображении разделяется подобно тому, как это было использовано ранее в примере `TileImage`. Мы получаем последовательность, в которой будут отображаться ячейки, из параметра `sequence` дескриптора `<param>`. Она представляет собой список номеров ячеек, разделенных запятыми, которые начинаются с нуля и следуют далее по сетке слева направо и сверху вниз.

После того как аплет проанализирует дескрипторы `<param>` и загрузит исходное изображение, он расчленит изображение на несколько небольших второстепенных изображений. Затем запускается поток, благодаря которому изображения будут отображаться в порядке, описанном в `sequence`. Поток бездействует достаточно времени, чтобы поддерживать частоту кадров `framerate`. Ниже показан исходный код.

```
// Пример анимации.
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
import java.util.*;
```

```
public class Animation extends Applet implements Runnable {
  Image cell[];
  final int MAXSEQ = 64;
  int sequence[];
  int nseq;
  int idx;
  int framerate;
  boolean stopFlag;

  private int intDef(String s, int def) {
    int n = def;
    if (s != null)
      try {
        n = Integer.parseInt(s);
      } catch (NumberFormatException e) {
        System.out.println("Number Format Exception");
      }
    return n;
  }
  public void init() {
    framerate = intDef(getParameter("framerate"), 5);
    int tilex = intDef(getParameter("cols"), 1);
    int tiley = intDef(getParameter("rows"), 1);
    cell = new Image[tilex*tiley];

    StringTokenizer st = new StringTokenizer(getParameter("sequence"), ",");
    sequence = new int[MAXSEQ]
    nseq = 0;
    while(st.hasMoreTokens() && nseq < MAXSEQ) {
      sequence[nseq] = intDef(st.nextToken(), 0);
      nseq++;
    }
    try {
      Image img = getImage(getDocumentBase(), getParameter("img"));
      MediaTracker t = new MediaTracker(this);
      t.addImage(img, 0);
      t.waitForID(0);
      int iw = img.getWidth(null);
      int ih = img.getHeight(null);
      int tw = iw / tilex;
      int th = ih / tiley;
      CropImageFilter f;
      FilteredImageSource fis;
        for (int y=0; y<tiley; y++) {
          for (int x=0; x<tilex; x++) {
            f = new CropImageFilter(tw*x, th*y, tw, th);
            fis = new FilteredImageSource(img.getSource(), f);
            int i = y*tilex+x;
            cell[i] = createImage(fis);
            t.addImage(cell[i], i);
          }
        }
        t.waitForAll();
      } catch (InterruptedException e) {
        System.out.println("Image Load Interrupted");
      }
    }
```

```
    public void update(Graphics g) { }
    public void paint(Graphics g) {
      g.drawImage(cell[sequence[idx]], 0, 0, null);
    }
    Thread t;
    public void start() {
      t = new Thread(this);
      stopFlag = false;
      t.start();
    }
    public void stop() {
      stopFlag = true;
    }
    public void run() {
      idx = 0;
      while (true) {
        paint(getGraphics());
        idx = (idx + 1) % nseq;
        try {
          Thread.sleep(1000/framerate);
        } catch (InterruptedException e) {
          System.out.println("Animation Interrupted");
          return;
        }
        if(stopFlag)
          return;
      }
    }
}
```

Следующий дескриптор аплета иллюстрирует известное учение Эдварда Майбриджа (Eadweard Muybridge) о движении, которое гласит, что в действительности лошади касаются земли всеми четырьмя копытами. (Естественно, в вашем аплете вы можете использовать другой файл с изображением.)

```
<applet code=Animation width=67 height=48>
<param name=img value=horse.gif>
<param name=rows value=3>
<param name=cols value=4>
<param name=sequence value=0,1,2,3,4,5,6,7,8,9,10>
<param name=framerate value=15>
</applet>
```

На рис. 25.12 показано выполнение аплета. Обратите внимание на то, что в исходном изображении, загруженном после аплета, используется обычный дескриптор `<img>`.

Дополнительные классы обработки изображений

Кроме классов обработки изображений, описанных в этой главе, пакет `java.awt.image` содержит несколько других классов, которые предлагают расширенные возможности управления процессом визуализации изображений и поддерживают усовершенство-

ванные технологии визуализации. Имеется также пакет визуализации изображений `javax.imageio`, поддерживающий дополнения, с помощью которых можно реализовать обработку различных форматов изображений. Если вас интересуют возможности усовершенствованного графического вывода, вам придется изучить дополнительные классы, которые можно найти в пакетах `javax.awt.image` и `javax.imageio`.

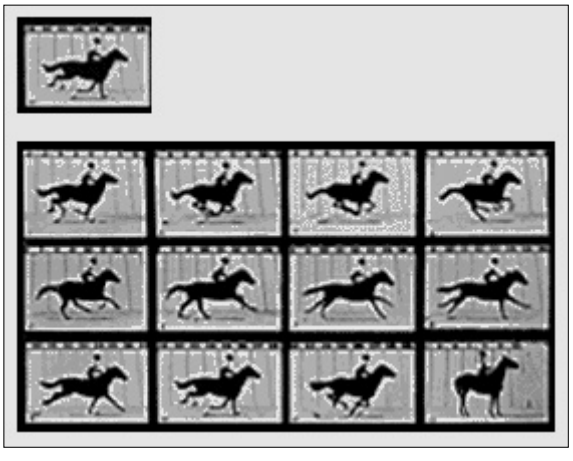

Рис. 25.12. Результат выполнения аплета `Animation`

26
ГЛАВА

Параллельные утилиты

С самого начала в Java была предусмотрена встроенная поддержка многопоточности и синхронизации. Например, новые потоки можно создавать путем реализации интерфейса `Runnable` или расширения класса `Thread`, синхронизация доступна посредством использования ключевого слова `synchronized`, а межпотоковые коммуникации поддерживаются методами `wait()` и `notify()`, которые определены в классе `Object`. Вообще говоря, эта встроенная поддержка многопоточности была одним из наиболее важных новшеств в Java и остается одной из главных ее сильных сторон.

Однако какой бы концептуально "чистой" ни была поддержка многотопочности в Java, она не является идеальной для всех приложений, особенно для тех, в которых широко используется множество потоков. Например, первоначальная поддержка многопоточности лишена некоторых высокоуровневых функциональных возможностей (например, семафоров, пулов потоков и диспетчеров), которые способствуют созданию программ, работающих в параллельном режиме.

Необходимо понять с самого начала, что многопоточность используется во многих Java-программах, которые в результате становятся "параллельными" (concurrent). Например, многопоточность используется в большинстве аплетов. Однако что касается настоящей главы, то термин *параллельная программа* относится к программе, которая в *полной мере* использует параллельно выполняющиеся потоки, являющиеся ее *неотъемлемой частью*. Примером такой программы является программа, где отдельные потоки служат для одновременного вычисления частичных результатов, используемых в более крупных расчетах. Другим примером является программа, координирующая активность нескольких потоков, каждый из которых пытается обратиться к информации в базе данных. В этом случае доступ в режиме только для чтения может обрабатываться отдельно от доступа, при котором необходимо обеспечить чтение и запись.

Для поддержки параллельных программ в JDK 5 были добавлены *параллельные утилиты*, которые также часто упоминаются как *параллельный API*. Параллельные утилиты предоставляют множество возможностей, о которых уже давно мечтали программисты, занимающиеся разработкой параллельных приложений. Например, они предлагают семафоры, циклические барьеры, защелки с обратным отсчетом, пулы потоков, диспетчеры выполнения, блокировки, множество параллельных коллекций, а также элегантный способ использования потоков для получения результатов вычислений.

Параллельный API достаточно большой, и многие вопросы, связанные с его использованием, являются довольно-таки сложными. Он выходит за рамки контекста настоящей книги. Более того, альтернативные варианты, предлагаемые параллельными утилитами, не предназначены для использования в большинстве программ. Вы должны просто иметь в виду, что если только вы не пишете программы с множеством параллельных процессов, то в большинстве случаев традиционной поддержки многопоточности и синхронизации Java будет вполне достаточно, а во многих случаях предпочтение отдается именно ей, а не параллельному API.

И все же каждому программисту важно иметь общее рабочее представление о параллельном API. Кроме того, существуют некоторые его части, такие как синхронизаторы, вызываемые потоки и исполнители, которые во многих случаях могут оказаться исключительно полезными. В связи с этим в этой главе приводится обзор параллельных утилит вместе с примерами их применения.

Пакеты параллельного API

Параллельные утилиты содержатся в пакете `java.util.concurrent` и в его двух подпакетах, `java.util.concurrent.atomic` и `java.util.concurrent.locks`. Далее следует краткий обзор их содержимого.

java.util.concurrent

Пакет `java.util.concurrent` определяет основные функциональные возможности, которые поддерживают альтернативные варианты встроенных методов синхронизации и межпотоковых коммуникаций. Он определяет следующие ключевые функциональные особенности:

- синхронизаторы;
- исполнители;
- параллельные коллекции.

Синхронизаторы предлагают высокоуровневые способы синхронизации взаимодействий между несколькими потоками. В пакете `java.util.concurrent` определен набор классов синхронизаторов, описанных в табл. 26.1.

Таблица 26.1. Классы синхронизаторов, определенные в пакете `java.util.concurrent`

Класс	Описание
`Semaphore`	Реализует классический семафор.
`CountDownLatch`	Ожидание длится до тех пор, пока не произойдет определенное количество событий.
`CyclicBarrier`	Позволяет группе потоков войти в режим ожидания в предварительно заданной точке выполнения.
`Exchanger`	Осуществляет обмен данными между двумя потоками.

Обратите внимание, что каждый синхронизатор предлагает решение задачи синхронизации определенного рода. Благодаря этому можно оптимизировать работу каждого синхронизатора. Раньше эти типы объектов синхронизации необходимо было конструировать вручную. Параллельный API стандартизирует их и делает доступными для всех программистов, работающих с Java.

Исполнители управляют выполнением потоков. Первым в иерархии исполнителей является интерфейс `Executor`, который служит для запуска потока. `ExecutorService` расширяет `Executor` и предлагает методы, управляющие выполнением. Существуют две реализации `ExecutorService`: `ThreadPoolExecutor` и `ScheduledThreadPoolExecutor`. Пакет `java.util.concurrent` также определяет служебный класс `Executors`, который содержит несколько статических методов, упрощающих создание разнообразных исполнителей.

С исполнителями связаны интерфейсы `Future` и `Callable`. `Future` содержит значение, возвращаемое потоком после его выполнения. Таким образом, его значение определяется "в будущем", когда поток завершит свое выполнение. `Callable` определяет поток, возвращающий значение.

`java.util.concurrent` определяет несколько параллельных классов коллекций, включая `ConcurrentHashMap`, `ConcurrentLinkedQueue` и `CopyOnWriteArraylist`. Они предлагают параллельные альтернативные варианты для связанных с ними классов, определенных в каркасе Collections Framework.

Наконец, для улучшенной обработки синхронизации потоков пакет `java.util.concurrent` определяет перечисление `TimeUnit`.

java.util.concurrent.atomic

`java.util.concurrent.atomic` упрощает использование переменных в параллельной среде. Он предлагает средства эффективного обновления значений переменной без применения блокировок. Для этих целей используются такие классы, как `AtomicInteger` и `AtomicLong`, и методы вроде `compareAndSet()`, `decrementAndGet()` и `getAndSet()`. Эти методы работают в режиме одной непрерывной операции.

java.util.concurrent.locks

Пакет `java.util.concurrent.locks` предлагает альтернативный вариант работы с методами `synchronized`. В его основе лежит интерфейс `Lock`, определяющий основной механизм, который используется для получения доступа к объекту и отказа в доступе. Ключевыми методами являются `lock()`, `tryLock()` и `unlock()`. Преимущество этих методов состоит в том, что они расширяют возможности управления синхронизацией.

В оставшейся части этой главы мы займемся подробным рассмотрением компонентов параллельного API.

Использование объектов синхронизации

Вполне возможно, что наиболее широко используемой частью параллельного API будут его объекты синхронизации. Они поддерживаются классами `Semaphore`, `CountDownLatch`, `CyclicBarrier` и `Exchanger`. Вместе они позволяют без особых сложностей решать некоторые задачи синхронизации, справиться с которыми ранее было довольно-таки непросто. Их также можно применять к широкому диапазону программ — даже к тем, которые поддерживают только ограниченный параллелизм. Поскольку объекты синхронизации могут встречаться практически во всех Java-программах, остановимся на них более подробно.

Semaphore

Первым объектом синхронизации, который могут сразу же вспомнить большинство читателей, является `Semaphore`, реализующий классический семафор. Семафор управляет доступом к общему ресурсу с помощью счетчика. Если счетчик больше нуля, доступ разрешается. Если он равен нулю, в доступе будет отказано. В действительности этот счетчик подсчитывает *разрешения*, открывающие доступ к общему ресурсу. Следовательно, чтобы получить доступ к ресурсу, поток должен получить разрешение на доступ у семафора.

В общем случае, чтобы использовать семафор, поток, которому требуется доступ к общему ресурсу, пытается получить разрешение. Если значение счетчика семафора будет больше нуля, поток получит разрешение, после чего значение счетчика семафора уменьшается на единицу. В противном случае поток будет заблокирован до тех пор, пока он не сможет получить разрешение. Если доступ к общему ресурсу потоку больше не нужен, он освобождает разрешение, в результате чего значение счетчика семафора увеличивается на единицу. Если в это время другой поток ожидает разрешения, то он сразу же его получает. Описанный механизм реализуется в Java классом `Semaphore`.

`Semaphore` имеет два конструктора, показанных далее:

```
Semaphore(int num)
Semaphore(int num, boolean how)
```

Здесь параметр *num* указывает исходное значение счетчика разрешений. Таким образом, *num* определяет количество потоков, которым может быть предоставлен доступ к общему ресурсу в любое определенное время. Если параметр *num* будет равен единице, только один поток сможет обратиться к ресурсу в любое определенное время. По умолчанию ожидающим потокам предоставляется разрешение в неопределенном порядке. Если параметру *how* присвоить значение `true`, то вы тем самым определите, что ожидающим потокам будут выдаваться разрешения в том порядке, в каком они запрашивали доступ.

Чтобы получить разрешение, вызовите метод `acquire()`, который имеет следующие две формы:

```
void acquire() throws InterruptedException
void acquire(int num) throws InterruptedException
```

Первая форма запрашивает одно разрешение, а вторая — *num* разрешений. Обычно используется первая форма. Если разрешение не будет предоставлено во время вызова метода, то вызывающий поток будет приостановлен до тех пор, пока не будет получено разрешение.

Чтобы освободить разрешение, вызовите метод `release()`, который имеет следующие две формы:

```
void release()
void release(int num)
```

Первая форма освобождает одно разрешение. Вторая форма освобождает то количество разрешений, которое указано в параметре *num*.

Чтобы использовать семафор для управления доступом к ресурсу, каждый поток, которому необходимо использовать этот ресурс, должен вначале вызвать метод `acquire()`, прежде чем обращаться к ресурсу. Когда поток завершает свою работу с ресурсом, он должен вызвать метод `release()`. Ниже приведен пример использования семафора.

```
// Пример простого 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 выполнения, 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 " + name);
  try {
    // Сначала получаем разрешение.
    System.out.println(name + " ожидает разрешения.");
    sem.acquire();
    System.out.println(name + " получает разрешение.");
      // Теперь обращаемся к общему ресурсу.
      for(int i=0; i < 5; i++) {
        Shared.count++;
        System.out.println(name + ": " + Shared.count);
        // Если это возможно, разрешаем контекстное переключение.
        Thread.sleep(10);
      }
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }
    // Освобождаем разрешение.
    System.out.println(name + " освобождает разрешение.");
    sem.release();
  }
}
// Поток выполнения, 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("Starting " + name);
    try {
    // Сначала получаем разрешение.
    System.out.println(name + " ожидает разрешения.");
    sem.acquire();
    System.out.println(name + " получает разрешение.");
    // Теперь обращаемся к общему ресурсу.
    for(int i=0; i < 5; i++) {
      Shared.count--;
      System.out.println(name + ": " + Shared.count);
        // Если это возможно, разрешаем контекстное переключение.
        Thread.sleep(10);
      }
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }
    // Освобождаем разрешение.
    System.out.println(name + " освобождает разрешение.");
    sem.release();
  }
}
```

Ниже показаны результаты выполнения этой программы:

```
Запуск A
A ожидает разрешения.
A получает разрешение.
A: 1
Запуск B
B ожидает разрешения.
A: 2
A: 3
A: 4
A: 5
A освобождает разрешение.
B получает разрешение.
B: 4
B: 3
B: 2
B: 1
B: 0
B освобождает разрешение.
```

В программе используется семафор для управления доступом к переменной `count`, которая является статической переменной класса `Shared`. `Shared.Count` увеличивается на 5 в методе `run()` из `IncThread` и уменьшается на 5 в одноименном методе в `DecThread`. Для защиты этих двух потоков от доступа к `Shared.count` в одно и то же время доступ предоставляется только после того, как будет получено разрешение от управляющего семафора. После того как доступ будет завершен, разрешение освобождается. Таким образом, только один поток одновременно получит доступ к `Shared.count`, что можно видеть из результатов вывода.

Обратите внимание на то, что в IncThread и DecThread в методе run() вызывается метод sleep(). Он "гарантирует", что доступы к Shared.count будут синхронизироваться семафором. В методе run() вызов метода sleep() приводит к тому, что вызывающий поток будет приостанавливаться в промежутках между каждым доступом к Shared.count. Как правило, благодаря этому должен выполняться второй поток. Однако поскольку мы работаем с семафором, то второй поток должен ожидать до тех пор, пока первый поток не освободит разрешение, а это происходит только после того, как будут завершены все доступы, производимые первым потоком. Таким образом, Shared.count вначале увеличивается на 5 в IncThread, а затем уменьшается на 5 в DecThread. Увеличения и уменьшения значений происходят *строго по порядку*.

Если бы мы не использовали семафор, то доступы к Shared.count, производимые каждым потоком, осуществлялись бы одновременно, поэтому увеличение и уменьшение значения происходило бы вперемешку. Чтобы убедиться в этом, попробуйте закомментировать вызовы методов acquire() и release(). Запустив программу, вы увидите, что доступ к Shared.count больше не является синхронизированным, и каждый поток обращается к Shared.count сразу же, как только выделяется временной интервал.

Несмотря на то что во многих случаях применение семафора не представляет сложности, как это можно было видеть в предыдущей программе, возможны также и более сложные варианты его использования. Ниже показан один из таких примеров. Это переработанная версия программы поставщика и потребителя, представленной в главе 11. Здесь используется два семафора, которые регулируют потоки поставщика и потребителя и гарантируют, что за каждым вызовом 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;
  // Начинаем 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("Произошло InterruptedException");
    }

     System.out.println("Получено: " + n);
     semProd.release();
  }
  void put(int n) {
    try {
      semProd.acquire();
    } catch(InterruptedException e) {
      System.out.println("Произошло 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 выходных результатов:

```
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, здесь происходит синхронизация вызовов методов `put()` и `get()`. То есть после каждого вызова метода `put()` следует вызов метода `get()`, поэтому ни одно значение не может быть пропущено. Если бы семафоры не использовались, вызовы метода `put()` могли бы происходить без чередования с вызовами метода `get()`, в результате чего значения были бы пропущены. (Чтобы убедиться в этом, удалите код семафора и посмотрите на полученные результаты.)

Последовательность вызовов методов put() и get() обрабатывается двумя семафорами: semProd и semCon. Прежде чем метод put() сможет произвести значение, он должен получить разрешение от семафора semProd. После того как значение будет определено, он освобождает semProd. Прежде чем метод get() сможет использовать значение, он должен получить разрешение от семафора semCon. После того как он закончит работу с этим значением, он освобождает semCon. Такой механизм "передачи и принятия" гарантирует, что за каждым вызовом метода put() будет следовать вызов метода get().

Обратите внимание, что семафор semCon инициализируется без доступных разрешений. Поэтому метод put() выполняется первым. Возможность задавать исходное состояние синхронизации является одной из наиболее ярких характеристик семафоров.

CountDownLatch

Иногда бывает необходимо сделать так, чтобы поток находился в режиме ожидания до тех пор, пока не произойдет одно или несколько событий. Для этих целей параллельный API предлагает защелку CountDownLatch. CountDownLatch изначально создается с количеством событий, которые должны произойти до того момента, как будет снята защелка. Каждый раз, когда происходит событие, значение счетчика уменьшается. Когда значение счетчика станет равным нулю, защелка будет снята.

CountDownLatch имеет следующий конструктор:

```
CountDownLatch(int num)
```

Здесь *num* определяет количество событий, которые должны произойти для того, чтобы снять защелку.

Для обслуживания защелки поток вызывает метод await(), который имеет следующие формы:

```
void await() throws InterruptedException
void await(long wait, TimeUnit tu) throws InterruptedException
```

В первом случае ожидание длится до тех пор, пока значение счета, связанного с вызывающим CountDownLatch, не станет равно нулю. Во втором случае ожидание длится только в течение определенного периода времени, определенного параметром *wait*. Единицы, представляемые этим параметром, определяются в *tu*, который является объектом перечисления TimeUnit. (TimeUnit рассматривается далее в этой главе.)

Чтобы сигнализировать о событии, нужно вызвать метод countDown():

```
void countDown()
```

При каждом вызове метода countDown() значение счета, связанного с вызывающим объектом, уменьшается на единицу.

В следующей программе представлен пример применения CountDownLatch. В ней создается защелка, снять которую можно будет только по прошествии пяти событий.

```
// Демонстрация применения CountDownLatch.

import java.util.concurrent.CountDownLatch;

class CDLDemo {
  public static void main(String args[]) {
    CountDownLatch cdl = new CountDownLatch(5);

    System.out.println("Запуск");

    new MyThread(cdl);
```

```
    try {
      cdl.await();
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }
    System.out.println("Завершение");
  }
}
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 показаны результаты выполнения этой программы.

```
Запуск
0
1
2
3
4
Завершение
```

Внутри метода `main()` создается защелка `CountDownLatch` по имени `cdl`, исходное значение которой равно 5. После этого создается экземпляр `MyThread`, который начинает выполнение нового потока. Обратите внимание, что `cdl` передается в качестве параметра конструктору `MyThread` и сохраняется в переменной экземпляра `latch`. Затем главный поток вызывает метод `await()` для `cdl`, в результате чего выполнение главного потока приостанавливается до тех пор, пока счетчик `cdl` пять раз не уменьшится на единицу.

Внутри метода `run()` конструктора `MyThread` создается цикл, который повторяется пять раз. Во время каждого повторения вызывается метод `countDown()` для `latch`, который ссылается на `cdl` в методе `main()`. После пятого повторения защелка снимается, позволяя возобновить главный поток.

`CountDownLatch` является мощным и простым в использовании объектом синхронизации, который будет полезен в тех случаях, когда поток должен находиться в режиме ожидания, пока не произойдет одно или несколько событий.

CyclicBarrier

В программировании нередко возникают такие ситуации, когда два или более потоков должны находиться в режиме ожидания в предварительно определенной точке выполнения до тех пор, пока все потоки из этого набора не достигнут этой точки. Для этих целей параллельный API предлагает класс `CyclicBarrier`. Он позволяет определить объект

синхронизации, который приостанавливается до тех пор, пока определенное количество потоков не достигнет барьерной точки.

`CyclicBarrier` имеет следующие два конструктора:

```
CyclicBarrier(int numThreads)
CyclicBarrier(int numThreads, Runnable action)
```

Здесь параметр *numThreads* определяет количество потоков, которые должны достигнуть барьера до того, как выполнение будет продолжено. Во втором варианте параметр *action* определяет поток, который будет выполняться по достижении барьера.

Общая процедура использования `CyclicBarrier` выглядит следующим образом. Во-первых, необходимо создать объект `CyclicBarrier`, указав количество потоков для ожидания. Затем, когда каждый из потоков достигнет барьера, нужно вызвать метод `await()` для данного объекта. В результате этого выполнение потока будет приостановлено до тех пор, пока все остальные потоки также не вызовут метод `await()`. После того как указанное количество потоков достигнет барьера, метод `await()` вернет результат, и выполнение будет возобновлено. Кроме того, если определить какое-нибудь действие, то этот поток будет выполнен.

Метод `await()` имеет следующие две формы:

```
int await() throws InterruptedException, BrokenBarrierException
int await(long wait, TimeUnit tu) throws InterruptedException,
BrokenBarrierException, TimeoutException
```

В первом случае ожидание длится до тех пор, пока каждый из потоков не достигнет барьерной точки. Во втором случае ожидание длится в течение определенного периода времени *wait*. Единицы времени этого параметра определяются параметром *tu*. В обоих случаях возвращается значение, показывающее порядок, в соответствии с которым потоки будут достигать барьерной точки. Первый поток возвращает значение, равное количеству ожидаемых потоков минус 1. Последний поток возвращает нуль.

Ниже показан пример, иллюстрирующий `CyclicBarrier`. Он ожидает, пока совокупность трех потоков не достигнет барьерной точки. После того как это произойдет, будет выполнен поток, определяемый посредством `BarAction`.

```
// Демонстрация применения CyclicBarrier.

import java.util.concurrent.*;

class BarDemo {
  public static void main(String args[]) {
    CyclicBarrier cb = new CyclicBarrier(3, new BarAction() );

    System.out.println("Запуск");

    new MyThread(cb, "A");
    new MyThread(cb, "B");
    new MyThread(cb, "C");

  }
}

// Поток выполнения, использующий 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 после завершения выполнения CyclicBarrier.
class BarAction implements Runnable {
  public void run() {
    System.out.println("Барьер достигнут!");
  }
}
```

Ниже показаны результаты выполнения.

```
Запуск
A
B
C
Барьер достигнут!
```

`CyclicBarrier` можно использовать повторно, так как он освобождает ожидающие потоки каждый раз, когда определенное количество потоков вызывает метод `await()`. Например, если метод `main()` в предыдущей программе изменить следующим образом:

```
public static void main(String args[]) {
CyclicBarrier cb = new CyclicBarrier(3, new BarAction() );
  System.out.println("Запуск");
  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 выполнения будет таким:

```
Запуск
A
B
C
Барьер достигнут!
X
Y
Z
Барьер достигнут!
```

Как можно видеть из предыдущего примера, CyclicBarrier предлагает элегантное решение ранее сложной задачи.

Exchanger

Вероятно, наиболее интересным классом синхронизации является Exchanger. Он предназначен для упрощения процесса обмена данными между двумя потоками. Работа класса Exchanger довольно-таки проста: он просто ожидает, пока два отдельных потока не вызовут его метод exchange(). Как только это произойдет, он произведет обмен данными, имеющимися у обоих потоков. В своем использовании этот механизм является одновременно и элегантным, и простым. Варианты применения класса Exchanger можно представить очень просто. Например, один поток подготавливает буфер для получения информации через сетевое соединение. Другой поток заполняет этот буфер информацией, получаемой посредством соединения. Оба потока работают совместно, поэтому каждый раз, когда возникает необходимость в использовании нового буфера, осуществляется обмен данными.

Класс Exchanger является обобщенным классом и имеет следующее объявление:

```
Exchanger<V>
```

Здесь V определяет тип данных для обмена.

Класс Exchanger определяет единственный метод exchange(), который имеет следующие две формы:

```
V exchange(V buffer) throws InterruptedException

V exchange(V buffer, long wait, TimeUnit tu) throws InterruptedException,
TimeoutException
```

Здесь параметр *buffer* представляет ссылку на данные для обмена. Данные, полученные из другого потока, возвращаются. Вторая форма метода exchange() позволяет определить время простоя. Ключевая особенность метода exchange() заключается в том, что его выполнение не будет продолжено до тех пор, пока он не будет вызван для одного и того же объекта Exchanger двумя отдельными потоками. Таким образом, exchange() синхронизирует обмен данными.

Ниже показан пример применения класса Exchanger. В нем создается два потока. Один поток создает пустой буфер, который получает данные, занесенные в него другим потоком. Таким образом, первый поток меняет пустой поток на полный.

```
// Пример использования 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 Thread, формирующий 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++) {
      // Заполнение буфера
      for(int j = 0; j < 5; j++)
        str += (char) ch++;
      try {
        // Заполненный буфер становится пустым.
        str = ex.exchange(str);
      } catch(InterruptedException exc) {
        System.out.println(exc);
      }
    }
  }
}
// Поток Thread, использующий 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 {
        // Пустой буфер становится заполненным.
        str = ex.exchange(new String());
        System.out.println("Получено: " + str);
      } catch(InterruptedException exc) {
        System.out.println(exc);
      }
    }
  }
}
```

Ниже показаны результаты выполнения этой программы.

```
Получено: ABCDE
Получено: FGHIJ
Получено: KLMNO
```

В этой программе метод `main()` создает объект `Exchanger` для строк. Этот объект затем служит для синхронизации обмена строками между классами `MakeString` и `UseString`. Класс `MakeString` заполняет строку данными. Класс `UseString` заполняет пустой буфер. Затем он отображает содержимое только что созданной строки. Замена пустого буфера полным синхронизируется с помощью метода `exchange()`, который вызывается методом `run()` обоих классов.

Использование исполнителя

Параллельный API поддерживает функциональную возможность, называемую *исполнителем* (executor), которая генерирует потоки и управляет ими. По сути, исполнитель предлагает альтернативный вариант управления потоками с помощью класса `Thread`.

В основе исполнителя лежит интерфейс `Executor`, который определяет следующий метод:

```
void execute(Runnable thread)
```

Выполняется поток, указанный в параметре *thread*. Таким образом, метод `execute()` запускает заданный поток.

Интерфейс `ExecutorService` расширяет `Executor` за счет добавления методов, помогающих управлять и контролировать выполнение потоков. Например, `ExecutorService` определяет метод `shutdown()`, показанный ниже, который останавливает все потоки, находящиеся в данный момент под управлением `ExecutorService`.

```
void shutdown()
```

`ExecutorService` также определяет методы, которые выполняют потоки, возвращающие результаты, которые выполняют совокупность потоков и определяют состояние останова. Некоторые из этих методов мы рассмотрим чуть позже.

Определяется также и интерфейс `ScheduledExecutorService`, который расширяет `ExecutorService` в целях поддержки планирования потоков.

В параллельном API имеются два предварительно определенных класса исполнителей: `ThreadPoolExecutor` и `ScheduledThreadPoolExecutor`. Класс `ThreadPoolExecutor` реализует интерфейсы `Executor` и `ExecutorService` и обеспечивает поддержку управляемого пула потоков. Класс `ScheduledThreadPoolExecutor` тоже реализует интерфейс `ScheduledExecutorService` для поддержки возможности планирования пула потоков.

Пул потоков предлагает набор потоков, которые служат для решения разнообразных задач. Вместо того чтобы каждая задача имела дело со своим собственным потоком, используются потоки из пула. Это позволяет сократить нагрузку, связанную с созданием множества отдельных потоков.

Хотя классы `ThreadPoolExecutor` и `ScheduledThreadPoolExecutor` можно использовать напрямую, чаще всего вам будет необходимо получать исполнителя посредством вызова одного из следующих статических фабричных методов, определенных в классе-утилите `Executors`. Далее показаны некоторые примеры:

```
static ExecutorService newCachedThreadPool()
static ExecutorService newFixedThreadPool(int numThreads)
static ScheduledExecutorService newScheduledThreadPool(int numThreads)
```

`newCachedThreadPool` создает пул потоков, который не только добавляет потоки по мере необходимости, но и по возможности повторно их использует.

`newFixedThreadPool` создает пул потоков, состоящий из определенного количества потоков. `newScheduledThreadPool` создает пул потоков, в котором можно осуществлять планирование потоков. Каждый из них возвращает ссылку на `ExecutorService`, который можно использовать для управления пулом.

Простой пример исполнителя

Прежде чем продолжить обсуждение далее, давайте рассмотрим простой пример применения исполнителя. В следующей программе создается фиксированный пул, содержащий два потока. Затем этот пул используется для выполнения четырех задач. Таким образом, четыре задачи разделяют два потока, находящихся в пуле. После того как задачи будут выполнены, пул закрывается и программа завершает выполнение.

```
// Простой пример, в котором используется исполнитель.

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");

    // Начало потоков.
    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");
  }
}

  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 показаны результаты выполнения программы.

```
Запуск
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
```

Судя по результатам, несмотря на то, что в пуле содержится всего два потока, выполняются все четыре задачи. Однако только две из них могут быть выполнены одновременно. Остальные должны ожидать, пока один из потоков пула не станет свободным, и поэтому его можно будет использовать.

Вызов метода `shutdown()` очень важен. Если бы его не было в 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.

Использование `Callable` и `Future`

Одним из наиболее важных и, без сомнений, ярких новшеств в параллельном API является новый интерфейс `Callable`. Он представляет поток, возвращающий значение. Приложение может использовать объекты `Callable` для вычисления результатов, которые затем будут возвращены вызывающему потоку. Это мощный механизм, поскольку он облегчает написание кода для множества различных числовых расчетов, в которых частичные результаты вычисляются одновременно. Его также можно использовать и для запуска потока, возвращающего код состояния, который свидетельствует об успешном выполнении потока.

Интерфейс `Callable` является обобщенным интерфейсом, который определяется следующим образом:

```
interface Callable<V>
```

Здесь `V` показывает тип данных, возвращаемых задачей. `Callable` определяет только один метод `call()`:

```
V call() throws Exception
```

Внутри метода `call()` определяется задача, которую требуется выполнить. После того как она будет выполнена, возвращается результат. Если результат невозможно вычислить, метод `call()` генерирует исключение.

Задача `Callable` решается с помощью `ExecutorService` посредством вызова метода `submit()`. Метод `submit()` может иметь три формы, однако для выполнения `Callable` используется только одна из них:

```
<T> Future<T> submit(Callable<T> task)
```

Здесь параметр *task* представляет объект `Callable`, который будет выполняться в своем собственном потоке. Результат возвращается через объект типа `Future`.

`Future` является обобщенным интерфейсом, представляющим значение, которое будет возвращено посредством объекта `Callable`. Поскольку это значение будет получено в некотором будущем, то имя интерфейса (`Future`) говорит само за себя. `Future` определяется следующим образом:

```
interface Future<V>
```

Здесь `V` определяет тип результата.

Чтобы получить значение, нужно вызвать метод `get()` интерфейса `Future`, который имеет следующие две формы:

```
V get()
throws InterruptedException, ExecutionException
V get(long wait, TimeUnit tu) throws InterruptedException,
ExecutionException, TimeoutException
```

В первом случае ожидание получения результатов длится бесконечно долго. Во втором случае можно указать период времени в параметре *wait*. Единицы периода времени в этом параметре задаются параметром *tu*, который представляет собой объект перечисления `TimeUnit`, рассматриваемый далее в этой главе.

В следующей программе показан пример применения интерфейсов `Callable` и `Future`. В ней будут созданы три задачи, выполняющий три разных вычисления. Первая задача возвращает суммарное значение, вторая находит длину гипотенузы прямоугольного треугольника с известными значениями длин его сторон, а третья определяет факториал для заданного значения. Все три вычисления производятся одновременно.

```
// Пример, в котором используется 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 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 side1, side2;

  Hypot(double s1, double s2) {
    side1 = s1;
    side2 = s2;
  }

  public Double call() {
    return Math.sqrt((side1*side1) + (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 приводятся результаты выполнения программы:

```
Запуск
55
5.0
120
Завершение
```

Перечисление TimeUnit

Параллельный API определяет методы, принимающие параметр типа TimeUnit, который служит для определения периода времени. TimeUnit является перечислением, которое используется для определения *временного разбиения* (или разрешения). TimeUnit определяется в пакете java.util.concurrent. Он может принимать одно из следующих значений:

- DAYS
- HOURS
- MINUTES
- SECONDS
- MICROSECONDS
- MILLISECONDS
- NANOSECONDS

Первые три появились в Java SE 6.

Несмотря на то что с помощью TimeUnit можно определить любое из этих значений в вызовах методов, принимающих параметр синхронизации, нет гарантии того, что система сможет работать с заданным разрешением.

Ниже следует пример, в котором используется TimeUnit. Класс CallableDemo, показанный в предыдущем разделе, был изменен, чтобы использовать вторую форму метода get(), принимающего параметр TimeUnit.

```
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 этом варианте ни один из вызовов get() не будет ожидать дольше 10 миллисекунд.

Перечисление TimeUnit определяет различные методы, выполняющие преобразование единиц.

```
long convert(long tval, TimeUnit tu)
long toMicros(long tval)
long toMillis(long tval)
long toNanos(long tval)
long toSeconds(long tval)
long toDays(long tval)
long toHours(long tval)
long toMinutes(long tval)
```

Метод `convert()` преобразует *`tval`* в определенные единицы и возвращает результат. Методы `to` выполняют указанное преобразование и возвращают результат.

`TimeUnit` также определяет следующие методы синхронизации:

```
void sleep(long delay) throws InterruptedExecution
void timedJoin(Thread thrd, long delay) throws InterruptedExecution
void timedWait(Object obj, long delay) throws InterruptedExecution
```

`sleep()` приостанавливает выполнение на определенный период времени, который задается в виде вызывающей константы перечисления. Он преобразуется в вызов `Thread.sleep()`.

`timedJoin()` является специализированной версией метода `Thread.join()`, в котором поток (параметр *`thrd`*) приостанавливается на период времени, указанный в *`delay`*. `timedWait()` является специализированной версией метода `Object.wait()`, в котором *`obj`* ожидает период времени, заданный в *`delay`*, который исчисляется в вызывающих единицах времени.

Параллельные коллекции

Как уже отмечалось, параллельный API определяет несколько коллекций, которые были разработаны для выполнения параллельных операций. К ним относятся следующие коллекции:

- `ArrayBlockingQueue`
- `ConcurrentHashMap`
- `ConcurrentLinkedQueue`
- `ConcurrentSkipListMap` (Добавлена в Java SE 6.)
- `ConcurrentSkipListSet` (Добавлена в Java SE 6.)
- `CopyOnWriteArraylist`
- `CopyOnWriteArraySet`
- `DelayQueue`
- `LinkedBlockingDeque` (Добавлена в Java SE 6.)
- `LinkedBlockingQueue`
- `PriorityBlockingQueue`
- `SynchronousQueue`

Они предлагают параллельные альтернативы связанным с ними классам, определенным в каркасе Collections Framework. Эти коллекции работают подобно другим коллекциям, за исключением того, что они поддерживают параллелизм. Программисты, знакомые с каркасом Collections Framework, не будут иметь проблем с использованием этих параллельных коллекций.

Блокировки

Пакет `java.util.concurrent.locks` предлагает поддержку *блокировок* (lock), которые являются объектами, предлагающими альтернативу использованию `synchronized` для управления доступом к общему ресурсу. Давайте разберемся с тем, как работают блокировки. Прежде чем получить доступ к общему ресурсу, запрашивается блокировка,

защищающая этот ресурс. Когда доступ к ресурсу будет завершен, блокировка снимается. Если второй поток попытается запросить блокировку в тот момент, когда она используется еще каким-нибудь потоком, первый поток будет ожидать, пока блокировка не будет снята. Благодаря этому появляется возможность избежать возникновения конфликта доступа к общему ресурсу.

Блокировки особенно полезны тогда, когда нескольким потокам нужно получить доступ к значению из общих данных. Например, приложение складского учета может иметь поток, который сначала подтверждает, что товар имеется на складе, а затем уменьшает количество доступных товаров после каждой продажи. Если будет выполняться два или более таких потока, то без синхронизации может получиться так, что один поток начнет свою транзакцию в момент выполнения транзакции другим потоком. В результате этого оба потока будут предполагать о существовании достаточного количества товара, хотя на самом деле товара будет ровно столько, сколько требуется для осуществления одной продажи. В подобных ситуациях с помощью блокировок можно организовывать синхронную работу потоков.

Каждая блокировка реализует интерфейс `Lock`. В табл. 26.2 перечислены методы, определенные интерфейсом `Lock`. В общем случае для запроса блокировки необходимо вызвать метод `lock()`. Если блокировка не будет доступна, метод `lock()` войдет в режим ожидания. Для снятия блокировки вызовите метод `unlock()`. Чтобы узнать, является ли блокировка свободной, а также, чтобы запросить ее, если она свободна, вызовите метод `tryLock()`. Метод не будет ожидать блокировку, если она не является доступной. Наоборот, он возвращает `true`, если блокировка получена, и `false` — если нет. Метод `newCondition()` возвращает объект `Condition`, связанный с блокировкой. Применение `Condition` позволяет расширить возможности управления блокировками с помощью методов `await()` и `signal()`, которые обеспечивают функциональность, подобную той, что предлагается методами `Object.wait()` и `Object.notify()`.

Таблица 26.2. Методы `Lock`

Метод	Описание
`void lock()`	Ожидание длится до тех пор, пока вызываемая блокировка не может быть получена.
`void lockInterruptibly()` `throws InterruptedException`	Ожидание длится до тех пор, пока вызываемая блокировка не может быть получена, если только не произойдет прерывание.
`Condition newCondition()`	Возвращает объект `Condition`, связанный с вызываемой блокировкой.
`boolean tryLock()`	Пытается запросить блокировку. Этот метод не входит в режим ожидания, если блокировка не является свободной. Вместо этого он возвращает `true`, если блокировка была получена, и `false`, если на данный момент блокировка используется другим потоком.
`boolean tryLock(long` *`wait`*`,` `TimeUnit` *`tu`*`)` `throws InterruptedException`	Пытается получить блокировку. Если блокировка недоступна, то метод будет ожидать столько времени, сколько указано в параметре *`wait`*, единицы которого определены в *`tu`*. Он возвращает `true`, если блокировка была получена, и `false`, если блокировка не была получена в течение заданного периода.
`void unlock()`	Снимает блокировку.

Пакет java.util.concurrent.locks имеет реализацию Lock и ReentrantLock. ReentrantLock реализует *реентерабельную блокировку*, представляющую собой блокировку, в которую поток, удерживающий на данный момент эту блокировку, может входить повторно. (Естественно, если поток входит в блокировку повторно, все вызовы метода lock() должны быть смещены на равное количество вызовов unlock().) В противном случае поток, пытающийся запросить блокировку, перейдет в режим ожидания до тех пор, пока она не будет освобождена.

В следующей программе демонстрируется пример использования блокировок. В ней создается два потока, которые обращаются к общему ресурсу Shared.count. Прежде чем поток сможет обратиться к Shared.count, он должен получить блокировку. После получения блокировки Shared.count увеличивается, после чего, не снимая блокировки, поток входит в режим простоя. Вследствие этого другой поток будет пытаться получить блокировку. Однако поскольку блокировка все еще удерживается первым потоком, другой поток будет ожидать до тех пор, пока первый поток не выйдет из режима простоя и не снимет блокировку. Результаты выполнения показывают, что доступ к 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 выполнения, увеличивающий значение счета.
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 " + name);

    try {
      // Сначала блокируется счетчик.
      System.out.println(name + " ожидает блокирования счетчика.");
      lock.lock();
      System.out.println(name + " блокирует счетчик.");

      Shared.count++;
      System.out.println(name + ": " + Shared.count);
```

```
        // Теперь, если это возможно, разрешается контекстное переключение.
        System.out.println(name + " простаивает.");
        Thread.sleep(1000);
      } catch (InterruptedException exc) {
        System.out.println(exc);
      } finally {
        // Снятие блокировки
        System.out.println(name + " разблокирует счетчик.");
        lock.unlock();
      }
    }
  }
```

Ниже показаны результаты выполнения. (У вас порядок выполнения потоков может быть другим.)

```
Запуск A
A ожидает блокирования счетчика.
A блокирует счетчик.
A: 1
A простаивает.
Запуск B
B ожидает блокирования счетчика.
A разблокирует счетчик.
B блокирует счетчик.
B: 2
B простаивает.
B разблокирует счетчик.
```

Пакет `java.util.concurrent.locks` также определяет интерфейс `ReadWriteLock`. Этот интерфейс определяет реентерабельную блокировку, которая поддерживает отдельные блокировки для доступа к чтению и записи. Это позволит предоставлять читателям ресурса несколько блокировок до его записи. `ReentrantReadWriteLock` предлагает реализацию `ReadWriteLock`.

Атомарные операции

Пакет `java.util.concurrent.atomic` предлагает альтернативный вариант другим функциональным возможностям синхронизации при чтении или записи значения некоторых типов переменных. В этом пакете доступны методы, которые получают, задают или сравнивают значение переменной во время одной непрерывной (то есть атомарной) операции. А это значит, что теперь не будет нужна ни блокировка, ни любой другой механизм синхронизации.

Атомарные операции выполняются с помощью классов `AtomicInteger` и `AtomicLomg` и методов `get()`, `set()`, `compareAndSet()`, `decrementAndGet()` и `getAndSet()`, которые реализуют действия, соответствующие их именам.

Ниже показан пример, показывающий, как можно синхронизировать доступ к общему ресурсу с помощью `AtomicInteger`.

```
// Простой пример 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 выполнения, при котором увеличивается значение счета.
class AtomThread implements Runnable {
  String name;
  AtomThread(String n) {
    name = n;
    new Thread(this).start();
  }
  public void run() {
    System.out.println("Запуск " + name);
    for(int i=1; i <= 3; i++)
      System.out.println(name + " получено: " +
            Shared.ai.getAndSet(i));
  }
}
```

В этой программе посредством `Shared` создается статический `AtomicInteger` по имени `ai`. Затем создаются три потока типа `AtomThread`. В методе `run()` `Shared.ai` изменяется вызовом метода `getAndSet()`. Этот метод возвращает предыдущее значение и устанавливает то значение, которое было передано в качестве параметра. Благодаря `AtomicInteger` исключается вероятность того, что два потока будут одновременно осуществлять запись в `ai`.

В общем случае атомарные операции предлагают удобную (и, возможно, более эффективную) альтернативу другим механизмам синхронизации при работе с одной переменной.

Параллельные утилиты в сравнении с традиционным подходом в Java

Если принять во внимание ту мощность и гибкость, которую предлагают новые параллельные утилиты, то возникает следующий резонный вопрос: могут ли они служить заменой традиционному для Java подходу к реализации многопоточности и синхронизации? Однозначно нет! Первоначальная поддержка многопоточности и встроенные функциональные возможности синхронизации по-прежнему следует реализовывать во множестве Java-программ, аплетов и сервлетов. Например, `synchronized`, `wait()` и `notify()` предлагают элегантные решения широкого круга задач. Однако если вам понадобится расширить возможности управления, то для этих целей можно воспользоваться параллельными утилитами.

27
ГЛАВА

NIO, регулярные выражения и другие пакеты

Когда язык Java впервые увидел свет, он включал восемь пакетов, называемых *API ядра*. С каждым последующим выпуском API пополнялся новыми пакетами. На сегодняшний день Java API содержит большое количество пакетов. Многие из них являются специализированными и выходящими за рамки контекста этой книги. Тем не менее, пять пакетов мы все-таки рассмотрим: `java.nio`, `java.util.regex`, `java.lang.reflect`, `java.rmi` и `java.text`. Они поддерживают ввод-вывод на основе NIO, обработку регулярных выражений, рефлексию, удаленный вызов методов (Remote Method Invocation — RMI) и обработку текстов.

Интерфейс *NIO* API предлагает отличный способ обработки некоторых типов операций ввода-вывода. Пакет *регулярных выражений* позволит выполнять сложные операции сопоставления с шаблонами. В этой главе будет дан углубленный анализ обоих пакетов и предложены многочисленные их примеры. *Рефлексией* называется способность программного обеспечения к самоанализу.

Рефлексия является неотъемлемой частью технологии Java Beans, которая рассматривается в главе 28. *Удаленный вызов методов* позволяет создавать Java-приложения, которые будут распределяться среди нескольких компьютеров. В этой главе будет предложен один простой клиент-серверный пример, в котором используется RMI. Возможности *форматирования текста*, предлагаемые пакетом `java.text`, имеют самое широкое применение. В этой главе будут рассмотрены примеры форматирования строк даты и времени.

Пакеты API ядра

В табл. 27.1 приведен список всех пакетов API ядра с указанием их назначения.

Таблица 27.1. Пакеты Core API

Пакет	Основное назначение
java.applet	Поддерживает процесс создания аплетов.
java.awt	Предлагает особенности для графических пользовательских интерфейсов.
java.awt.color	Поддерживает цветовые пространства и профили.
java.awt.datatransfer	Передает данные в системный буфер обмена и обратно.
java.awt.dnd	Поддерживает операции перетаскивания.
java.awt.event	Обрабатывает события.
java.awt.font	Представляет различные типы шрифтов.
java.awt.geom	Позволяет работать с геометрическими формами.
java.awt.im	Позволяет вводить японские, китайские и корейские символы в компоненты редактирования текста.
java.awt.im.spi	Поддерживает альтернативные устройства ввода.
java.awt.image	Обрабатывает изображения.
java.awt.image.renderable	Поддерживает изображения с независимой визуализацией.
java.awt.print	Поддерживает общие свойства печати.
java.beans	Позволяет создавать компоненты программного обеспечения.
java.beans.beancontext	Обеспечивает среду выполнения для Bean-компонентов.
java.io	Вводит и выводит данные.
java.lang	Обеспечивает базовую функциональность.
java.lang.annotation	Поддерживает аннотации (метаданные).
java.lang.instrument	Поддерживает инструментальные средства для программ.
java.lang.management	Поддерживает управление средой выполнения.
java.lang.ref	Позволяет взаимодействовать со сборщиком мусора.
java.lang.reflect	Анализирует код во время выполнения.
java.math	Обрабатывает большие целые и десятичные числа.
java.net	Поддерживает работу в сети.
java.nio	Пакет верхнего уровня для классов NIO. Инкапсулирует буферы.
java.nio.channels	Инкапсулирует каналы, используемые в системе NIO.
java.nio.channels.spi	Поддерживает поставщиков услуг для каналов.
java.nio.charset	Инкапсулирует наборы символов.

Окончание табл. 27.1

Пакет	Основное назначение
java.nio.charset.spi	Поддерживает поставщиков услуг для наборов символов.
java.rmi	Обеспечивает удаленный вызов методов.
java.rmi.activation	Активизирует постоянные объекты.
java.rmi.dgc	Управляет распределенным сбором мусора.
java.rmi.registry	Отображает имена как ссылки на удаленные объекты.
java.rmi.server	Поддерживает удаленный вызов методов.
java.security	Обрабатывает сертификаты, ключи, дайджесты, подписи и другие функции, связанные с безопасностью.
java.security.acl	Поддерживает списками управления доступом.
java.security.cert	Выполняет синтаксический анализ и управление сертификатами.
java.security.interfaces	Определяет интерфейсы для ключей DSA (Digital Signature Algorithm — алгоритм цифровой подписи).
java.security.spec	Определяет параметры ключей и алгоритмов.
java.sql	Взаимодействует с базой данных SQL (Structured Query Language — язык структурированных запросов).
java.text	Форматирует, производит поиск и манипулирует текстом.
java.text.spl	Поддерживает поставщиков услуг для классов форматирования текста в Java.text. (Добавлен в Java SE 6.)
java.util	Содержит общие утилиты.
java.util.concurrent	Поддерживает параллельные утилиты.
java.util.concurrent.atomic	Поддерживает атомарные (то есть неделимые) операции в отношении переменных без использования блокировок.
java.util.concurrent.locks	Поддерживает синхронизационные блокировки.
java.util.jar	Создает и считывает JAR-файлы.
java.util.logging	Поддерживает регистрацию информации, связанной с выполнением программы.
java.util.prefs	Инкапсулирует информацию, связанную с предпочтениями пользователя.
java.util.regex	Поддерживает обработку регулярных выражений.
java.util.spl	Поддерживает поставщиков услуг классов-утилит в java.util. (Добавлен в Java SE 6.)
java.util.zip	Считывает и записывает упакованные и распакованные ZIP-файлы.

NIO

Одним из наиболее примечательных пакетов в Java, появившихся в последнее время, является *NIO* (New I/O). Этот пакет интересен тем, что для выполнения операций ввода-вывода он поддерживает каналы. Классы NIO содержатся в пяти пакетах, описанных в табл. 27.2.

Таблица 27.2. Пакеты, поддерживающие классы NIO

Пакет	Назначение
`java.nio`	Пакет верхнего уровня в системе NIO. Инкапсулирует различные типы буферов, содержащих данные, с которыми работает система NIO.
`java.nio.channels`	Поддерживает каналы, открывающие соединения ввода-вывода.
`java.nio.channels.spi`	Поддерживает поставщиков услуг для каналов.
`java.nio.charset`	Инкапсулирует наборы символов. Также поддерживает работу кодировщиков и декодеров, преобразующих символы в байты и байты в символы соответственно.
`java.nio.charset.spi`	Поддерживает поставщиков услуг для наборов символов.

Прежде чем приступить к рассмотрению NIO, следует понять, что эта подсистема не предназначена для замены классов ввода-вывода, хранящихся в пакете `java.io`, о которых шла речь в главе 19. Наоборот, классы NIO дополняют стандартную систему ввода-вывода, предлагая альтернативный принцип, применение которого в некоторых случаях может быть предпочтительным.

Основы NIO

Система NIO построена на двух фундаментальных элементах: буферах и каналах. *Буфер* хранит данные, а *канал* представляет открытое соединение с устройством ввода-вывода — с файлом или сокетом. В общем случае для использования системы NIO необходимо получить канал для устройства ввода-вывода и буфер для хранения данных. После этого вы будете работать с буфером, вводя или выводя данные по мере необходимости. В следующих разделах мы рассмотрим подробно буферы и каналы.

Буферы

Буферы определены в пакете `java.nio`. Все буферы являются подклассами класса `Buffer`, который определяет основные функциональные особенности, характерные для каждого буфера: текущая позиция, лимит и вместимость. *Текущая позиция* представляет собой индекс в буфере, с которого в следующий раз начнется операция чтения или записи данных. Текущая позиция перемещается после выполнения большинства операций чтения или записи. *Лимит* представляет собой индекс, обозначающий конец буфера. *Вместимость* определяет количество элементов, которые может хранить буфер. Класс `Buffer` также поддерживает метку и сброс. Класс `Buffer` определяет несколько методов, перечисленных в табл. 27.3.

Таблица 27.3. Методы, определенные в классе `Buffer`

Метод	Описание
`abstract Object array()`	Если вызывающий буфер поддерживается массивом, возвращается ссылка на массив. В противном случае генерируется исключение `UnsupportedOperationException`. Если массив доступен только для чтения, генерируется исключение `ReadOnlyBufferException`. (Добавлен в Java SE 6.)
`abstract int arrayOffset()`	Если вызывающий буфер поддерживается массивом, возвращается индекс первого элемента. В противном случае генерируется исключение `UnsupportedOperationException`. Если массив доступен только для чтения, генерируется исключение `ReadOnlyBufferException`. (Добавлен в Java SE 6.)
`final int capacity()`	Возвращает количество элементов, которые может хранить вызывающий буфер.
`final Buffer clear()`	Очищает вызывающий буфер и возвращает ссылку на буфер.
`final Buffer flip()`	Устанавливает лимит вызывающего буфера по текущей позиции и сбрасывает текущую позицию до нуля. Возвращает ссылку на буфер.
`abstract boolean has Array()`	Возвращает `true`, если вызывающий буфер поддерживается массивом, доступным для чтения и записи. В противном случае возвращает `false`. (Добавлен в Java SE 6.)
`final boolean hasRemaining()`	Возвращает `true`, если существуют элементы, оставшиеся в вызывающем буфере. В противном случае возвращает `false`.
`abstract isDirect()`	Возвращает `true`, если вызывающий буфер прямой — другими словами, с ним часто можно работать напрямую, а не через его копию. В противном случае возвращает `false`. (Добавлен в Java SE 6.)
`abstract boolean isReadOnly()`	Возвращает `true`, если вызывающий буфер является буфером только для чтения. В противном случае возвращает `false`.
`final int limit()`	Возвращает лимит вызывающего буфера.
`final Buffer limit(int n)`	Устанавливает лимит вызывающего буфера в `n`. 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)`	Устанавливает текущую позицию буфера в `n`. Возвращает ссылку на буфер.
`final Buffer reset()`	Сбрасывает текущую позицию вызывающего буфера в предварительно установленную метку. Возвращает ссылку на буфер.
`final Buffer rewind()`	Устанавливает позицию вызывающего буфера в ноль. Возвращает ссылку на буфер.

На основе класса `Buffer` можно получить следующие специфические классы буферов, тип хранимых данных которых можно определить по их именам:

- `ByteBuffer`
- `CharBuffer`
- `DoubleBuffer`
- `FloatBuffer`
- `IntBuffer`
- `LongBuffer`
- `MappedByteBuffer`
- `ShortBuffer`

Класс `MappedByteBuffer` является подклассом класса `ByteBuffer`, используемого для отображения файла в виде буфера.

Каждый буфер поддерживает различные методы `get()` и `put()`, которые позволяют получать данные из буфера или вносить их в него. Например, в табл. 27.4 представлены методы `get()` и `put()`, определяемые классом `ByteBuffer`. (Другие классы буферов имеют похожие методы.) Все классы буферов также поддерживают методы, выполняющие различные операции. Например, с помощью метода `allocate()` можно вручную распределить память под буфер. С помощью метода `wrap()` можно организовать массив внутри буфера. С помощью метода `slice()` можно создать подпоследовательность буфера.

Таблица 27.4. Методы `get()` и `put()`, определенные в классе `ByteBuffer`

Метод	Описание
`abstract byte get()`	Возвращает байт в текущей позиции.
`ByteBuffer get(byte vals[] )`	Копирует вызывающий буфер в массив, на который указывает параметр *vals*. Возвращает ссылку на буфер.
`ByteBuffer get(byte vals[], int start, int num)`	Копирует *num* элементов из вызывающего буфера в массив, на который указывает параметр *vals*, начиная с индекса, заданного в параметре *start*. Возвращает ссылку на буфер. Если в буфере не остались элементы, количество которых определено в соответствующем параметре, генерируется исключение `BufferUnderflowException`.
`abstract byte get(int idx)`	Возвращает байт по индексу, заданному в параметре *idx*, из вызывающего буфера.
`abstract ByteBuffer put(byte b)`	Копирует *b* в текущую позицию вызывающего буфера. Возвращает ссылку на буфер.
`final ByteBuffer put(byte vals[] )`	Копирует все элементы *vals* в вызывающий буфер, начиная с текущей позиции. Возвращает ссылку на буфер.
`ByteBuffer put(byte vals[], int start, int num)`	Копирует *num* элементов из *vals*, начиная с позиции *start*, в вызывающий буфер. Возвращает ссылку на буфер. Если буфер не может хранить все элементы, генерируется исключение `BufferOverflowException`.
`ByteBuffer put(ByteBuffer bb)`	Копирует элементы из *bb* в вызывающий буфер, начиная с текущей позиции. Если буфер не может хранить все элементы, генерируется исключение `BufferOverflowException`. Возвращает ссылку на буфер.
`abstract ByteBuffer put(int idx, byte b)`	Копирует *b* в позицию вызывающего буфера, заданную d параметрt *idx*. Возвращает ссылку на буфер.

Каналы

Каналы определены в пакете `java.io.channels`. Канал представляет открытое соединение с источником или приемником ввода-вывода. Получение канала осуществляется с помощью метода `getChannel()` объекта, поддерживающего каналы. Метод `getChannel()` поддерживается следующими классами ввода-вывода:

- `DatagramSocket`
- `FileInputStream`
- `FileOutputStream`
- `RandomAccessFile`
- `ServerSocket`
- `Socket`

Таким образом для получения канала потребуется сначала получить объект одного из этих классов, а затем вызвать метод `getChannel()` для данного объекта.

Специфический тип возвращаемого канала зависит от типа объекта, для которого вызывается метод `getChannel()`. Например, при вызове `FileInputStream`, `FileOutputStream` или `RandomAccessFile` метод `getChannel()` возвращает канал типа `FileChannel`. При вызове `Socket` метод `getChannel()` возвращает `SocketChannel`.

Каналы вроде `FileChannel` и `SocketChannel` поддерживают различные методы `read()` и `write()`, которые позволяют выполнять операции ввода-вывода через канал. Например, в табл. 27.5 показано несколько методов `read()` и `write()`, определенных для `FileChannel`. Каждый из них может сгенерировать исключение `IOException`.

Таблица 27.5. Методы `read()` и `write()`, определенные в классе `FileChannel`

Метод	Описание
`abstract int read(ByteBuffer bb)`	Считывает байты из вызывающего канала в *bb* до тех пор, пока буфер не будет заполнен или пока не закончатся входные данные. Возвращает количество прочитанных байтов.
`abstract int read(ByteBuffer bb, long start)`	Начиная с позиции, определяемой посредством параметра *start*, считывает байты из вызывающего канала в *bb* до тех пор, пока буфер не будет заполнен или пока не закончатся входные данные. Текущая позиция не изменяется. Возвращает количество прочитанных байтов или -1, если начальная позиция (параметр *start*) окажется за границами файла.
`abstract int write(ByteBuffer bb)`	Записывает содержимое байтового буфера в вызывающий канал, начиная с текущей позиции. Возвращает количество записанных байтов.
`abstract int write(ByteBuffer bb, long start)`	Начиная с позиции файла, определяемой посредством параметра *start*, записывает содержимое байтового буфера в вызывающий канал. Текущая позиция не изменяется. Возвращает количество записанных байтов.

Все каналы поддерживают традиционные методы, которые предоставляют доступ к каналу и дают возможность управлять ним. Например, `FileChannel` поддерживает методы

для получения или настройки текущей позиции, передачи информации между файловыми каналами, получения текущего размера канала и блокировки канала. `FileChannel` также предлагает метод `map()`, с помощью которого можно отобразить файл в виде буфера.

Наборы символов и селекторы

Двумя другими категориями, используемыми системой NIO, являются наборы символов и селекторы. *Набор символов* определяет способ отображения байтов в виде символов. С помощью *кодировщика* можно зашифровать последовательность символов в виде байтов. Процесс расшифровки производится с помощью *декодера*. Наборы символов, кодировщики и декодеры поддерживаются классами, определенными в пакете `java.nio.charset`.

Поскольку кодировщики и декодеры предлагаются по умолчанию, работать с наборами символов явным образом вам придется очень редко.

Селектор поддерживает возможность многоканального ввода-вывода на основе ключей без применения блокировки. Другими словами, с помощью селекторов можно выполнять операции ввода-вывода посредством нескольких каналов. Селекторы поддерживаются классами, определенными в пакете `java.nio.channels`. Селекторы чаще всего применяются в каналах на основе сокетов.

В этой главе мы не будем использовать наборы символов или селекторы, однако их применение может оказаться очень полезным в ряде приложений.

Использование системы NIO

В связи с тем, что наиболее распространенным устройством ввода-вывода является дисковый файл, в оставшейся части раздела будет показано, как осуществляется доступ к дисковому файлу с помощью системы NIO. Поскольку все операции с файловыми каналами построены на основе байтов, мы будем работать с буферами, имеющими тип `ByteBuffer`.

Чтение файла

Прочитать данные из файла с помощью NIO можно несколькими способами. Мы рассмотрим два из них. В первом способе для чтения файла сначала вручную назначается буфер, а затем явным образом производится операция чтения. Во втором способе используется отображенный файл, который позволяет автоматизировать этот процесс.

Чтобы прочитать файл с помощью канала и вручную назначить буфер, необходимо выполнить следующую процедуру. Сначала откройте файл для ввода данных с помощью `FileInputStream`. Затем с помощью метода `getChannel()` получите канал для этого файла. Этот метод имеет следующую общую форму:

```
FileChannel getChannel()
```

Метод возвращает объект `FileChannel`, который инкапсулирует канал для операций с файлами. После того как канал файла будет открыт, с помощью метода `size()` получите размер файла, как показано далее:

```
long size() throws IOException
```

Этот метод возвращает текущий размер канала, в байтах, который отражает данный файл. Далее вызовите метод `allocate()` для распределения памяти под буфер, размер которого позволит сохранить содержимое файла. Поскольку каналы файлов работают

только с байтовыми буферами, вы будете использовать метод `allocate()`, определяемый классом `ByteBuffer`. Он имеет следующую общую форму:

```
static ByteBuffer allocate(int cap)
```

Здесь параметр *cap* определяет вместимость буфера. Метод возвращает ссылку на буфер. После создания буфера вызовите метод `read()` для канала, передавая ссылку на буфер.

В следующей программе показано, как производится чтение текстового файла `test.txt` посредством канала с использованием явных операций ввода.

```
// Использование NIO для чтения текстового файла.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class ExplicitChannelRead {
  public static void main(String args[])
    FileInputStream fIn;
    FileChannel fChan;
    long fSize;
    ByteBuffer mBuf;
    try
      // Сначала открываем файл для ввода данных.
      fIn = new FileInputStream("test.txt");

      // После этого получаем канал для этого файла.
      fChan = fIn.getChannel();

      // Теперь узнаем размер файла.
      fSize = fChan.size();

      // Выделяем буфер требуемого размера.
      mBuf = ByteBuffer.allocate((int)fSize);

      // Считываем файл в буфер.
      fChan.read(mBuf);

      // Производим "перемотку" буфера для того, чтобы его можно было читать.
      mBuf.rewind();

      // Считываем байты из буфера.
      for(int i=0; i < fSize; i++)
        System.out.print((char)mBuf.get());

      System.out.println();

      fChan.close(); // закрываем канал
      fIn.close(); // закрываем файл
    } catch (IOException exc) {
      System.out.println(exc);
      System.exit(1);
    }
  }
}
```

Теперь давайте разберемся с тем, как работает эта программа. Вначале с помощью конструктора `FileInputStream` открывается файл, а `fIn` получает ссылку на этот объект. Затем с помощью метода `getChannel()` производится получение канала, связанного с этим файлом, а с помощью метода `size()` определяется размер файла. После этого программа вызывает метод `allocate()` класса `ByteBuffer` для назначения буфера, ко-

торый будет хранить содержимое файла при его чтении. В данном случае используется байтовый буфер, поскольку FileChannel работает именно с байтами. Ссылка на этот буфер хранится в mBuf. Затем с помощью метода read() содержимое файла считывается в mBuf. После этого с помощью метода rewind() производится "перемотка" буфера. Перемотка необходима по той причине, что после выполнения метода read() текущая позиция находится в конце буфера. Ее необходимо переустановить на начало буфера, чтобы во время вызова метода get() можно было считывать байты в mBuf. Так как mBuf является байтовым буфером, то метод get() возвращает байты. Они приводятся к типу char, чтобы файл можно было отобразить в виде текста. (Как вариант, можно создать буфер, который будет зашифровывать байты в символы, и затем читать этот буфер.) Программа завершается закрытием канала и файла.

Другой способ чтения файла, реализовать который во многих случаях бывает гораздо легче, заключается в отображении файла в виде буфера. Преимущество такого подхода заключается в том, что буфер автоматически будет хранить содержимое файла. При этом не требуется производить явную операцию чтения. Чтобы отобразить файл в виде буфера и прочитать его содержимое, следуйте основной процедуре. Сначала нужно открыть файл с помощью FileInputStream. Затем нужно получите канал для объекта файла с помощью метода getChannel(). После этого нужно отобразите канал в виде буфера путем вызова метода map() для объекта FileChannel. Метод map() показан далее:

```
MappedByteBuffer map(FileChannel.MapMode how,
                     long pos, long size) throws IOException
```

В результате выполнения метода map() данные, которые хранит файл, будут отображены в буфере в памяти. Значение параметра *how* определяет, какой тип операций будет разрешен. Этот параметр может принимать одно из следующих значений:

- MapMode.READ_ONLY
- MapMode.READ_WRITE
- MapMode.PRIVATE

Для чтения файла используйте MapMode.READ_ONLY. Чтобы прочитать файл и записать в него данные, применяйте MapMode.READ_WRITE. Если параметру *how* присвоить значение MapMode.PRIVATE, то будет произведена приватная копия файла, а изменения в буфере на данный файл не повлияют. Позиция внутри файла, с которой начнется отображение, определяется посредством параметра *pos*, а количество байтов для отображения — с помощью параметра *size*. Ссылка на этот буфер возвращается в виде MappedByteBuffer, который является подклассом ByteBuffer. После того как файл будет отображен в виде буфера, его можно будет оттуда прочитать.

Следующая программа является модификацией первого примера; теперь используется отображенный файл.

```
// Использование отображенного файла для чтения текстового файла.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MappedChannelRead {
  public static void main(String args[]) {
    FileInputStream fIn;
    FileChannel fChan;
    long fSize;
    MappedByteBuffer mBuf;
```

```
    try {
      // Сначала открываем файл для ввода данных.
      fIn = new FileInputStream("test.txt");

      // Затем получаем для этого файла канал.
      fChan = fIn.getChannel();

      // Определяем размер файла.
      fSize = fChan.size();

      // Теперь отображаем файл в виде буфера.
      mBuf = fChan.map(FileChannel.MapMode.READ_ONLY, 0, fSize);

      // Считываем байты из буфера.
      for(int i=0; i < fSize; i++)
      System.out.print((char)mBuf.get());

      fChan.close(); // закрываем канал
      fIn.close();   // закрываем файл
    } catch (IOException exc) {
      System.out.println(exc);
      System.exit(1);
    }
  }
}
```

Как и прежде, файл открывается с помощью конструктора `FileInputStream`, а `fIn` получает ссылку на этот объект. С помощью метода `getChannel()` осуществляется получение канала, связанного с этим файлом, а затем определяется размер файла. После этого с помощью метода `map()` весь файл отображается в памяти, а ссылка на буфер сохраняется в `mBuf`. Байты из `mBuf` считываются с помощью метода `get()`.

Запись в файл

Существует несколько способов записи данных в файл через канал. Мы рассмотрим два таких способа. Во-первых, данные в выходной файл можно записать посредством канала, используя явные операции записи. Во-вторых, если файл открыт для операций чтения-записи, то можно отобразить файл на буфер, и затем производить запись в этот буфер. Изменения в буфере автоматически повлияют на изменения в файле. Оба эти способа описаны далее.

Чтобы записать файл через канал с помощью явных вызовов метода `write()`, необходимо выполнить следующие действия. Сначала потребуется открыть файл для вывода данных. Затем нужно выделить байтовый буфер, поместить в него данные, которые вы хотите записать, и вызвать метод `write()` для канала. В следующей программе демонстрируется эта процедура. В ней записывается алфавит в файл с именем `test.txt`.

```
// Запись данных в файл с использованием NIO.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class ExplicitChannelWrite {
  public static void main(String args[]) {
    FileOutputStream fOut;
    FileChannel fChan;
    ByteBuffer mBuf;

    try {
      fOut = new FileOutputStream("test.txt");
```

```
      // Получаем канал для выходного файла.
      fChan = fOut.getChannel();

      // Создаем буфер.
      mBuf = ByteBuffer.allocateDirect(26);

      // Записываем в него несколько байтов.
      for(int i=0; i<26; i++)
        mBuf.put((byte)('A' + i));

      // Перематываем буфер, для того чтобы в него можно было произвести запись.
      mBuf.rewind();

      // Записываем буфер в выходной файл.
      fChan.write(mBuf);

      // Закрываем канал и файл.
      fChan.close();
      fOut.close();
    } catch (IOException exc) {
      System.out.println(exc);
      System.exit(1);
    }
  }
}
```

Вызов метода `rewind()` для `mBuf` необходим для того, чтобы сбросить текущую позицию до нуля после того, как данные будут записаны в `mBuf`. Не забывайте о том, что каждый вызов метода `put()` перемещает текущую позицию. Следовательно, прежде чем вызывать метод `write()`, необходимо сбросить текущую позицию на начало буфера. Если этого не сделать, метод `write()` не найдет в буфере никаких данных.

Чтобы записать данные в файл с помощью отображенного файла, выполните следующие действия. Откройте файл для операций чтения-записи. Затем отобразите этот файл на буфер с помощью метода `map()`. После этого произведите запись в буфер. Поскольку буфер отображен на файл, то любые изменения в этом буфере автоматически скажутся на этом файле. Таким образом, явные операции записи в канал не требуются. Далее представлена предыдущая программа, переработанная таким образом, что в ней используется отображенный файл. Обратите внимание, что файл открывается как `RandomAccessFile`. Это необходимо для того, чтобы разрешить чтение и запись данных в файл.

```
// Запись в отображенный файл.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MappedChannelWrite {
  public static void main(String args[]) {
    RandomAccessFile fOut;
    FileChannel fChan;
    ByteBuffer mBuf;

    try {
      fOut = new RandomAccessFile("test.txt", "rw");

      // Получение канала для этого файла.
      fChan = fOut.getChannel();

      // Отображение файла в виде буфера.
      mBuf = fChan.map(FileChannel.MapMode.READ_WRITE, 0, 26);
```

```
      // Запись нескольких байтов в буфер.
      for(int i=0; i<26; i++)
      mBuf.put((byte)('A' + i));

      // закрытие канала и файла.
      fChan.close();
      fOut.close();
    } catch (IOException exc) {
      System.out.println(exc);
      System.exit(1);
    }
  }
}
```

Как видите, для самого канала нет явных операций записи. Поскольку `mBuf` отображается на файл, то изменения в `mBuf` автоматически отразятся в данном файле.

Копирование файла с использованием NIO

Система NIO упрощает некоторые разновидности операций с файлами. Например, в следующей программе осуществляется копирование файла. Для этого открывается входной канал для исходного файла и выходной канал для искомого файла. Затем выполняется запись в отображенный на выходной файл входной буфер в рамках одной операции. Вы можете сравнить эту программу копирования файла с такой же программой из главы 13. При сравнении будет видно, что часть программы, которая отвечает за копирование файла, стала заметно короче.

```
// Копирование файла с использованием NIO.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class NIOCopy {

  public static void main(String args[]) {
    FileInputStream fIn;
    FileOutputStream fOut;
    FileChannel fIChan, fOChan;
    long fSize;
    MappedByteBuffer mBuf;

    try {
      fIn = new FileInputStream(args[0]);
      fOut = new FileOutputStream(args[1]);

      // Получение каналов для входного и выходного файлов.
      fIChan = fIn.getChannel();
      fOChan = fOut.getChannel();

      // Получение размера файла.
      fSize = fIChan.size();

      // Отображение входного файла в виде буфера.
      mBuf = fIChan.map(FileChannel.MapMode.READ_ONLY, 0, fSize);

      // Запись буфера в выходной файл.
      fOChan.write(mBuf); // копирует файл

      // Закрываем каналы и файлы.
      fIChan.close();
      fIn.close();
```

```
        fOChan.close();
        fOut.close();
      } catch (IOException exc) {
        System.out.println(exc);
        System.exit(1);
      } catch (ArrayIndexOutOfBoundsException exc) {
        System.out.println("Usage: Copy from to");
        System.exit(1);
      }
    }
  }
```

Поскольку входной файл отображается на `mBuf`, он содержит весь исходный файл. Таким образом, при вызове метода `write()` в искомый файл копируется весь `mBuf`. Это, естественно, означает, что искомый файл является идентичной копией исходного файла.

Будущие перспективы NIO

API-интерфейсы NIO предлагают новый интересный способ обработки некоторых типов операций с файлами. Что, в свою очередь, порождает естественный вопрос: есть ли у NIO будущее в области обработки ввода-вывода? Конечно, каналы и буферы позволяют четко представить работу операций ввода-вывода. Но в то же время они добавляют еще один уровень абстракции. Более того, традиционный подход, основанный на работе с потоками, хорошо понятен и широко используется. Как было сказано в самом начале, выполнение операций ввода-вывода с использованием каналов в настоящее время рассматривается как дополнение, а не замена стандартных механизмов ввода-вывода, определенных в пакете `java.io`. В этой роли принцип применения каналов и буферов, задействованный в API-интерфейсах NIO, подходит как нельзя кстати. Станет ли когда-нибудь новый принцип достойной заменой традиционного подхода — покажет время.

Обработка регулярных выражений

Пакет `java.util.regex` поддерживает обработку регулярных выражений. Используемый здесь термин *регулярное выражение* относится к строке символов, описывающей последовательность символов. Это общее описание, называемое *шаблоном*, впоследствии может использоваться для поиска совпадений в других последовательностях символов. Регулярные выражения могут определять групповые символы, наборы символов и разнообразные квантификаторы. Таким образом, вы можете задать регулярное выражение, представляющее общую форму, которая может совпадать с несколькими различными специфическими последовательностями символов.

Существуют два класса, поддерживающие обработку регулярных выражений: `Pattern` и `Matcher`. Эти классы работают вместе. Класс `Pattern` применяется для задания регулярного выражения. Сопоставление шаблона с последовательностью символов реализуется с помощью класса `Matcher`.

Класс `Pattern`

В классе `Pattern` нет определений конструкторов. Наоборот, шаблон формируется с помощью вызова фабричного метода `compile()`. Одна из форм этого метода показана далее:

```
static Pattern compile(String pattern)
```

Здесь *pattern* представляет регулярное выражение, которое вы хотите использовать. Метод compile() преобразует строку *pattern* в шаблон, который можно использовать для сопоставления, осуществляемого с помощью класса Matcher. Этот метод возвращает объект Pattern, содержащий данный шаблон.

После того как вы создадите объект Pattern, вы будете использовать его для создания Matcher. Для этого вызывается фабричный метод matcher(), определяемый классом Pattern. Он показан далее:

```
Matcher matcher(CharSequence str)
```

Здесь *str* — это последовательность символов, которая будет сопоставляться с шаблоном. Она называется *входной последовательностью*. CharSequence — это интерфейс, который определяет набор символов только для чтения. Помимо всего прочего, он реализуется классом String. Таким образом, вы передаете строку методу matcher().

Класс Matcher

Класс Matcher не имеет конструкторов. Наоборот, вы создаете Matcher посредством вызова фабричного метода matcher(), определяемого в классе Pattern, о чем мы уже говорили. После того как вы создадите Matcher, вы будете использовать его методы для выполнения различных операций по сопоставлению с шаблонами.

Самым простым методом сопоставления с шаблоном является matches(), который просто определяет, совпадает ли последовательность символов с шаблоном. Этот метод показан ниже:

```
boolean matches()
```

Он возвращает true, если последовательность и шаблон совпадают; в противном случае он возвращает false. Следует иметь в виду, что с шаблоном должна совпадать не вся последовательность, а только ее часть (подпоследовательность).

Чтобы определить, совпадает ли подпоследовательность с входящей последовательностью, используется метод find(). Ниже показана одна из его версий:

```
boolean find()
```

Этот метод возвращает true, если имеет место совпадение; в противном случае он возвращает false. Метод можно вызывать неоднократно, благодаря чему можно будет находить все совпадающие подпоследовательности. При каждом вызове метода find() сравнение начинается с того места, где было закончено предыдущее.

Строку, содержащую последнюю совпавшую последовательность, можно получить с помощью метода group(). Одна из его форм показана ниже:

```
String group()
```

Метод возвращает совпавшую строку. Если совпадение не было обнаружено, генерируется исключение IllegalStateException.

С помощью метода start() можно получить индекс внутри входной последовательности текущего совпадения. Индекс, следующий за окончанием текущего совпадения, можно получить с помощью метода end(). Эти методы показаны ниже:

```
int start()
int end()
```

Каждый из них в случае отсутствия несовпадения генерирует исключение IllegalStateException.

Каждую совпавшую последовательность можно заменить другой последовательностью, если вызвать метод `replaceAll()`, который показан ниже:

```
String replaceAll(String newStr)
```

Здесь параметр *`newStr`* определяет новую последовательность символов, которая заменит последовательности, совпавшие с шаблоном. Обновленная входная последовательность будет возвращена в качестве строки.

Синтаксическая структура регулярного выражения

Прежде чем продемонстрировать работу классов `Pattern` и `Matcher`, необходимо объяснить, как формируется регулярное выражение. Хотя ни одно из правил не является сложным, их существует большое количество, и полностью описать правила в этой книге невозможно. Однако здесь вы найдете несколько наиболее распространенных конструкций.

В общем случае регулярное выражение состоит из обычных символов, классов символов (наборов символов), групповых символов и квантификаторов. Обычный символ сопоставляется по принципу "как есть". Таким образом, если шаблон содержит пару символов "xy", то единственной входящей последовательностью, которая может совпасть с этим шаблоном, является "xy". Символы вроде новой строки и табуляции определяются посредством стандартных управляющих последовательностей, которые начинаются со слэша (\). Например, символ новой строки определяется последовательностью `\n`. В языке регулярных выражений обычный символ называется также литералом.

Класс символов является набором символов. Класс символов можно задать, заключив символы класса в квадратные скобки. Например, класс [wxyz] совпадает с w, x, y или z. Чтобы задать обратный (инвертированный) набор, перед символами необходимо поставить знак ^. Например, [^wxyz] совпадает с любым символом, за исключением w, x, y или z. С помощью дефиса указывается диапазон символов. Например, чтобы задать класс символов, которые будут совпадать с цифрами от 1 до 9, используется запись [1-9].

Групповым символом является символ точки (`.`), который совпадает с любым символом вообще. Таким образом, шаблон, состоящий из ".", будет совпадать с любой из следующих (и любыми другими) входящих последовательностей: "A", "a", "x" и т.д.

Квантификатор определяет, сколько раз совпадает выражение. В табл. 27.6 представлены возможные квантификаторы.

Таблица 27.6. Квантификаторы, применяемые в регулярных выражениях

Квантификатор	Описание
+	Совпадает с одним или более.
*	Совпадает с нулем или более.
?	Совпадает с нулем или одним.

Например, шаблон "x+" будет совпадать с "x", "xx", "xxx" и т.п.

И еще одно: вообще, если вы определите неверное выражение, будет сгенерировано исключение `PatternSyntaxException`.

Пример совпадения с шаблоном

Чтобы лучше понять, как осуществляется сопоставление с шаблоном в регулярном выражении, давайте рассмотрим несколько примеров. В первом из них, показанном далее, производится 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();         // поиск совпадения

    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 SE 6.");
    mat = pat.matcher("Java SE 6"); // создание нового обнаружителя совпадений

    found = mat.matches();         // поиск совпадения

    if(found) System.out.println("Совпадает");
    else System.out.println("Не совпадает");
  }
}
```

Ниже представлены результаты выполнения этой программы:

```
Проверка совпадения Java с Java.
Совпадает

Проверка совпадения Java с Java SE 6.
Не совпадает
```

Давайте проанализируем эту программу. Она начинается с создания шаблона, состоящего из последовательности “Java”. После этого для данного шаблона создается `Matcher` с входящей последовательностью “Java”. Затем вызывается метод `matches()`, с помощью которого определяется, совпадает ли входящая последовательность с шаблоном. Так как последовательность и шаблон одинаковые, метод `matches()` возвращает `true`. Затем создается новый `Matcher` с входящей последовательностью “Java SE 6” и опять вызывается метод `matches()`. В этом случае шаблон и входящая последовательность отличаются друг от друга, поэтому совпадение не обнаруживается. Помните, что метод `matches()` возвращает `true` только в том случае, когда входящая последовательность в точности совпадает с шаблоном. Он не возвращает `true` просто потому, что подпоследовательность не совпадает с шаблоном.

Чтобы определить, содержит ли входная последовательность подпоследовательность, совпадающую с шаблоном, можно вызвать метод `find()`. Рассмотрим следующую программу.

```
// Использование метода 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 SE 6");

    System.out.println("Поиск Java в Java SE 6.");

    if(mat.find()) System.out.println("Подпоследовательность найдена");
    else System.out.println("Совпадений нет");
  }
}
```

Ниже показаны результаты выполнения этой программы:

```
Поиск Java в Java SE 6.
Подпоследовательность найдена
```

В данном случае метод `find()` ищет подпоследовательность "Java".

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. Например, следующая программа находит два случая совпадения с шаблоном "test":

```
// Использование метода find() для нахождения нескольких
подпоследовательностей.
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("test найдена по индексу " + mat.start());
    }
  }
}
```

Ниже представлены результаты выполнения этой программы:

```
test найдена по индексу 0
test найдена по индексу 11
```

Как можно видеть из этих результатов, было найдено два случая совпадения. Программа использует метод `start()` для получения индекса каждого совпадения.

Использование групповых символов и квантификаторов

Хотя в предыдущей программе была показана общая технология использования классов `Pattern` и `Matcher`, она не раскрывает полностью их возможности. Реальное преимущество обработки регулярных выражений оценить невозможно, не используя групповые символы и квантификаторы. Для начала давайте рассмотрим следующий пример, в котором квантификатор + используется для сопоставления любой произвольной последовательности символов W.

```
// Использование квантификатора.
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 представлены результаты выполнения этой программы:

```
Совпадение: W
Совпадение: WW
Совпадение: WWW
```

Как можно видеть из результатов, комбинация "W+" в регулярном выражении совпадает с последовательностью символов W какой угодно длины.

В следующей программе групповой символ используется для формирования шаблона, который будет сопоставляться с любой последовательностью, начинающейся с *e* и заканчивающейся на *d*. Для этого в ней используется групповой символ точки и квантификатор +.

```
// Использование группового символа и квантификатора.
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());
  }
}
```

Результаты выполнения этой программы могут быть для вас неожиданными:

```
Совпадение: extend cup end
```

Был найден только один случай совпадения — это самая длинная последовательность, которая начинается с *e* и заканчивается *d*. Вы могли предположить, что в результате выполнения будет найдено два случая совпадения: "extend" и "end". Более длинная последовательность была обнаружена по той причине, что по умолчанию метод `find()` осуществляет сопоставление с самой длинной последовательностью, которая соответствует всему шаблону. Это называется *поглощающим поведением*. Можно задать *принудительное поведение*, если добавить в комбинацию квантификатор `?`, как показано в следующем варианте программы. В результате будет получен более короткий шаблон для сопоставления.

```
// Использование квантификатора ?.
import java.util.regex.*;

class RegExpr6 {
  public static void main(String args[]) {
    // Использование поведения принудительного сопоставления.
    Pattern pat = Pattern.compile("e.+?d");
    Matcher mat = pat.matcher("extend cup end table");

    while(mat.find())
      System.out.println("Совпадение: " + mat.group());
  }
}
```

Ниже представлены результаты выполнения этой программы:

```
Совпадение: extend
Совпадение: end
```

Как можно видеть из результатов, шаблон “e.+?d” будет совпадать с более короткой последовательностью, которая начинается с *e* и заканчивается на *d*. Таким образом, будет обнаружено два случая совпадения.

Работа с классами символов

Иногда возникает необходимость в сопоставлении какой-нибудь последовательности, состоящей из одного или нескольких символов, в любом порядке, которая будет являться частью набора символов. Например, чтобы сопоставить целые слова, вам потребуется сопоставить любую последовательность букв алфавита. Проще всего это можно сделать с использованием класса символов, определяющего набор символов. Напомним, что для того чтобы сформировать класс символов, необходимых для сопоставления, их потребуется заключить в квадратные скобки. Например, чтобы сопоставить символы нижнего регистра от a до z, используется запись `[a-z]`. В следующей программе демонстрируется этот способ.

```
// Использование класса символов.
import java.util.regex.*;

class RegExpr7 {
  public static void main(String args[]) {
    // Сопоставление слов в нижнем регистре.
    Pattern pat = Pattern.compile("[a-z]+");
    Matcher mat = pat.matcher("this is a test.");

    while(mat.find())
      System.out.println("Совпадение: " + mat.group());
  }
}
```

Ниже показаны результаты выполнения этой программы:

```
Совпадение: this
Совпадение: is
Совпадение: a
Совпадение: test
```

Использование метода `replaceAll()`

С помощью метода `replaceAll()`, определенного в классе `Matcher`, можно выполнять полноценные операции поиска и замены, в которых используются регулярные выражения. Например, в следующей программе каждая последовательность “Jon” заменяется последовательностью “Eric”.

```
// Использование метода 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("Начальная последовательность: " + str);
```

```
    str = mat.replaceAll("Eric ");

    System.out.println("Измененная последовательность: " + str);

  }
}
```

Ниже представлены результаты выполнения этой программы:

```
Начальная последовательность: Jon Jonathan Frank Ken Todd
Измененная последовательность: Eric Eric Frank Ken Todd
```

Поскольку регулярное выражение “Jon.*?” сопоставляет любую строку, начинающуюся с последовательности Jon, за которой больше нет символов или стоит несколько символов, заканчивающихся пробелом, его можно применить для сопоставления и замены имен Jon и Jonathan именем Eric. Такую замену нельзя было бы выполнить, если бы не было возможности осуществить сопоставление с шаблоном.

Использование метода `split()`

С помощью метода `split()` можно сократить входящую последовательность до ее индивидуальных лексем, разделяемых пробелами, запятыми, точками и знаками восклицания.

```
// Использование метода split().
import java.util.regex.*;

class RegExpr9 {
  public static void main(String args[]) {

    // Сопоставление слов в нижнем регистре.
    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 представлены результаты выполнения программы:

```
Следующая лексема: one
Следующая лексема: two
Следующая лексема: alpha9
Следующая лексема: 12
Следующая лексема: done
```

Как можно видеть из результатов, входная последовательность сокращается до ее индивидуальных лексем. Обратите внимание, что разделители не включаются.

Два варианта сопоставления с шаблоном

Хотя описанные методы сопоставления с шаблонами характеризуются высокой степенью гибкости и хорошей производительностью, существуют две альтернативы, применение которых в некоторых обстоятельствах может быть очень полезным. Если вам необходимо только одноразовое сопоставление с шаблоном, вы можете использовать метод `matches()`, определяемый классом `Pattern`. Он показан ниже:

```
static boolean matches(String pattern, CharSequence str)
```

Метод возвращает `true`, если шаблон *`pattern`* совпадает со строкой в параметре *`str`*; в противном случае он возвращает `false`. Этот метод автоматически компилирует *`pattern`*, после чего производит поиск совпадения. Если использовать один и тот же шаблон несколько раз подряд, то применение метода `matches()` будет менее эффективным, чем компиляция шаблона и использование методов сопоставления с шаблонами, определяемых классом `Matches`, о чем было сказано ранее.

Сопоставление с шаблоном можно также выполнять с помощью метода `matches()`, реализуемого классом `String`. Этот метод показан ниже:

```
boolean matches(String pattern)
```

Если вызывающая строка совпадает с регулярным выражением в параметре *`pattern`*, то метод `matches()` возвращает `true`. В противном случае он возвращает `false`.

Изучение регулярных выражений

Обзор регулярных выражений, предложенный в этом разделе, лишь отчасти позволяет постичь их настоящие возможности. Поскольку синтаксический анализ текста, манипулирование и разбиение на лексемы являются частью программирования, то вы, скорее всего, придете к выводу, что подсистема регулярных выражений в Java является хорошим инструментом, который можно использовать с большой пользой. Поэтому вам следует потратить некоторое время на изучение свойств регулярных выражений. Поэкспериментируйте с несколькими различными типами шаблонов и входящих последовательностей. После того как вы разберетесь с тем, как осуществляется сопоставление с шаблонами в регулярных выражениях, вы поймете, что использование этого подхода будет очень полезным при решении многих задач программирования.

Рефлексия

Рефлексия — это способность программного обеспечения к самоанализу. Эта способность обеспечивается пакетом `java.lang.reflect` и элементами `Class`. Рефлексия является важным свойством, особенно для Java Beans. С ее помощью вы сможете анализировать компоненты программного обеспечения и описывать их свойства динамически, во время выполнения, а не во время компиляции. Например, с помощью рефлексии можно определить, какие методы, конструкторы и поля поддерживает конкретный класс. О рефлексии мы начали говорить в главе 12. В настоящей главе мы продолжим эту тему.

Пакет `java.lang.reflect` имеет несколько интерфейсов. Особый интерес представляет интерфейс `Member`, определяющий методы, которые позволяют получить информацию о поле, конструкторе или методе класса. В этом пакете существует также еще восемь классов. Все они перечислены в табл. 27.7.

Следующее приложение иллюстрирует простой вариант использования свойств рефлексии в Java. Оно выводит на экран конструкторы, поля и методы класса `java.awt.Dimension`. Программа начинается с использования метода `forName()` класса `Class` для получения объекта класса для `java.awt.Dimension`. После того как он будет получен, используются методы `getConstructors()`, `getFileds()` и `getMethods()` для анализа этого объекта класса. Они возвращают массивы `Constructor`, `Field` и `Method`, содержащие информацию об объекте.

Таблица 27.7. Классы, определенные в `java.lang.reflect`

Класс	Основное назначение
`AccessibleObject`	Позволяет обходить стандартные проверки управления доступом.
`Array`	Позволяет динамически создавать и манипулировать массивами.
`Constructor`	Предлагает информацию о конструкторе.
`Field`	Предлагает информацию о поле.
`Method`	Предлагает информацию о методе.
`Modifier`	Предлагает информацию о модификаторах доступа к классу и его членам.
`Proxy`	Поддерживает динамические proxy-классы.
`ReflectPermission`	Разрешает рефлексию приватных или защищенных членов класса.

Классы `Constructor`, `Field` и `Method` определяют несколько методов, которые могут использоваться для получения информации об объекте. Вам придется изучить их самостоятельно. Однако каждый из них поддерживает метод `toString()`. Таким образом, использование объектов `Constructor`, `Field` и `Method` в качестве параметров метода `println()` не представляет сложности, что можно видеть из самой программы.

```
// Демонстрация применения рефлексии.
import java.lang.reflect.*;
public class ReflectionDemo1 {
  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 приводятся результаты выполнения этой программы (у вас они могут немного отличаться).

```
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 используются особенности рефлексии в Java для получения общедоступных методов класса. Программа начинается с реализации класса `A`. Метод `getClass()` применяется к этой ссылке на объект и возвращает объект `Class` для класса `A`. Метод `getDeclaredMethods()` возвращает массив объектов `Method`, который описывает только методы, объявленные в этом классе. Методы, унаследованные от суперклассов, например, `Object`, не включаются.

После этого обрабатывается каждый элемент массива `methods`. Метод `getModifiers()` возвращает `int`, содержащий флаги, которые описывают, какие модификаторы доступа применимы к этому элементу. Класс `Modifier` предлагает набор методов, перчисленных в табл. 27.8, которые можно использовать для проверки этого значения. Например, статический метод `isPublic()` возвращает `true`, если параметр включает модификатор доступа `public`. В противном случае он возвращает `false`.

Таблица 27.8. Методы, определяемые классом `Modifier`, которые определяют модификаторы доступа

Методы	Описание
`static boolean isAbstract(int val)`	Возвращает `true`, если *`val`* имеет установленный флаг `abstract`; в противном случае возвращает `false`.
`static boolean isFinal(int val)`	Возвращает `true`, если *`val`* имеет установленный флаг `final`; в противном случае возвращает `false`.
`static boolean isInterface(int val)`	Возвращает `true`, если *`val`* имеет установленный флаг `interface`; в противном случае возвращает `false`.

Окончание табл. 27.8

Методы	Описание
`static boolean isNative(int val)`	Возвращает `true`, если *`val`* имеет установленный флаг `native`; в противном случае возвращает `false`.
`static boolean isPrivate(int val)`	Возвращает `true`, если *`val`* имеет установленный флаг `private`; в противном случае возвращает `false`.
`static boolean isProtected(int val)`	Возвращает `true`, если *`val`* имеет установленный флаг `protected`; в противном случае возвращает `false`.
`static boolean isPublic(int val)`	Возвращает `true`, если *`val`* имеет установленный флаг `public`; в противном случае возвращает `false`.
`static boolean isStatic(int val)`	Возвращает `true`, если *`val`* имеет установленный флаг `static`; в противном случае возвращает `false`.
`static boolean isStrict(int val)`	Возвращает `true`, если *`val`* имеет установленный флаг `strict`; в противном случае возвращает `false`.
`static boolean isSynchronized(int val)`	Возвращает `true`, если *`val`* имеет установленный флаг `synchronized`; в противном случае возвращает `false`.
`static boolean isTransient(int val)`	Возвращает `true`, если *`val`* имеет установленный флаг `transient`; в противном случае возвращает `false`.
`static boolean isVolatile(int val)`	Возвращает `true`, если *`val`* имеет установленный флаг `volatile`; в противном случае возвращает `false`.

В следующей программе, если метод является общедоступным, то с помощью метода `getName()` будет получено его имя, и затем выведено на экран.

```
// Отображение общедоступных методов.
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("Общедоступные 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() {
  }
}
```

Ниже приводятся результаты выполнения этой программы:

```
Общедоступные методы:
a1
a2
```

Удаленный вызов методов

Удаленный вызов методов (Remote Method Invocation — RMI) позволяет Java-объектам, выполняющимся на одном компьютере, вызывать методы Java-объектов, которые выполняются на другом компьютере. Это важная возможность, поскольку она позволяет создавать распределенные приложения. Несмотря на то что обсуждение RMI выходит за рамки этой книги, в приведенном ниже примере демонстрируются основные принципы этого подхода.

Клиент-серверное приложение, использующее RMI

В этом разделе предложено пошаговое руководство к созданию простого клиент-серверного приложения с использованием RMI. Сервер получает запрос от клиента, обрабатывает его и возвращает результат. В этом примере запрос состоит из двух чисел. Сервер суммирует их и возвращает полученный результат.

Шаг первый: ввод и компиляция исходного кода

Это приложение использует четыре исходных файла. В первом файле, `AddServerIntf.java`, определяется удаленный интерфейс, который будет предоставлен сервером. Он содержит один метод, принимающий два параметра `double`, и возвращает их сумму. Все удаленные интерфейсы должны расширять интерфейс `Remote`, который является частью `java.rmi`. `Remote` не определяет членов. Он предназначен лишь для того, чтобы показать, что интерфейс использует удаленные методы. Все удаленные методы могут генерировать исключение `RemoteException`.

```
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. Все удаленные объекты должны расширять `UnicastRemoteObject`, который обеспечивает функциональность, необходимую для того, чтобы сделать доступными объекты для удаленных компьютеров.

```
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 компьютера-сервера. Его главная задача — обновлять реестр RMI на этом компьютере. Это делается с помощью метода rebind() класса Naming (пакет java.rmi). Этот метод связывает имя со ссылкой на объект. Первым параметром метода rebind() является строка, которая присваивает серверу имя "AddServer". Его второй параметр является ссылкой на экземпляр 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 сторона распределенного приложения. Этому файлу требуются три аргумента командной строки. Первый из них является IP-адресом или именем компьютера-сервера. Второй и третий параметры являются двумя числами, которые необходимо суммировать.

Приложение начинается с формирования строки, имеющей синтаксическую структуру URL-адреса. Этот URL-адрес использует протокол rmi. Строка включает IP-адрес или имя сервера и строку "AddServer". Затем программа вызывает метод lookup() класса Naming. Этот метод принимает один параметр, URL-адрес rmi, и возвращает ссылку на объект типа AddServerIntf. Впоследствии всякий удаленный вызов метода можно будет направлять этому объекту.

Затем программа отображает параметры и вызывает удаленный метод add(). Этот метод возвращает результат суммирования, который выводится на экран.

```
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);
    }
  }
}
```

После того как вы введете весь код, используйте javac для компилирования четырех исходных файлов, созданных вами.

Шаг второй: генерирование заглушки

Прежде чем вы сможете работать с клиентом и сервером, нужно сгенерировать заглушку. В контексте RMI *заглушка* представляет собой Java-объект, который находится на компьютере-клиенте. Ее функция заключается в представлении тех же интерфейсов, которые предлагает удаленный сервер. Удаленные вызовы методов, генерируемые клиентом, направляются в заглушку. Заглушка работает с остальными частями системы RMI для формирования запроса, отправляемого удаленному компьютеру.

Удаленный метод может принимать параметры, которые представляют собой простые типы или объекты. В последнем случае объект может иметь ссылки на другие объекты. Вся эта информация должна быть отправлена удаленному компьютеру. То есть объект, передаваемый в качестве аргумента вызову удаленного метода, должен быть сериализован и отправлен на удаленный компьютер. Вспомните, в главе 19 мы говорили, что при сериализации также рекурсивно обрабатываются все объекты, на которые имеются ссылки.

Если клиенту необходимо вернуть отклик, то весь процесс происходит в обратном порядке. Обратите внимание, что сериализация и десериализация используются также и при возвращении объектов клиенту.

Для генерации заглушки применяется инструментальное средство, называемое *компилятором RMI*, вызов которого производится из командной строки, как показано ниже:

```
rmic AddServerImpl
```

Эта команда генерирует файл `AddServerImpl_Stub.class`. В случае использования `rmic` убедитесь в том, что текущий каталог включен в переменную `CLASSPATH`.

Шаг третий: инсталляция файлов на компьютере-сервере и клиенте

Скопируйте `AddClient.class`, `AddServerImpl_Stub.class` и `AddServerIntf.class` в каталог на компьютере-клиенте. Скопируйте `AddServerIntf.class`, `AddServerImpl.class`, `AddServerImpl_Stub.class` и `AddServer.class` в каталог на компьютере-сервере.

На заметку! *RMI обладает технологиями динамической загрузки классов, однако они не применяются в приведенных здесь примерах. Наоборот, все файлы, используемые приложениями клиента и сервера, должны быть инсталлированы вручную на соответствующих компьютерах.*

Шаг четвертый: запуск реестра RMI на компьютере-сервере

В Java SE 6 имеется программа `rmiregistry`, которая выполняется на стороне сервера. Она преобразует имена в ссылки на объекты. Для начала нужно проверить, включен ли каталог, в котором находятся ваши файлы, в переменную окружения `CLASSPATH`. Затем потребуется запустить RMI Registry из командной строки, как показано ниже:

```
start rmiregistry
```

После выполнения этой команды на экране появится новое окно. Это окно нужно оставлять открытым все время, пока вы будете экспериментировать с примером RMI.

Шаг пятый: запуск сервера

Код сервера запускается из командной строки, как показано ниже:

```
java AddServer
```

Вспомните, что код AddServer реализует AddServerImpl и регистрирует этот объект под именем "AddServer".

Шаг шестой: запуск клиента

Программное обеспечение AddClient требует для своей работы три аргумента: имя или IP-адрес компьютера-сервера и два числа, которые будут просуммированы. Вы можете вызвать его из командной строки, используя один из следующих форматов:

```
java AddClient server1 8 9
java AddClient 11.12.13.14 8 9
```

В первой строке указывается имя сервера. Во второй строке используется его IP-адрес (11.12.13.14).

Можно попытаться выполнить этот пример и без удаленного сервера. Для этого достаточно установить все программы на одном и том же компьютере, запустить rmiregistry, запустить AddServer и после этого выполнить AddClient следующим образом:

```
java AddClient 127.0.0.1 8 9
```

Здесь адрес 127.0.0.1 является адресом "обратной связи" для локального компьютера. Использование этого адреса позволит выполнить весь механизм RMI, не инсталлируя сервер на удаленном компьютере.

В любом из случаев, результат выполнения этой программы будет таким:

```
Первое число: 8
Второе число: 9
Сумма: 17.0
```

Форматирование текста

Пакет java.text позволяет форматировать, производить поиск и манипулировать текстом. В главе 32 продемонстрирован класс NumberFormat, который используется для форматирования числовых данных. В этом разделе рассматриваются два наиболее часто используемых класса, предназначенные для форматирования даты и времени.

Класс DateFormat

DateFormat является абстрактным классом, с помощью которого можно форматировать и анализировать показания даты и времени. Метод getDateInstance() возвращает экземпляр DateFormat, который может форматировать информацию о дате. Можно использовать одну из следующих его форм:

```
static final DateFormat getDateInstance()
static final DateFormat getDateInstance(int style)
static final DateFormat getDateInstance(int style, Locale locale)
```

Параметр *style* может принимать следующие значения: DEFAULT, SHORT, MEDIUM, LONG или FULL. Они представляют собой константы int, определяемые в классе DateFormat. С их помощью можно представлять различные подробные сведения, связанные с датой. Параметр *locale* представляет одну из статических ссылок, определяемых классом Locale (см. главу 18). Если параметры *style* и/или *locale* не будут заданы, будут использоваться значения, принятые по умолчанию.

Одним из наиболее часто используемых методов в этом классе является `format()`. Он имеет несколько перегруженных форм, среди которых можно выделить следующую:

```
final String format(Date d)
```

Параметром является объект `Date`, который необходимо отобразить. Метод возвращает строку, содержащую отформатированную информацию.

В следующем коде показано, как производится форматирование информации о дате. Он начинается с создания объекта `Date`. Этот объект хранит информацию о текущей дате и времени. Затем он выводит информацию о дате с использованием различных стилей и с учетом местной специфики представления времени.

```
// Иллюстрация форматов даты.
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 представлен пример результатов выполнения этой программы:

```
Япония: 07/05/23
Корея: 2007. 5.23
Великобритания: 23 May 2007
США: Wednesday, May 23, 2007
```

Метод `getTimeInstance()` возвращает экземпляр `DateFormat`, который может форматировать информацию о времени. Он имеет следующие варианты:

```
static final DateFormat getTimeInstance()
static final DateFormat getTimeInstance(int style)
static final DateFormat getTimeInstance(int style, Locale locale)
```

Параметр *`style`* принимает одно из следующих значений: `DEFAULT`, `SHORT`, `MEDIUM`, `LONG` и `FULL`. Они являются константами `int`, определяемыми в классе `DateFormat`. С их помощью можно определить, насколько подробным будет представление времени. Параметр *`locale`* является одной из статических ссылок, определенных в классе `Locale`. Если параметры *`style`* и *`locale`* не будут заданы, будут использоваться значения, принятые по умолчанию.

В следующем коде показано, как производится форматирование информации о времени. Он начинается с создания объекта `Date`, который получает информацию о текущей дате и времени, а затем выводит информацию о времени, используя различные стили и учитывая местную специфику представления времени.

```
// Иллюстрация форматов 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 показан пример результатов выполнения этой программы:

```
Япония: 20:25
Великобритания: 20:25:14 CDT
Канада: 8:25:14 o'clock PM CDT
```

Класс `DateFormat` также имеет метод `getDateTimeInstance()`, который может форматировать информацию о дате и времени. Если хотите, поэкспериментируйте с ним самостоятельно.

Класс `SimpledateFormat`

Класс `SimpleDateFormat` является конкретным подклассом класса `DateFormat`. Он позволяет определять свои собственные шаблоны форматирования, которые можно будет использовать для отображения информации о дате и времени.

Ниже показан один из его конструкторов:

```
SimpleDateFormat(String formatString)
```

Параметр *`formatString`* описывает способ отображения даты и времени. Ниже показан один из примеров его применения:

```
SimpleDateFormat sdf = SimpleDateFormat("dd MMM yyyy hh:mm:ss zzz");
```

Символы, используемые в строке форматирования, определяют отображаемую информацию. В табл. 27.9 приводятся эти символы, и дается описание каждого из них.

Таблица 27.9. Символы форматирования строки для `SimpleDateFormat`

Символ	Описание
`A`	AM или PM
`D`	День месяца (1–31)
`H`	Часы в формате AM/PM (1–12)
`K`	Часы в формате суток (1–24)
`M`	Минуты в часе (0–59)
`S`	Секунды в минуте (0–59)
`W`	Неделя в году (1–52)
`y`	Год

Окончание табл. 27.9

Символ	Описание
z	Часовой пояс
D	День в году (1–366)
E	День недели (например, Thursday)
F	День недели в месяце
G	Эпоха (т.е. AD — после Рождества Христова, BC — до Рождества Христова)
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)
Z	Часовой пояс в формате, описанном в документе RFC 822

В большинстве случаев количество повторений символа определяет способ представления даты. Текстовая информация отображается в виде аббревиатуры, если буква шаблона повторяется менее четырех раз. В противном случае используется форма без применения аббревиатуры. Например, шаблон zzzz может отобразить Pacific Daylight Time (тихоокеанское время), а шаблон zzz может отобразить PDT.

Для чисел количество повторений буквы шаблона определяет количество цифр в представлении. Например, hh:mm:ss может представить 01:51:15, в то время как h:m:s показывает то же время в виде 1:51:15.

И, наконец, M или MM отвечает за отображение месяца в виде одной или двух цифр. Три или более повторения M приведет к тому, что месяц будет отображаться в виде текстовой строки.

В следующей программе демонстрируется использование этого класса.

```
// Демонстрация применения 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 пример результатов выполнения этой программы:

```
10:25:03
23 may 2007 10:25:03 CDT
Wed May 23 2007
```

III

ЧАСТЬ

Разработка программного обеспечения с использованием Java

28 ГЛАВА

Java Beans

В этой главе представлен обзор Java Beans (bean-компонентов Java). Важность bean-компонентов бесспорна, так как они позволяют строить сложные системы на основе программных компонентов. Эти компоненты можно создавать самостоятельно, а также приобретать у сторонних разработчиков. Bean-компоненты Java определяют архитектуру, устанавливающую порядок взаимодействия этих строительных блоков.

Чтобы вам легче было понять, что собой представляют bean-компоненты, рассмотрим пример. У разработчиков компьютерного оборудования имеется множество компонентов, которые можно интегрировать друг с другом для построения системы. Резисторы, конденсаторы и катушки индуктивности являются примерами простых компоновочных блоков. Интегральные схемы предлагают еще больше функциональных возможностей. При этом каждая из этих различных частей пригодна для многократного использования. Их не нужно компоновать заново каждый раз, когда возникает необходимость в новой системе. А еще эти же элементы могут использоваться в схемах различных типов. Это возможно благодаря тому, что поведение этих компонентов понятно и хорошо документировано.

В индустрии программного обеспечения также пытались воспользоваться преимуществами многократного использования компонентов и их способностью к взаимодействию. Для этого необходимо было разработать такую компонентную архитектуру, которая позволила бы собирать программы на основе программных компоновочных блоков, пусть даже и предлагаемых сторонними разработчиками. Кроме того, проектировщик должен был иметь возможность выбрать компонент, разобраться с его функциями и внедрить его в приложение. С появлением новой версии компонента необходимо, чтобы его функциональные возможности можно было без труда внедрять в существующий код. К счастью, такую архитектуру как раз и предлагают bean-компоненты Java.

Что такое bean-компонент Java?

Bean-компонент Java представляет собой компонент программного обеспечения и предназначен для многократного использования во множестве различных сред. Bean-компонент не имеет ограничений в плане функциональности. Он может выполнять простую

функцию, например, получать стоимость товарно-материальных запасов, а может реализовать и более сложную функцию, например, предсказывать котировку акций компании. Bean-компонент может быть видимым для конечного пользователя. Одним из примеров такого компонента является кнопка графического интерфейса пользователя. Bean-компонент может быть также и невидимым для пользователя. Таким компоновочным блоком является ПО для декодирования потока мультимедиа-информации в реальном времени. И, наконец, bean-компонент может быть предназначен для работы в автономном режиме на рабочей станции пользователя или для работы в комплексе с другими распространяемыми компонентами. Примером bean-компонента, который может работать локально, является ПО для генерирования секторной диаграммы на основе набора точек данных. А, например, bean-компонент, предоставляющий информацию о ценах на фондовой или товарной бирже в реальном времени, может работать только в комплексе с другим распространяемым ПО, получая от него необходимые данные.

Преимущества bean-компонентов Java

В следующем списке перечислены некоторые преимущества, которые предлагает технология bean-компонентов Java разработчику компонентов.

- Bean-компонент обладает всеми преимуществами парадигмы Java, гласящей: “написано однажды, работает везде”.
- Можно управлять свойствами, событиями и методами bean-компонента, доступными другому приложению.
- Для конфигурирования bean-компонента можно применять вспомогательное ПО. Оно будет необходимо только при настройке параметров времени проектирования данного компонента. В среду времени выполнения его включать не нужно.
- Настройки параметров конфигурации bean-компонента можно хранить на постоянном носителе информации и восстанавливать по мере необходимости.
- bean-компонент может регистрироваться на получение событий от других объектов, и может генерировать события, отправляемые другим объектам.

Самодиагностика

В основе bean-компонентов Java лежит свойство *самодиагностики* (introspection). Самодиагностика — это процесс анализа bean-компонента, при котором определяются его характеристики. На самом деле это очень важная особенность Java Beans API, поскольку с ее помощью другое приложение, например инструмент разработки, может получать информацию о данном компоненте. Без самодиагностики технология bean-компонентов Java работать не будет.

Существуют два способа, с помощью которых разработчик bean-компонента может показать, какие его свойства, события и методы будут доступны. В одном случае используются простые соглашения об именовании. Они позволяют механизмам самодиагностики логически выводить информацию о bean-компоненте. В другом случае применяется дополнительный класс, расширяющий интерфейс `BeanInfo`, который явным образом предоставляет эту информацию. Рассмотрением обеих способов мы сейчас и займемся.

Проектные шаблоны для свойств

Свойство (property) bean-компонента представляет собой сокращенный вариант его состояния. Значения, присваиваемые свойствам, определяют поведение и появление этого компонента. Настройка свойства осуществляется посредством *метода записи* (setter method), а его получение — посредством *метода чтения* (getter method). Свойства бывают двух видов: простые и индексированные.

Простые свойства

Простое свойство имеет одно значение. Его можно идентифицировать с помощью следующих проектных шаблонов, в которых `N` — это имя свойства, а `T` — его тип:

```
public T getN( );
public void setN(T arg);
```

Каждый метод имеет свойство чтения/записи для доступа к своим значениям. Свойство только для чтения имеет только метод `get`. Свойство только для записи имеет только метод `set`.

Ниже представлен пример трех простых свойств чтения/записи, а также их методов чтения и записи.

```
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;
}
```

Индексированные свойства

Индексированное свойство состоит из нескольких значений. Его можно идентифицировать с помощью следующих проектных шаблонов, в которых `N` — это имя свойства, а `T` — его тип:

```
public T getN(int index);
public void setN(int index, T value);
public T[] getN( );
public void setN(T values[]);
```

Ниже показан пример индексированного свойства по имени `data`, а также его методов чтения и записи.

```
private double data[ ];

public double getData(int index) {
  return data[index];
}
public void setData(int index, double value) {
  data[index] = value;
}
public double[ ] getData( ) {
  return data;
}
public void setData(double[ ] values) {
  data = new double[values.length];
  System.arraycopy(values, 0, data, 0, values.length);
}
```

Проектные шаблоны для событий

Bean-компоненты используют модель делегации событий, которая уже была рассмотрена в этой книге. Bean-компоненты способны генерировать события и посылать их другим объектам. События могут идентифицироваться с помощью следующих проектных шаблонов, в которых `T` представляет тип события:

```
public void addTListener(TListener eventListener)
public void addTListener(TListener eventListener)
                    throws java.util.TooManyListenersException
public void removeTListener(TListener eventListener)
```

Эти методы используются для того, чтобы добавить или удалить слушателя для указанного события. Вариант метода `AddTListener()`, который не генерирует исключение, можно применять для *групповой передачи* события. Под групповой передачей понимается то, что на получение уведомлений о событии может быть зарегистрировано несколько слушателей. Вариант метода, генерирующего исключение `TooManyListenersException`, передает событие *индивидуально* — уведомление о нем может получить только один слушатель. В любом случае, метод `removeTListener()` служит для удаления слушателя. Например, если предположить, что существует тип интерфейса события, называемый `TemperatureListener`, то bean-компонент, который следит за температурой, может предложить следующие методы:

```
public void addTemperatureListener(TemperatureListener tl) {
...
}
public void removeTemperatureListener(TemperatureListener tl) {
...
}
```

Методы и проектные шаблоны

Проектные шаблоны не используются для именования методов, не связанных со свойствами. Механизм самодиагностики находит все общедоступные методы bean-компонента. Защищенные и приватные методы остаются недоступными.

Использование интерфейса BeanInfo

Как было сказано выше, проектные шаблоны *неявно* определяют, какая информация является доступной пользователю bean-компонента. Интерфейс BeanInfo позволяет *явно* управлять доступом к информации. Интерфейс BeanInfo определяет несколько методов, включая перечисленные ниже:

```
PropertyDescriptor[] getPropertyDescriptors()
EventSetDescriptor[] getEventSetDescriptors()
MethodDescriptor[] getMethodDescriptors()
```

Эти методы возвращают массивы объектов, содержащие информацию о свойствах, событиях и методах bean-компонента. Классы PropertyDescriptor, EventSetDescriptor и MethodDescriptor определены в модуле java.beans и описывают указанные элементы. Реализуя эти методы, разработчик может в точности определить, что конкретно доступно пользователю, не прибегая к самодиагностике, выполняемой на основе проектных шаблонов.

Создавая класс, реализующий интерфейс BeanInfo, к нему нужно обращаться как к *имя*BeanInfo, где *имя* — имя bean-компонента. Например, если bean-компонент называется MyBean, то информационный класс должен выглядеть как MyBeanBeanInfo.

Чтобы упростить использование интерфейса BeanInfo, bean-компоненты Java предлагают класс SimpleBeanInfo. Он обеспечивает стандартные реализации интерфейса BeanInfo, включая только что представленные три метода. Этот класс можно расширить и переопределить один или более методов, чтобы явно управлять доступными аспектами bean-компонента. Если метод не переопределить, будет использоваться самодиагностика проектного шаблона. Например, если не переопределять метод getPropertyDescriptors(), то для определения свойств bean-компонента будут применяться проектные шаблоны. Далее в этой главе вы сможете увидеть класс SimpleBeanInfo в действии.

Связанные и ограниченные свойства

Bean-компонент, имеющий *связанное* (bound) свойство, генерирует событие во время изменения свойства. Генерируемое событие имеет тип PropertyChangeEvent и отправляется объектам, которые предварительно зарегистрировались на получение таких уведомлений. Класс, осуществляющий обработку этого события, должен реализовывать интерфейс PropertyChangeListener.

Bean-компонент, имеющий *ограниченное* (constrained) свойство, генерирует событие при попытке изменения его значения. Он также генерирует событие, имеющее тип PropertyChangeEvent. Это событие посылается объектам, предварительно зарегистрировавшимся на получение таких уведомлений. Однако эти объекты могут отклонить предложенное изменение, сгенерировав исключение PropertyVetoException. Благодаря этой особенности bean-компонент может работать по-разному в зависимости от среды времени выполнения. Класс, осуществляющий обработку этого события, должен реализовывать интерфейс VetoableChangeListener.

Постоянство

Постоянство (persistence) — это способность сохранять текущее состояние bean-компонента (включая значения свойств bean-компонентов и переменные экземпляров) на энергонезависимом запоминающем устройстве и извлекать его по мере необходимости.

Для обеспечения постоянства bean-компонентов используются возможности сериализации, которые предлагают библиотеки классов Java.

Самый простой способ сериализировать bean-компонент заключается в том, чтобы он реализовывал интерфейс `java.io.Serializable`, который является просто маркерным интерфейсом. Реализация интерфейса `java.io.Serializable` обеспечит автоматическое выполнение сериализации. Вашему bean-компоненту не нужно будет предпринимать никаких других действий. Автоматическую сериализацию также можно наследовать. Другими словами, если какой-то суперкласс bean-компонента будет реализовывать интерфейс `java.io.Serializable`, он приобретет возможность автоматической сериализации. Существует одно важное ограничение: любой класс, реализующий интерфейс `java.io.Serializable`, должен предоставлять конструктор без параметров.

При автоматической сериализации можно выборочно отключить сохранение отдельного поля с помощью ключевого слова `transient`. Таким образом, элементы данных bean-компонента, определенные как `transient`, сериализироваться не будут.

Если bean-компонент не реализует интерфейс `java.io.Serializable`, вы должны обеспечить возможность сериализации самостоятельно (например, с помощью интерфейса `java.io.Externalizable`). В противном случае контейнеры не смогут хранить конфигурацию вашего компонента.

Конфигураторы

Разработчик bean-компонентов может предусмотреть возможность использования *конфигуратора* (customizer), с помощью которого другой разработчик сможет конфигурировать эти bean-компоненты. Конфигуратор может обеспечивать пошаговое руководство всем процессом конфигурирования, выполняя указания которого можно добиться того, чтобы компонент использовался в определенном контексте. Можно также предоставить онлайновую документацию. У разработчика bean-компонентов будет достаточно возможностей для того, чтобы разработать такой конфигуратор, который сможет обособить его продукт на рынке.

Java Beans API

Функциональные возможности bean-компонентов Java обеспечиваются классами и интерфейсами модуля `java.beans`. В этом разделе приводится краткий обзор содержимого этого модуля. В табл. 28.1 представлен список интерфейсов модуля `java.beans` и дается краткое описание их функциональных возможностей. В табл. 28.2 приведен список классов модуля `java.beans`.

Хотя эта глава не позволяет рассказать обо всех классах, мы поговорим о четырех из них, представляющих определенный интерес: `Introspector`, `PropertyDescriptor`, `EventSetDescriptor` и `MethodDescriptor`.

Introspector

Класс `Introspector` предлагает несколько статических методов, поддерживающих самодиагностику. Наиболее интересным из них является метод `getBeaninfo()`. Он возвращает объект `BeanInfo`, который может использоваться для получения информации о bean-компоненте.

Таблица 28.1. Интерфейсы модуля `java.beans`

Интерфейс	Описание
`AppletInitializer`	Методы этого интерфейса используются для инициализации bean-компонентов, которые также являются аплетами.
`BeanInfo`	Этот интерфейс позволяет разработчику определять информацию о свойствах, событиях и методах bean-компонента.
`Customizer`	Этот интерфейс позволяет разработчику предоставить графический интерфейс пользователя, посредством которого можно конфигурировать bean-компонент.
`DesignMode`	Методы этого интерфейса определяют, выполняется ли bean-компонент в режиме проектирования.
`ExceptionListener`	Метод этого интерфейса вызывается во время возникновения исключения.
`PropertyChangeListener`	Метод этого интерфейса вызывается при изменении ограниченного свойства.
`PropertyEditor`	Объекты, реализующие этот интерфейс, позволяют разработчикам изменять и отображать значения свойств.
`VetoableChangeListener`	Метод этого интерфейса вызывается при изменении ограниченного свойства.
`Visibility`	Методы этого интерфейса позволяют bean-компоненту работать в тех средах, в которых нет графического интерфейса пользователя.

Таблица 28.2. Классы модуля `java.beans`

Класс	Описание
`BeanDescriptor`	Этот класс предлагает информацию о bean-компоненте. Он также позволяет связывать конфигуратор с bean-компонентом.
`Beans`	Этот класс используется для получения информации о bean-компоненте.
`DefaultPersistenceDelegate`	Подкласс класса `PersistenceDelegate`.
`Encoder`	Зашифровывает состояние совокупности bean-компонентов. Может использоваться для записи этой информации в поток.
`EventHandler`	Поддерживает динамическое создание слушателя событий.
`EventSetDescriptor`	Экземпляры этого класса описывают событие, которое может генерироваться bean-компонентом.
`Expression`	Инкапсулирует вызов метода, возвращающего результат.
`FeatureDescriptor`	Суперкласс классов `Propertydescriptor`, `EventSetDescriptor` и `MethodDescriptor`.
`IndexedPropertyChangeEvent`	Подкласс класса `PropertyChangeEvent`, представляющий изменение индексированного свойства.
`IndexedPropertyDescriptor`	Экземпляры этого класса описывают индексированное свойство bean-компонента.

Окончание табл. 28.2

Класс	Описание
IntrospectionException	Выражение этого типа генерируется, если во время анализа bean-компонента возникает ошибка.
Introspector	Этот класс анализирует bean-компонент и создает объект BeanInfo, описывающий компонент.
MethodDescriptor	Экземпляры этого класса описывают метод bean-компонента
ParameterDescriptor	Экземпляры этого класса описывают параметр метода.
PersistenceDelegate	Обрабатывает информацию о состоянии объекта.
PropertyChangeEvent	Это событие генерируется при изменении связанных или ограниченных свойств. Оно посылается объектам, которые зарегистрировались на получение уведомлений об этих событиях и которые реализуют один из интерфейсов PopertyChangeListener и VetoableChangeListener.
PropertyChangeListenerProxy	Расширяет класс EventListenerProxy и реализует интерфейс PropertyChangeListener.
PropertyChangeSupport	Bean-компоненты, поддерживающие ограниченные свойства, могут использовать этот класс для уведомления объектов PropertyChangeListener.
PropertyDescriptor	Экземпляры этого класса описывают свойство bean-компонента.
PropertyEditorManager	Этот класс обнаруживает объект PropertyEditor для данного типа.
PropertyEditorSupport	Этот класс предлагает функциональные возможности, которые могут использоваться при написании редакторов свойств.
PropertyVetoException	Исключение данного типа генерируется при отклонении изменения ограниченного свойства.
SimpleBeanInfo	Этот класс предлагает функциональные возможности, которые могут использоваться при написании классов BeanInfo.
Statement	Инкапсулирует вызов метода.
VetoableChangeListenerProxy	Расширяет класс EventListenerproxy и реализует интерфейс VetoableChangeListener.
VetoableChangeSupport	Bean-компоненты, которые поддерживают ограниченные свойства, могут использовать этот класс для уведомления объектов VetoableChangeListener.
XMLDecoder	Используется для чтения bean-компонента из XML-документа.
XMLEncoder	Используется для записи bean-компонента в XML-документ.

Метод getBeanInfo() имеет несколько форм, включая следующую:

```
static BeanInfo getBeanInfo(Class<?> bean) throws IntrospectionException
```

Возвращаемый объект содержит информацию о заданном bean-компоненте.

PropertyDescriptor

Класс PropertyDescriptor описывает свойство bean-компонента. Он поддерживает несколько методов, которые управляют свойствами и описывают их. Например, вы можете определить, ограничено ли свойство, вызовом функции isBound(). Чтобы определить, является ли свойство ограниченным, нужно вызвать функцию isConstrained(). Имя свойства можно получить посредством вызова функции getName().

EventSetDescriptor

Класс EventSetDescriptor представляет событие bean-компонента. Он поддерживает несколько методов, которые получают методы, используемые bean-компонентом для добавления/удаления слушателей событий или для управления событиями. Например, чтобы получить метод, служащий для добавления слушателей, нужно вызвать метод getAddListenerMethod(). Чтобы получить метод, используемый для удаления слушателей, необходимо вызвать метод getRemoveListenerMethod(). Чтобы получить тип слушателя, следует вызвать метод getListenerType(). Имя события можно получить, если вызвать метод getName().

MethodDescriptor

Класс MethodDescriptor представляет метод bean-компонента. Чтобы получить имя метода, нужно вызвать метод getName(). Информацию о методе можно получить с помощью метода getMethod(), который показан ниже:

```
Method getMethod()
```

При этом возвращается объект типа Method, описывающий данный метод.

Пример bean-компонента

В завершение этой главы предлагается пример, иллюстрирующий различные аспекты программирования bean-компонентов, включая самодиагностику и использование класса BeanInfo. В нем также участвуют классы Introspector, PropertyDesciptor и EventSetDescriptor. В примере используется три класса. Ниже показан первый из них, который является bean-компонентом Colors.

```
// Простой bean-компонент.
import java.awt.*;
import java.awt.event.*;
import java.io.Serializable;

public class Colors extends Canvas implements Serializable {
  transient private Color color; // не постоянный
  private boolean rectangular;   // постоянный

  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);
    }
  }
}
```

Bean-компонент `Colors` отображает цветной объект в рамке. Цвет компонента определяется приватной переменной `color` типа `Color`, а его форма — приватной переменной `rectangular` типа `boolean`. Конструктор определяет анонимный внутренний класс, расширяющий класс `MouseAdapter`, и переопределяет метод `mousePressed()`. Метод `change()` вызывается в ответ на щелчок кнопкой мыши. Он выбирает случайный цвет и перекрашивает компонент. Методы `getRectangular()` и `setRectangular()` обеспечивают доступ к одному свойству данного bean-компонента. Метод `change()` вызывает метод `randomColor()` для выбора цвета, а затем вызывает метод `repaint()`, чтобы сделать изменение видимым. Обратите внимание, что метод `paint()` использует переменные `rectangular` и `color` для определения представления bean-компонента.

Следующим классом является `ColorsBeanInfo`. Он является подклассом класса `SimpleBeanInfo`, который предлагает явную информацию о `Colors`. В нем переопределяется метод `getPropertyDescriptors()` для указания того, какие свойства будут доступны пользователю bean-компонента. В данном случае единственным свойством, доступным пользователю, является `rectangular`. Метод создает и возвращает объект `PropertyDescriptor` для свойства `rectangular`. Ниже показан используемый конструктор `PropertyDescriptor`:

```
PropertyDescriptor(String property, Class<?> beanCls)
throws IntrospectionException
```

Два параметра представляют, соответственно, имя свойства и класс bean-компонента.

```
// Информационный класс bean-компонента.
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("Возникло исключение. " + e);
    }
    return null;
  }
}
```

Последним классом является класс `IntrospectorDemo`. Он использует самодиагностику для отображения свойств и событий, доступных в bean-компоненте `Colors`.

```
// Показ свойств и событий.
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("Возникло исключение. " + e);
    }
  }
}
```

Ниже показан вывод, полученный в результате выполнения этой программы:

```
Свойства:
      rectangular
```

```
События:
       mouseWheel
       mouse
       mouseMotion
       component
       hierarchyBounds
       focus
       hierarchy
       propertyChange
       inputMethod
       key
```

В этом выводе следует обратить внимание на две особенности. Во-первых, поскольку класс `ColorsBeanInfo` заменяет метод `getPropertyDescriptors()`, так что единственным возвращаемым свойством является `rectangular`, то отображается только свойство `rectangular`. Однако поскольку метод `getEventSetDescriptors()` не переопределяется в классе `ColorsBeanInfo`, используется самодиагностика проектного шаблона и обнаруживаются все события, включая те, которые принадлежат классу `Colors` суперкласса `Canvas`. Помните, что если не переопределить один из методов "get", определенных в классе `SimpleBeanInfo`, то по умолчанию будет использована самодиагностика проектного шаблона. Чтобы увидеть, какие изменения были произведены классом `ColorsBeanInfo`, нужно удалить его файл класса и запустить еще раз `IntrospectorDemo`. На этот раз он предоставит отчет по большему количеству свойств.

29 ГЛАВА

Введение в Swing

В части II настоящей книги вы могли видеть, как с помощью классов AWT можно создавать пользовательские интерфейсы. Несмотря на то что AWT по-прежнему является важной частью Java, его набор компонентов применяется для создания графических пользовательских интерфейсов уже не так широко. Сегодня многие программисты пользуются для этой цели классами Swing. Swing — это набор классов, которые предлагают более мощные и гибкие GUI-компоненты, чем AWT. Вообще говоря, современный графический пользовательский интерфейс Java построен на основе Swing.

Рассмотрению Swing посвящено две главы. В этой главе вы ознакомитесь с основами Swing. Она начинается с описания основных концепций Swing. Затем будет представлена общая форма Swing-программы, включая приложения и аплеты. В конце главы мы объясним, как с помощью Swing можно делать рисунки. В следующей главе будет рассмотрено несколько часто используемых компонентов Swing. Важно понять, что количество классов и интерфейсов в пакетах Swing довольно большое, и что каждый из них просто невозможно рассмотреть в этой книге. (Вообще говоря, чтобы полностью рассказать о Swing, потребуется отдельная книга.) И все же в этих двух главах вы сможете узнать о самом главном, что касается Swing.

На заметку! *Более содержательное описание Swing можно найти в моей книге Swing: A Beginner's Guide, вышедшей в издательстве McGraw-Hill/Osborne (2007 год).*

Истоки Swing

В начале существования Java классов Swing не было вообще. Причиной разработки классов Swing было слабое место в исходной подсистеме GUI Java: AWT. AWT определяет базовый набор элементов управления, окон и диалогов, которые поддерживают пригодный к использованию, но ограниченный в возможностях графический интерфейс. Одной из причин ограниченности AWT является то, что AWT преобразует свои различные визуальные компоненты в соответствующие им эквиваленты, не зависящие от платформы, которые называются *равноправными компонентами*. Это означает, что внешний вид ком-

понента определяется платформой, а не закладывается в Java. Поскольку компоненты AWT используют "родные" ресурсы кода, они называются *тяжеловесными*.

Использование "родных" равноправных компонентов порождает некоторые проблемы. Во-первых, в связи с различиями, существующими между операционными системами, компонент может выглядеть или даже вести себя по-разному на различных платформах. Такая изменчивость шла вразрез с философией Java: "написано однажды, работает везде". Во-вторых, внешний вид каждого компонента был фиксированным (так как все зависело от платформы), и это нельзя было изменить (по крайней мере, делалось это с трудом). В-третьих, применение тяжеловесных компонентов влекло за собой появление новых ограничений. К примеру, тяжеловесный компонент всегда имеет прямоугольные очертания и является непрозрачным.

Вскоре после появления первоначальной версии Java стало очевидным, что ограничения, присущие AWT, были настолько серьезными, что нужно было найти лучший подход. В итоге появились классы Swing как часть Java Foundation Classes (JFC). В 1997 году они были включены в Java 1.1 в виде отдельной библиотеки. А начиная с версии Java 1.2, классы Swing (а также все остальное, что входило в состав JFC) стали полностью интегрированными в Java.

Классы Swing построены на основе AWT

Прежде чем продолжить, необходимо сделать одно важное замечание: хотя Swing снимает некоторое количество ограничений, оставшихся от AWT, Swing *не заменяет* AWT. Наоборот, классы Swing построены на основе AWT. Вот почему AWT до сих пор является важной частью Java. Кроме того, Swing использует тот же механизм обработки событий, что и AWT. Таким образом, перед использованием Swing нужно разобраться с тем, как работает AWT. (Об AWT речь велась в главах 23 и 24. Обработке событий была посвящена глава 22.)

Две ключевых особенности Swing

Как только что было сказано, классы Swing были созданы для того, чтобы устранить ограничения, присущие AWT. Этого удалось достичь благодаря двум ключевым особенностям: облегченным компонентам и подключаемому внешнему виду. Вместе они предлагают элегантное и простое в использовании решение проблем AWT. Именно эти две особенности и определяют суть Swing. О каждой из них мы поговорим прямо сейчас.

Компоненты Swing являются облегченными

За некоторыми исключениями, компоненты Swing являются *облегченными*. Это означает, что они написаны исключительно на Java и не преобразуются в равноправные компоненты, специфические для данной платформы. Поскольку облегченные компоненты визуализируются с использованием графических примитивов, они могут быть прозрачными, что позволяет получать непрямоугольные формы. Следовательно, облегченные компоненты являются более эффективными и гибкими. Более того, поскольку облегченные компоненты не преобразуются в равноправные "родные" компоненты, внешний вид каждого компонента определяется классами Swing, а не базовой операционной системой. Это означает, что каждый компонент будет работать одинаково на всех платформах.

Swing поддерживает подключаемый внешний вид

Swing поддерживает принцип *подключаемого внешнего вида* (pluggable look and feel — PLAF). Поскольку каждый компонент Swing визуализируется кодом Java, а не "родными" равноправными компонентами, внешний вид компонента находится под контролем Swing. Это говорит о том, что внешний вид можно отделить от логики компонента, что и делается в Swing. В этом есть и свое преимущество: так можно изменить способ визуализации компонента, не затрагивая какой-либо из его остальных аспектов. Другими словами, можно "подключить" новый внешний вид любого заданного компонента, не порождая при этом каких-либо побочных эффектов в коде, использующем данный компонент. Более того, можно определить целые наборы внешних видов, которые будут представлять разные стили пользовательского графического интерфейса. Чтобы использовать определенный стиль, нужно просто "подключить" его внешний вид. Как только это будет сделано, все компоненты автоматически будут визуализироваться с помощью этого стиля.

Возможность подключаемого внешнего вида имеет несколько важных преимуществ. Например, вы можете задать параметры, которые будут одинаковыми для всех платформ. И наоборот, можно задать внешний вид, специфичный для определенной платформы. Например, если вы знаете, что приложение будет работать только в среде Windows, можно определить внешний вид для Windows. Можно также разработать и специальный внешний вид. И, наконец, внешний вид можно динамически изменять во время работы приложения.

Java SE 6 предлагает такие внешние виды, как metal и Motif, которые доступны всем пользователям Swing. Внешний вид metal (металлический) также называется *внешним видом Java*. Он не зависит от платформы и доступен во всех средах, в которых работает Java. Кроме того, он еще и выбирается по умолчанию. В среде Windows можно использовать внешние виды Windows и Windows Classic. В этой книге используется внешний вид metal, поскольку он не зависит от платформы.

MVC

В общем случае визуальный компонент определяется тремя отдельными аспектами:

- как компонент выглядит во время его визуализации на экране;
- как компонент взаимодействует с пользователем;
- информацией о состоянии компонента.

Независимо от того, какая архитектура используется для реализации компонента, она должна неявно включать эти три аспекта. Вот уже несколько лет свою исключительную эффективность доказала архитектура *"модель-представление-контроллер"* (Model-View-Controller — MVC).

Успех архитектуры MVC объясняется тем, что каждый элемент дизайна соответствует некоторому аспекту компонента. Исходя из терминологии MVC, *модель* соответствует информации о состоянии, связанной с компонентом. Например, в случае флажка модель содержит поле, которое показывает, отмечен ли флажок. *Представление* определяет, как компонент отображается на экране, включая любые аспекты представления, зависящие от текущего состояния модели. *Контроллер* определяет, как компонент будет реагировать на действия пользователя. Например, когда пользователь щелкает на флажке, контроллер реагирует изменением модели, чтобы отразить выбор пользователя (отметка флажка или снятие отметки). Это впоследствии приводит к обновлению представления. Разделяя компонент на модель, представление и контроллер, можно изменять конкретную реализа-

цию любого из этих аспектов, не затрагивая остальные. Например, различные реализации представлений могут визуализировать один и тот же компонент разными способами, однако это не будет влиять на модель или контроллер.

Хотя архитектура MVC и принципы, заложенные в ее основе, звучат концептуально, высокий уровень разделения между представлением и контроллером не дает никаких преимуществ компонентам Swing. Наоборот, Swing использует модифицированную версию MVC, которая объединяет представление и контроллер в один логический объект, называемый *делегатом пользовательского интерфейса* (UI delegate). В связи с этим подход, применяемый в Swing, называется или архитектурой *"модель-делегат"* (Model-Delegate), или архитектурой *"разделяемая модель"* (Separable Model). Поэтому, несмотря на то, что архитектура компонентов Swing основана на MVC, она не использует ее классическую реализацию.

Подключаемый внешний вид в Swing возможен благодаря архитектуре "модель-делегат". Поскольку представление и контроллер отделены от модели, внешний вид можно изменять, не влияя на способ использования компонента внутри программы. И наоборот, можно настроить модель, не влияя на способ отображения компонента на экране или реакцию на действия пользователя.

Для поддержания архитектуры "модель-делегат" большинство компонентов Swing содержат два объекта. Один из них представляет модель, а другой — делегата пользовательского интерфейса. Модели определяются интерфейсами. Например, модель кнопки определяется интерфейсом `ButtonModel`. Делегаты пользовательского интерфейса являются классами, унаследованными от `ComponentUI`. Например, делегатом пользовательского интерфейса кнопки является `ButtonUI`. Как правило, ваши программы не будут взаимодействовать напрямую с делегатами пользовательского интерфейса.

Компоненты и контейнеры

Графический пользовательский интерфейс Swing состоит из двух ключевых элементов: *компонентов* и *контейнеров*. Однако это в основном концептуальное разделение, так как все контейнеры тоже являются компонентами. Различие между этими двумя элементами кроется в их предназначении: компонент представляет собой независимый визуальный элемент управления вроде кнопки или ползунка, а контейнер вмещает группу компонентов. Таким образом, контейнер является особым типом компонента и предназначен для содержания других компонентов. Более того, чтобы можно было отобразить компонент, он должен находиться внутри контейнера. Так, во всех пользовательских графических интерфейсах Swing имеется как минимум один контейнер. Поскольку контейнеры являются компонентами, контейнер может содержать другие контейнеры. Благодаря этому принципу Swing может определить *иерархию вместимости*, на вершине которой должен находиться *контейнер верхнего уровня*.

А теперь давайте поближе познакомимся с компонентами и контейнерами.

Компоненты

В общем случае компоненты Swing происходят от класса `JComponent`. (Исключением является четыре контейнера верхнего уровня, о которых речь пойдет в следующем разделе.) `JComponent` предлагает функциональные возможности, общие для всех компонентов. Например, `JComponent` поддерживает подключаемый внешний вид. `JComponent` наследует AWT-классы `Container` и `Component`. Следовательно, компонент Swing основан на компоненте AWT и совместим с ним.

Все компоненты Swing представлены классами, определенными в пакете `javax.swing`. Ниже перечислены имена классов компонентов Swing (включая компоненты, используемые в качестве контейнеров).

`JApplet`	`JButton`	`JCheckBox`	`JCheckBoxMenuItem`
`JColorChooser`	`JComboBox`	`JComponent`	`JDesktopPane`
`JDialog`	`JEditorPane`	`JFileChooser`	`JFormattedTextField`
`JFrame`	`JInternalFrame`	`JLabel`	`JLayeredPane`
`JList`	`JMenu`	`JMenuBar`	`JMenuItem`
`JOptionPane`	`JPanel`	`JPasswordField`	`JPopupMenu`
`JProgressBar`	`JRadioButton`	`JRadioButtonMenuItem`	`JRootPane`
`JScrollBar`	`JScrollPane`	`JSeparator`	`JSlider`
`JSpinner`	`JSplitPane`	`JTabbedPane`	`JTable`
`JTextArea`	`JTextField`	`JTextPane`	`JTogglebutton`
`JToolBar`	`JToolTip`	`JTree`	`JViewport`
`JWindow`			

Обратите внимание, что все классы компонентов начинаются с буквы `J`. Например, классом для метки является `JLabel`, классом для кнопки — `JButton`, классом для ползунка — `JScrollBar`.

Контейнеры

В Swing определены два типа контейнеров. Первый тип — это контейнеры верхнего уровня. К ним относятся контейнеры `JFrame`, `JApplet`, `JWindow` и `JDialog`. Эти контейнеры не являются наследниками класса `JComponent`. Однако они наследуют AWT-классы `Component` и `Container`. В отличие от остальных компонентов Swing, которые являются облегченными, компоненты верхнего уровня являются тяжеловесными. Поэтому в библиотеке компонентов Swing они представляют собой частный случай.

Судя по названию, контейнер верхнего уровня должен находиться на вершине иерархии контейнеров. Контейнер верхнего уровня не содержится ни в каком другом контейнере. Более того, каждая иерархия вместимости должна начинаться с контейнера верхнего уровня. Таким контейнером в приложениях чаще всего является `JFrame`. В аплетах чаще всего используется `JApplet`.

Вторым типом контейнеров, которые поддерживает Swing, являются облегченные контейнеры. Они наследуются от `JComponent`. Примером облегченного контейнера является `JPanel`, который является контейнером общего назначения. Облегченные контейнеры часто используются для организации и управления группами связанных компонентов, поскольку облегченный контейнер может находиться внутри другого контейнера. Следовательно, вы можете применять облегченные контейнеры наподобие `JPanel` для создания подгрупп связанных элементов управления, содержащихся внутри внешнего контейнера.

Панели контейнеров верхнего уровня

Каждый контейнер верхнего уровня определяет набор *панелей* (pane). Вверху иерархии находится экземпляр `JRootPane`. `JRootPane` — это облегченный контейнер, предназначенный для управления остальными панелями. Он также помогает управлять необязательной линейкой меню. Панели, содержащие корневую панель, называются

“стеклянной” панелью (glass pane), *панелью содержимого* (content pane) и *многослойной панелью* (layered pane).

“Стеклянная” панель является панелью верхнего уровня. Она находится над всеми панелями и покрывает каждую из них. По умолчанию эта панель является прозрачным экземпляром `JPanel`. “Стеклянная” панель позволяет управлять событиями мыши, влияющими на весь контейнер (а не на отдельный элемент управления), или, к примеру, рисовать поверх любого другого компонента. В большинстве случаев вам не нужно будет использовать “стеклянную” панель напрямую, однако если она вам понадобится, то искать ее нужно именно здесь.

Многослойная панель является экземпляром `JLayeredPane`. Многослойная панель позволяет задать определенную глубину размещения компонентов. Значение, соответствующее этой глубине, определяет, какой компонент перекрывает собой другой компонент. (В связи с этим многослойные панели дают возможность задавать упорядоченность компонентов по Z-координате, хотя это не всегда бывает необходимо.) Многослойная панель содержит панель содержимого и (необязательно) линейку меню.

Несмотря на то что “стеклянная” и многослойная панели являются неотъемлемыми частями в операции контейнера верхнего уровня и служат для разных целей, большая часть того, что они предлагают, скрыта от пользователей. Ваше приложение будет работать чаще всего с панелью содержимого, поскольку именно на нее вы будете добавлять визуальные компоненты. Другими словами, когда вы добавляете компонент (например, кнопку) в контейнер верхнего уровня, вы добавляете его в панель содержимого. По умолчанию панель содержимого является непрозрачным экземпляром `JPanel`.

Пакеты Swing

Swing — это очень большая подсистема, в которой задействовано большое количество пакетов. Ниже перечислены пакеты, используемые в Swing, которые определены в версии Java SE 6.

`javax.swing`	`javax.swing.border`	`javax.swing.colorchooser`
`javax.swing.event`	`javax.swing.filechooser`	`javax.swing.plaf`
`javax.swing.plaf.basic`	`javax.swing.plaf.metal`	`javax.swing.plaf.multi`
`javax.swing.plaf.synth`	`javax.swing.table`	`javax.swing.text`
`javax.swing.text.html`	`javax.swing.text.html.parser`	`javax.swing.text.rtf`
`javax.swing.tree`	`javax.swing.undo`	

Самым главным пакетом является `javax.swing`. Его нужно импортировать в любую программу, использующую Swing. В этом пакете содержатся классы, реализующие базовые компоненты Swing, такие как кнопки, метки и флажки.

Простое Swing-приложение

Swing-программы отличаются и от консольных программ, и от AWT-программ, показанных в этой книге. Например, они используют другие наборы компонентов и другие иерархии контейнеров, чем AWT. Swing-программы имеют также особые требования, которые относятся к потоковой обработке. Самый лучший способ понять структуру Swing-программы — испытать в деле рабочий пример. Существуют два типа Java-программ, в которых обычно используется Swing. Первый тип — это настольные приложения, а вто-

рой тип — аплеты. В этом разделе будет рассмотрен пример создания Swing-приложения. Созданием Swing-аплета мы тоже будем заниматься в этой главе, но чуть позже.

Несмотря на то что следующая программа довольно небольшая, она демонстрирует один из способов написания Swing-приложения. Плюс ко всему, она иллюстрирует несколько ключевых возможностей Swing. В ней используется два Swing-компонента: `JFrame` и `JLabel`. `JFrame` — контейнер верхнего уровня, который обычно используется в Swing-приложениях. `JLabel` — Swing-компонент, создающий метку, которая является компонентом, отображающим информацию. Метка — это самый простой Swing-компонент, поскольку она является пассивным компонентом. То есть, метка не реагирует на действия пользователя. Она служит для отображения выходных данных. Программа использует контейнер `JFrame` для хранения экземпляра `JLabel`. Метка отображает короткое текстовое сообщение.

```
// Простое Swing-приложение.

import javax.swing.*;

class SwingDemo {

  SwingDemo() {

    // Создание нового контейнера JFrame.
    JFrame jfrm = new JFrame("A Simple Swing Application");
    // JFrame jfrm = new JFrame("Простое Swing-приложение");

    // Задаем фрейму исходный размер.
    jfrm.setSize(275, 100);

    // Прекращаем работу программы, если пользователь закрывает приложение.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Создаем метку с текстом.
    JLabel jlab = new JLabel("Swing means powerful GUIs.");
    // JLabel jlab = new JLabel("Swing означает мощный GUI.");

    // Добавляем метку на панель содержимого.
    jfrm.add(jlab);

    // Отображаем фрейм.
    jfrm.setVisible(true);
  }

  public static void main(String args[]) {
    // Создаем фрейм в потоке диспетчеризации событий.
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new SwingDemo();
      }
    });
  }
}
```

Swing-программы компилируются и выполняются точно таким же способом, как и остальные Java-приложения. Поэтому чтобы скомпилировать эту программу, можно воспользоваться следующей командной строкой:

```
javac SwingDemo.java
```

Чтобы запустить программу, нужно выполнить следующую команду:

```
java SwingDemo
```

Когда программа начнет работу, она создаст окно, показанное на рис. 29.1.

Рис. 29.1. Окно, созданное программой `SwingDemo`

Поскольку программа `SwingDemo` иллюстрирует несколько основополагающих концепций Swing, мы постараемся разобраться с ней тщательно, строка за строкой. Программа начинается с импорта `javax.swing`. Как уже упоминалось, этот пакет содержит компоненты, определяемые Swing. Например, `javax.swing` определяет классы, реализующие метки, кнопки, текстовые элементы управления и меню. Он будет включен во все программы, использующие Swing.

Затем объявляется класс `SwingDemo` и конструктор для этого класса. Конструктор — это объект, в котором выполняется большинство действий программы. Он начинается с создания `JFrame` с помощью следующей строки кода:

```
JFrame jfrm = new JFrame("A Simple Swing Application");
```

В результате будет создан контейнер `jfrm`, определяющий прямоугольное окно со строкой заголовка, кнопками закрытия, сворачивания, разворачивания и восстановления, а также с системным меню. Таким образом, программа создает стандартное окно верхнего уровня. Конструктору передается заголовок окна.

Затем задаются размеры окна с помощью следующего оператора:

```
jfrm.setSize(275, 100);
```

Метод `setSize()` (он наследуется классом `JFrame` из класса AWT `Component`) задает размеры окна, определяемые в пикселях. Он имеет такую форму:

```
void setSize(int width, int height)
```

В этом примере ширина окна (*`width`*) равняется 275 пикселям, а высота (*`height`*) — 100.

По умолчанию, когда закрывается окно верхнего уровня (например, когда пользователь щелкает на кнопке закрытия), окно удаляется из экрана, но работа приложения не прекращается. Несмотря на то что это поведение в некоторых ситуациях является полезным, для большинства приложений оно не подходит. Чаще всего при закрытии окна верхнего уровня нужно будет просто прекращать работу всего приложения. Это можно сделать двумя способами. Самый простой из них — вызов метода `setDefaultCloseOperation()`, что и делается в программе:

```
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Теперь работа всего приложения будет прекращаться при закрытии окна. Общая форма метода `setDefaultCloseOperation()` выглядит так:

```
void setDefaultCloseOperation(int what)
```

Значение *`what`* определяет, что происходит при закрытии окна. Помимо `JFrame.EXIT_On_CLOSE` доступно несколько других опций:

```
JFrame.DISPOSE_ON_CLOSE
JFrame.HIDE_ON_CLOSE
JFrame.DO_NOTHING_ON_CLOSE
```

В их именах отражены выполняемые действия. Эти константы объявлены в классе `WindowConstants`, который является интерфейсом, объявленным в пакете `javax.swing`, реализуемым контейнером `JFrame`:

```
JLabel jlab = new JLabel("Swing means powerful GUIs.");
```

`JLabel` — самый простой и легкий в использовании компонент, так как он не принимает от пользователя входных данных. Он просто отображает информацию, которая может представлять текст, значок или являться их комбинацией. Метка, созданная программой, содержит только текст, который передается ее конструктору.

Следующая строка кода добавляет метку в панель содержимого фрейма:

```
jfrm.add(jlab);
```

Как было сказано ранее, все контейнеры верхнего уровня имеют панель содержимого, в которой размещаются компоненты. Таким образом, чтобы добавить компонент во фрейм, нужно добавить его в панель содержимого фрейма. Это можно сделать, вызвав метод `add()` для ссылки на `JFrame` (в данном случае `jfrm`). Общая форма метода `add()` показана ниже:

```
Component add(Component comp)
```

Метод `add()` является наследником `JFrame` из AWT-класса `Container`.

По умолчанию панель содержимого, связанная с `JFrame`, использует граничную компоновку. Только что показанный вариант метода `add()` добавляет метку и помещает ее в центре. Другие варианты метода `add()` позволяют задать одну из граничных областей. Когда компонент добавляется в центр, его размер подгоняется автоматически таким образом, чтобы компонент смог уместиться в центре.

Прежде чем продолжить, нужно сделать одно важное замечание. До выхода JDK 5, когда компонент добавляли в панель содержимого, нельзя было вызывать метод `add()` непосредственно на экземпляре `JFrame`. Вместо этого нужно было вызывать метод `add()` для панели содержимого объекта `JFrame`. Панель содержимого можно было получить путем вызова метода `getContentPane()` на экземпляре `JFrame`. Метод `getContentPane()` показан ниже:

```
Container getContentPane( )
```

`Container` получает ссылку на окно содержимого. После этого производился вызов метода `add()` по этой ссылке для добавления компонента в панель содержимого. Таким образом, чтобы добавить `jlab` в `jfrm`, раньше нужно было использовать следующий оператор:

```
jfrm.getContentPane().add(jlab); // старый стиль
```

Здесь метод `getContentPane()` сначала получает ссылку на панель содержимого, после чего метод `add()` добавляет компонент в контейнер, присоединенный к этому окну. Эту же процедуру нужно было выполнять для вызова метода `remove()`, когда требовалось удалить компонент, и метода `setLayout()`, чтобы задать диспетчер компоновки для окна содержимого. В коде, написанном на Java до версии 5.0, вам будут часто попадаться вызовы метода `getContentPane()`. Однако теперь использовать этот метод больше не нужно. Можно просто вызывать методы `add()`, `remove()` и `setLayout()` непосредст-

венно на JFrame, так как они были изменены специально для того, чтобы автоматически работать с окном содержимого.

Последний оператор в конструкторе SwingDemo нужен для того, чтобы сделать окно видимым:

```
jfrm.setVisible(true);
```

Метод setVisible() является наследником AWT-класса Component. Если его аргумент будет равен true, то окно будет отображаться. В противном случае оно будет скрыто. По умолчанию JFrame является невидимым, поэтому чтобы показать его, нужно вызвать метод setVisible(true).

Внутри метода main() создается объект SwingDemo, который отображает окно и метку. Обратите внимание, что конструктор SwingDemo вызывается с помощью следующих трех строк кода:

```
SwingUtilities.invokeLater(new Runnable() {
  public void run() {
    new SwingDemo();
  }
});
```

Выполнение этой последовательности приводит к созданию объекта SwingDemo в *потоке диспетчеризации событий*, а не в главном потоке приложения. И вот почему. В общем случае Swing-программы управляются событиями. Например, когда пользователь взаимодействует с компонентом, генерируется событие. Событие передается в приложение путем вызова обработчика событий, определенного в приложении. Однако обработчик выполняется в потоке диспетчеризации событий, поддерживаемом Swing, а не в главном потоке приложения. Таким образом, хотя обработчики событий и определены в программе, они вызываются в потоке, который не был создан вашей программой.

Чтобы избежать возникновения этой проблемы (включая вероятность возникновения взаимной блокировки), все GUI-компоненты Swing нужно создавать и обновлять из потока диспетчеризации событий, а не из главного потока приложения. А метод add() выполняется в главном потоке. Таким образом, метод main() не может напрямую наследовать объект SwingDemo. Наоборот, он должен создать объект Runnable, который выполняется в потоке диспетчеризации событий, и заставить этот объект создать GUI.

Чтобы код GUI можно было создать в потоке диспетчеризации событий, необходимо использовать один из двух методов, определенных в классе SwingUtilities. Это методы invokeLater() и invokeAndWait():

```
static void invokeLater(Runnable obj)
static void invokeAndWait(Runnable obj)
  throws InterruptedException, InvocationTargetException
```

Здесь *obj* — это объект Runnable, метод run() которого будет вызываться потоком диспетчеризации событий. Единственное различие между этими двумя методами заключается в том, что метод invokeLater() возвращает результат немедленно, а метод invokeAndWait() ожидает возврата результата obj.run(). Вы можете использовать эти методы для вызова метода, создающего GUI для вашего Swing-приложения, или использовать их каждый раз, когда вам нужно будет изменить состояние GUI из кода, не выполняющегося в потоке диспетчеризации событий. Как правило, вам нужно будет использовать метод invokeLater(), как это было в предыдущей программе. А при создании исходного GUI для аплета вам понадобится метод invokeAndWait().

Обработка событий

В предыдущем примере была показана базовая форма Swing-программы, однако в ней не хватает одной важной части: обработки событий. Поскольку метка `JLabel` не принимает входных данных от пользователя, она не генерирует события, поэтому и обработка событий не была нужна. Однако остальные Swing-компоненты *реагируют* на вводимые пользователем данные, вследствие чего возникает необходимость в обработке событий, которые возникают в результате таких взаимодействий. Например, событие генерируется тогда, когда таймер завершает отсчет. В любом случае, обработка событий является большой частью любого приложения, построенного на основе Swing.

Механизм обработки событий, используемый в Swing, ничем не отличается от механизма, применяемого в AWT. Этот подход называется *моделью делегации событий*, и он рассматривался в главе 22. Во многих случаях Swing использует те же события, что и AWT, и эти события определены в пакете `java.awt.event`. События, являющиеся специфическими для Swing, определены в пакете `javax.swing.event`.

Несмотря на то что события обрабатываются в Swing точно так же, как и в AWT, будет полезно рассмотреть небольшой и простой пример. В следующей программе обрабатывается событие, сгенерированное кнопкой из класса Swing. Результат показан на рис. 29.2.

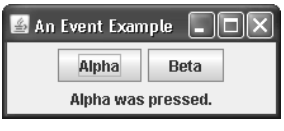

Рис. 29.2. Результат выполнения программы `EventDemo`

```
// Обработка события в Swing-программе.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class EventDemo {

  JLabel jlab;

  EventDemo() {

    // Создание нового контейнера JFrame.
    JFrame jfrm = new JFrame("An Event Example");
    // JFrame jfrm = new JFrame("Пример обработки событий");

    // Определение класса FlowLayout для диспетчера компоновки.
    jfrm.setLayout(new FlowLayout());

    // Установка исходных размеров фрейма.
    jfrm.setSize(220, 90);

    // Прекращение работы программы, если пользователь закрывает приложение.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Создаем две кнопки.
    JButton jbtnAlpha = new JButton("Alpha");
    JButton jbtnBeta = new JButton("Beta");

    // Добавляем слушатель действий для Alpha.
    jbtnAlpha.addActionListener(new ActionListener() {
```

```
      public void actionPerformed(ActionEvent ae) {
        jlab.setText("Alpha was pressed.");
        // jlab.setText("Нажата кнопка Alpha.");
      }
    });

    // Добавляем слушатель действий для Beta.
    jbtnBeta.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ae) {
        jlab.setText("Beta was pressed.");
        // jlab.setText("Нажата кнопка Beta.");
      }
    });

    // Добавляем кнопки в панель содержимого.
    jfrm.add(jbtnAlpha);
    jfrm.add(jbtnBeta);

    // Создаем текстовую метку.
    jlab = new JLabel("Press a button.");
    // jlab = new JLabel("Нажмите кнопку.");

    // Добавляем метку в панель содержимого.
    jfrm.add(jlab);

    // Отображаем фрейм.
    jfrm.setVisible(true);
  }

  public static void main(String args[]) {
    // Создаем фрейм в потоке диспетчеризации событий.
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new EventDemo();
      }
    });
  }
}
```

Для начала обратите внимание на то, что программа теперь импортирует пакеты `java.awt` и `java.awt.event`. Пакет `java.awt` нужен по той причине, что в нем содержится класс `FlowLayout`, поддерживающий стандартный диспетчер компоновки потоков, который используется для размещения компонентов во фрейме. (Диспетчеры компоновки рассматривались в главе 24.) Пакет `java.awt.event` необходим потому, что он определяет интерфейс `ActionListener` и класс `ActionEvent`.

Конструктор `EventDemo` начинает работу с создания контейнера `JFrame` по имени `jfrm`. Затем он устанавливает диспетчер компоновки для панели содержимого `jfrm` в `FlowLayout`. Вспомните, что по умолчанию панель содержимого использует диспетчер компоновки `BorderLayout`. Однако для данного примера удобнее применить именно `Flowlayout`. Обратите внимание на то, что `FlowLayout` назначается с помощью следующего оператора:

```
jfrm.setLayout(new FlowLayout());
```

Как уже говорилось, раньше приходилось явным образом вызывать метод `getContentPane()`, чтобы задать диспетчер компоновки для окна содержимого. После выхода JDK 5 этого делать не нужно.

После определения размеров и стандартной операции при закрытии метод EventDemo() создает две кнопки, как показано ниже:

```
JButton jbtnAlpha = new JButton("Alpha");
JButton jbtnBeta = new JButton("Beta");
```

Первая кнопка будет содержать текст "Alpha", а вторая — "Beta". Кнопки класса Swing являются экземплярами JButton. JButton предлагает несколько конструкторов. Одним из используемых здесь конструкторов является следующий:

```
JButton(String msg)
```

Параметр *msg* определяет строку, которая будет отображаться внутри кнопки.

При щелчке на кнопке генерируется событие ActionEvent. Таким образом, JButton предлагает метод addActionListener(), который используется для добавления слушателя событий. (JButton предлагает также метод removeActionListener() для удаления слушателя, однако он в этой программе не используется.) Как было сказано в главе 22, интерфейс ActionListener определяет только один метод: actionPerformed(). Чтобы вам было легче его вспомнить, мы приведем его еще раз:

```
void actionPerformed(ActionEvent ae)
```

Этот метод вызывается в результате щелчка на кнопке. Другими словами, это обработчик события, который вызывается в случае возникновения события при щелчке на кнопке.

После этого показанный ниже код добавляет слушатели событий для событий действий с кнопками:

```
// Добавляем слушатель действий для кнопки Alpha.
jbtnAlpha.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent ae) {
    jlab.setText("Alpha was pressed.");
  }
});

// Добавляем слушатель действий для кнопки Beta.
jbtnBeta.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent ae) {
    jlab.setText("Beta was pressed.");
  }
});
```

Здесь анонимные внутренние классы используются для того, чтобы предоставить обработчики событий двум кнопкам. Всякий раз при щелчке на кнопке строка, отображенная в jlab, изменяется в зависимости от того, на какой кнопке был произведен щелчок.

Затем кнопки добавляются в панель содержимого:

```
jfrm.add(jbtnAlpha);
jfrm.add(jbtnBeta);
```

И, наконец, в панель содержимого добавляется jlab, и окно становится видимым. Когда вы запустите программу, то при каждом щелчке на кнопке в метке будет отображаться сообщение, показывающее, какая кнопка была нажата.

И еще одно замечание: следует помнить о том, что все обработчики событий наподобие actionPerformed() вызываются в потоке диспетчеризации событий. Таким образом, обработчик событий должен быстро дать результат, чтобы не допустить замедления рабо-

ты приложения. Если вашему приложению нужно сделать что-то такое, для чего потребуется много времени, и что будет расцениваться как событие, то оно должно использовать отдельный поток.

Создание Swing-аплета

Вторым типом программы, которая обычно использует классы Swing, является аплет. Аплеты, созданные на основе Swing, подобны аплетам, созданным на основе AWT, но у них есть одно существенное отличие: Swing-аплет расширяет класс `JApplet`, а не `Applet`. Таким образом, `JApplet` включает все функциональные возможности `Applet` и добавляет поддержку Swing. `JApplet` является Swing-контейнером верхнего уровня; это означает, что он не является наследником `JComponent`. Так как `JApplet` является контейнером верхнего уровня, он включает различные панели, описанные ранее. Это означает, что все компоненты добавляются в панель содержимого `JApplet` точно так же, как и компоненты, добавляемые в панель содержимого `JFrame`.

Swing-аплеты используют те же методы обеспечения жизненного цикла, которые были описаны в главе 21: `init()`, `start()`, `stop()` и `destroy()`. Естественно, вам нужно переопределять только те методы, которые будут нужны вашему аплету. Процесс рисования в Swing производится иначе, чем в AWT, и Swing-аплет обычно не переопределяет метод `paint()`. (О рисовании в Swing мы поговорим далее в этой главе.)

Еще один нюанс: все взаимодействия с компонентами в Swing должны производиться в потоке диспетчеризации событий, о чем было сказано в предыдущем разделе. Это относится ко всем Swing-программам.

Ниже представлен пример Swing-аплета. Он снабжен теми же функциями, что и предыдущее приложение, но только является аплетом. На рис. 29.3 показана программа, выполняемая в `appleteviewer`.

Рис. 29.3. Результат работы Swing-аплета

```
// Простой аплет, основанный на Swing

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

/*
Этот HTML-код можно использовать для запуска аплета:

<object code="MySwingApplet" width=220 height=90>
</object>
*/
```

```
public class MySwingApplet extends JApplet {
  JButton jbtnAlpha;
  JButton jbtnBeta;

  JLabel jlab;

  // Инициализация аплета.
  public void init() {
    try {
      SwingUtilities.invokeAndWait(new Runnable () {
        public void run() {
          makeGUI(); // инициализация GUI
        }
      });
    } catch(Exception exc) {
      System.out.println("Can't create because of "+ exc);
      // System.out.println("Невозможно создать из-за "+ exc);
    }
  }

  // Этому аплету не нужно переопределять методы start(), stop()
  // или destroy().

  // Настройка и инициализация GUI.
  private void makeGUI() {

  // Настройка аплета для использования компоновки потоков.
  setLayout(new FlowLayout());

  // Создание двух кнопок.
  jbtnAlpha = new JButton("Alpha");
  jbtnBeta = new JButton("Beta");

  // Добавление слушателя действий для кнопки Alpha.
  jbtnAlpha.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
      jlab.setText("Alpha was pressed.");
      // jlab.setText("Нажата кнопка Alpha.");
    }
  });

  // Добавление слушателя действий для кнопки Beta.
  jbtnBeta.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent le) {
      jlab.setText("Beta was pressed.");
      // jlab.setText("Нажата кнопка Beta.");
    }
  });

  // Добавление кнопок в панель содержимого.
  add(jbtnAlpha);
  add(jbtnBeta);

  // Создание текстовой метки.
  jlab = new JLabel("Press a button.");
  // jlab = new JLabel("Нажмите кнопку.");

  // Добавление метки в панель содержимого.
  add(jlab);
  }
}
```

Необходимо сделать пару важных замечаний касательно аплетов. Во-первых, `MySwingApplet` расширяет `JApplet`. Как мы уже говорили, все аплеты, основанные на Swing, расширяют класс `JApplet`, а не `Applet`. Во-вторых, метод `init()` инициализирует Swing-компоненты в потоке диспетчеризации событий, устанавливая вызов метода `makeGUI()`. Обратите внимание, что это достигается путем использования метода `invokeAndWait()`, а не `invokeLater()`. Аплеты должны использовать метод `invokeAndWait()` потому, что метод `init()` не должен возвращать результат до тех пор, пока не будет выполнен весь процесс инициализации. По сути, метод `start()` нельзя вызывать прежде, чем закончится инициализация; это означает, что нужно полностью создать GUI.

Внутри метода `makeGUI()` создаются две кнопки и метка, а к кнопкам добавляются слушатели действий. Наконец, компоненты добавляются в панель содержимого. Несмотря на то что пример является довольно-таки простым, этот подход нужно применять при создании любого GUI Swing, который будет использован аплетом.

Рисование в Swing

Несмотря на то что компоненты Swing очень функциональны, вы не ограничены только их использованием, поскольку Swing позволяет также выводить информацию непосредственно в область отображения фрейма, панели или одного из компонентов Swing, таких как `JLabel`. Хотя во многих (возможно, в большинстве) случаях использования Swing не производится рисование прямо на поверхности компонента, это можно делать в приложениях, где подобное необходимо. Чтобы вывести данные прямо на поверхность компонента, нужно использовать один или несколько методов рисования, определенных в AWT, вроде `drawLine()` или `drawRect()`. Таким образом, большинство технологий и методов, описанных в главе 23, можно применять и к Swing. С другой стороны, между ними есть очень важные отличия, поэтому обо всем этом мы и поговорим подробно в текущем разделе.

Основы рисования

Подход к рисованию в Swing базируется на оригинальном механизме, построенном на основе AWT, однако Swing позволяет более качественно управлять этим процессом. Прежде чем приступить к изучению специфики рисования в Swing, будет полезно пересмотреть механизм рисования в AWT.

AWT-класс `Component` предлагает метод `paint()`, который используется для рисования выходных данных прямо на поверхности компонента. Как правило, метод `paint()` не вызывается программой. (В действительности, вызывать этот метод написанной вами программой придется в очень редких случаях.) Вместо этого метод `paint()` вызывается системой времени выполнения в процессе визуализации компонента. Такая ситуация может возникнуть в силу ряда причин. Например, окно, в котором отображается компонент, может быть перекрыто другим окном, а затем вновь появиться на экране. Или же окно может быть свернуто, а затем восстановлено. Метод `paint()` вызывается также в тех случаях, когда программа начинает свою работу. При написании кода, основанного на AWT, приложение будет переопределять метод `paint()`, когда ему будет необходимо вывести данные прямо на поверхности компонента.

Поскольку класс `JComponent` унаследован от `Component`, все облегченные компоненты Swing получают метод `paint()`. Однако вам *не* придется переопределять его, чтобы выводить информацию непосредственно на поверхности компонента. Дело в том, что при

рисовании Swing использует более изощренный подход, который включает три различных метода: `paintComponent()`, `paintBorder()` и `paintChildren()`. Эти методы рисуют заданную часть компонента и делят процесс рисования на три разных логических действия. В облегченном компоненте исходный метод AWT `paint()` просто выполняет вызовы этих методов, в показанном только что порядке.

Чтобы нарисовать поверхность компонента Swing, вы должны будете создать подкласс компонента, а затем переопределить его метод `paintComponent()`. Этот метод отвечает за прорисовку внутренней части компонента. Как правило, остальные два метода рисования вы переопределять не будете. Переопределяя метод `paintComponent()`, вы сначала должны вызвать метод `super.paintComponent()`, чтобы задействовать часть суперкласса процесса рисования. (Этого делать не нужно лишь в том случае, если вы вручную управляете способом отображения компонента.) После этого можно вывести данные, которые вы хотите отобразить. Ниже показан метод `paintComponent()`:

```
protected void paintComponent(Graphics g)
```

В параметре *g* указывается графическое содержимое выходных данных.

Чтобы программно нарисовать компонент, нужно вызвать метод `repaint()`. Он работает в Swing точно так же, как в AWT. Метод `repaint()` определен в классе `Component`. Если его вызвать, система вызывает метод `paint()` сразу же, как только представляется для этого возможность. Поскольку процесс рисования отнимает довольно много времени, этот механизм позволяет системе времени выполнения мгновенно задерживать рисование до тех пор, пока, например, не завершится выполнение другой задачи, имеющей более высокий приоритет. Естественно, в Swing вызов метода `paint()` приводит к вызову метода `paintComponent()`. Таким образом, чтобы вывести данные на поверхность компонента, ваша программа будет хранить эти данные до тех пор, пока не будет вызван метод `paintComponent()`. Внутри переопределенного метода `paintComponent()` вы будете рисовать хранимые данные.

Вычисление области рисования

Во время рисования на поверхности компонента нужно тщательно ограничить область рисования выходных данных, находящуюся внутри границ компонента. Хотя Swing автоматически отсекает любые выходные данные, выходящие за границы компонента, может получиться так, что рисование будет выполняться прямо на границе, которая затем будет заменена при перерисовке. Чтобы не допустить этого, вы должны вычислить *область рисования* в пределах компонента. Эта область определяется так: берется текущий размер компонента и из него вычитается пространство, занятое его границами. Таким образом, прежде чем нарисовать компонент, вы должны сначала узнать ширину границы, а затем уже подгонять область рисования.

Чтобы узнать ширину границы, вызовите метод `getInsets()`:

```
Insets getInsets()
```

Этот метод определен в классе `Container` и переопределяется в классе `JComponent`. Он возвращает объект `Insets`, содержащий размеры границ. Значения можно получить с помощью 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 узнать, обратившись к методам `getWidth()` и `getHeight()`. Они показаны ниже:

```
int getWidth()
int getHeight()
```

Вычитая значения границ, можно получить ширину и высоту области, в которой будет выполняться рисование.

Пример рисования

Сейчас мы рассмотрим программу, в которой будут реализованы рассмотренные методы. Она создает класс `PaintPanel`, расширяющий класс `JPanel`. Программа использует объект данного класса для отображения линий, конечные точки которых генерируются случайным образом. Результат выполнения программы показан на рис. 29.4.

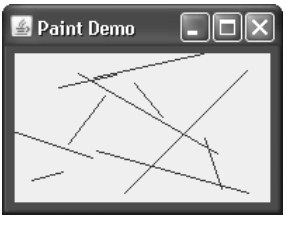

Рис. 29.4. Результат выполнения программы `PaintPanel`

```
// Рисование линий в панели.

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

// Этот класс расширяет класс JPanel. Он переопределяет
// метод paintComponent(), чтобы выводить
// в панели случайные линии.
class PaintPanel extends JPanel {
  Insets ins; // хранит размеры внутренней части панели

  Random rand; // используется для генерирования случайных чисел

  // Создаем панель.
  PaintPanel() {

    // Помещаем рамку вокруг панели (создаем ее границы).
    setBorder(
      BorderFactory.createLineBorder(Color.RED, 5));

    rand = new Random();
  }

  // Переопределяем метод paintComponent().
  protected void paintComponent(Graphics g) {
    // Первым всегда вызывается метод суперкласса.
    super.paintComponent(g);

    int x, y, x2, y2;
```

```
    // Получаем высоту и ширину компонента.
    int height = getHeight();
    int width = getWidth();

    // Получаем размеры внутренней части.
    ins = getInsets();

    // Рисуем десять линий, конечные точки которых
    // генерируются случайным образом.
    for(int i=0; i < 10; i++) {
      // Получаем случайные 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);

      // Рисуем линию.
      g.drawLine(x, y, x2, y2);
    }
  }
}

// Иллюстрация рисования непосредственно в панели.
class PaintDemo {

  JLabel jlab;
  PaintPanel pp;

  PaintDemo() {

   // Создаем новый контейнер JFrame.
   JFrame jfrm = new JFrame("Paint Demo");

   // Присваиваем фрейму исходные размеры.
   jfrm.setSize(200, 150);

   // Прекращаем работу программы, если пользователь
   // закрывает приложение.
   jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

   // Создаем панель, которую будем рисовать.
   pp = new PaintPanel();

   // Добавляем панель на панель содержимого. Так как используется
   // компоновка BorderLayout, размеры панели будут подбираться таким
   // образом, чтобы она заняла центральную часть области.
   jfrm.add(pp);

   // Отображаем фрейм.
   jfrm.setVisible(true);
 }

  public static void main(String args[]) {
    // Создаем фрейм в потоке диспетчеризации событий.
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new PaintDemo();
      }
    });
  }
}
```

А теперь давайте обсудим программу. Класс `PaintPanel` расширяет класс `JPanel`. Класс `JPanel` является одним из облегченных контейнеров Swing; то есть мы получаем компонент, который можно добавлять в панель содержимого `JFrame`. Для обработки процесса рисования `PaintPanel` переопределяет метод `paintComponent()`. Это позволяет классу `PaintPanel` производить рисование прямо на поверхности компонента. Размеры панели не определены, поскольку программа по умолчанию использует компоновку `BorderLayout`, а панель добавляется в центр области. В результате этого панель получает такие размеры, которые позволяют ей уместиться в центре. При изменении размеров окна будут соответствующим образом подгоняться и размеры панели.

Обратите внимание на то, что конструктор также определяет красную рамку (границу) толщиной в 5 пикселей. Для этого используется метод `setBorder()`, показанный ниже:

```
void setBorder(Border border)
```

`Border` — это интерфейс Swing, инкапсулирующий рамку. Рамку можно получить, вызвав один из методов, определенных в классе `BorderFactory`. В этой программе применяется один из таких методов — `createLineBorder()`. Он создает простую линейную рамку:

```
static Border createLineBorder(Color clr, int width)
```

Здесь *`clr`* определяет цвет рамки, а *`width`* — ее ширину в пикселях.

Внутри переопределения метода `paintComponent()` следует обратить внимание на то, что он сначала вызывает метод `super.paintComponent()`. Как уже было сказано, это необходимо для обеспечения надлежащего рисования. Затем производится вычисление ширины и высоты панели, а также внутренней части. Эти значения используются для того, чтобы рисуемые линии находились внутри области рисования панели. Область рисования — это общая ширина и высота компонента минус ширина рамки. Вычисления реализованы так, чтобы можно было оперировать с различными размерами `PaintPanel` и границ. Чтобы убедиться в этом, попробуйте изменить размеры окна. Линии по-прежнему будут рисоваться внутри границ панели.

Класс `PaintDemo` создает класс `PaintPanel`, а затем добавляет панель в панель содержимого. Во время первого отображения приложения вызывается переопределенный метод `paintComponent()` и рисуются линии. Каждый раз, когда вы будете изменять размеры окна или сворачивать и восстанавливать его, будет рисоваться новый набор линий. Во всех классах линии будут находиться внутри заданной области рисования.

30 ГЛАВА

Дополнительные сведения о Swing

В предыдущей главе были рассмотрены некоторые из базовых концепций Swing и показана общая форма приложения и аплета, основанных на Swing. В этой главе мы продолжим обсуждение Swing, предоставляя описание ряда компонентов Swing, таких как кнопки, флажки, деревья и таблицы. Компоненты Swing обладают богатыми функциональными возможностями и предлагают широкие возможности по их настройке. К сожалению, в этой книге невозможно описать все особенности и атрибуты компонентов, поэтому мы рассмотрим лишь некоторые из них.

Классы компонентов Swing, рассматриваемых в этой главе, показаны ниже в виде таблицы:

`JButton`	`JCheckBox`	`JComboBox`	`JLabel`
`JList`	`JRadioButton`	`JScrollPane`	`JTabbedPane`
`JTable`	`JTextField`	`JToggleButton`	`JTree`

Все эти компоненты являются облегченными; это означает, что все они происходят из класса `JComponent`.

В этой главе будет использоваться также класс `ButtonGroup`, который инкапсулирует взаимно исключающий набор кнопок Swing, и класс `ImageIcon`, инкапсулирующий графическое изображение. Каждый из них определен в Swing и включен в пакет `javax.swing`.

И еще один момент. Компоненты Swing демонстрируются в аплетах, поскольку код аплета более компактный, чем код настольного приложения, а к приложениям и аплетам применяются одни и те же технологии.

`JLabel` и `ImageIcon`

Самым простым в использовании компонентом Swing является `JLabel`. Мы рассматривали его в предыдущей главе, и вы уже знаете, что он создает метку. В этой главе мы поговорим о нем более подробно. `JLabel` можно использовать для отображения текста и/или значка (пиктограммы). Этот компонент является пассивным в том смысле, что он

не реагирует на данные, вводимые пользователем. JLabel определяет несколько конструкторов. Ниже показаны три из них:

```
JLabel(Icon icon)
JLabel(String str)
JLabel(String str, Icon icon, int align)
```

Здесь параметры *str* и *icon* представляют текст и значок, которые будут использоваться для метки. Параметр *align* определяет выравнивание текста и/или значка по горизонтали внутри метки. Он должен иметь одно из следующих значений: LEFT, RIGHT, CENTER, LEADING или TRAILING. Наряду с некоторыми другими константами, используемыми в классах Swing, эти константы определены в интерфейсе SwingConstants.

Обратите внимание, что значки определяются с помощью объектов, имеющих тип Icon, который представляет собой интерфейс, определенный в Swing. Получить значок проще всего можно посредством класса ImageIcon. Класс ImageIcon реализует Icon и инкапсулирует изображение. Таким образом, объект, имеющий тип ImageIcon, можно передать с помощью параметра Icon конструктора JLabel. Предоставить изображение можно несколькими способами, включая чтение изображения из файла или его загрузку из места, определенного с помощью URL-адреса. Ниже показан конструктор ImageIcon, используемый в примере этого раздела:

```
ImageIcon(String filename)
```

Он получает изображение из файла, имя которого указано в параметре *filename*.

Значок и текст, связанные с меткой, можно получить с помощью следующих методов:

```
Icon getIcon()
String getText()
```

Значок и текст, связанные с меткой, можно установить с применением таких методов:

```
void setIcon(Icon icon)
void setText(String str)
```

Здесь *icon* и *str* представляют значок и текст, соответственно. Таким образом, с помощью метода setText() можно изменить текст внутри метки во время работы программы.

На примере следующего аплета показано, как создается и отображается метка, содержащая значок и строку. Аплет начинается с того, что он создает объект ImageIcon из файла france.gif, который отображает государственный флаг Франции. Это изображение используется в качестве второго параметра конструктора JLabel. Первый и последний параметры конструктора JLabel представляют текст метки и его выравнивание. В конце аплета метка добавляется в панель содержимого.

```
// Пример JLabel и ImageIcon.
import java.awt.*;
import javax.swing.*;
/*
  <applet code="JLabelDemo" width=250 height=150>
  </applet>
*/

public class JLabelDemo 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);
    // System.out.println("Невозможно создать из-за " + exc);
  }
}

private void makeGUI() {
  // Создание значка.
  ImageIcon ii = new ImageIcon("france.gif");
  // Создание метки.
  JLabel jl = new JLabel("France", ii, JLabel.CENTER);
  // Добавление метки в панель содержимого.
  add(jl);
}
}
```

На рис. 30.1 показана полученная метка.

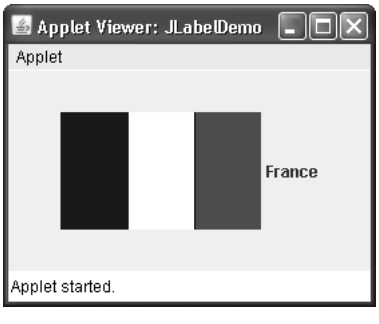

Рис. 30.1. Окно аплета `JLabelDemo`

JTextField

`JTextField` — это простейший текстовый компонент Swing. Более того, это, пожалуй, наиболее широко используемый текстовый компонент. `JTextField` позволяет редактировать одну строку текста. Он происходит из класса `JTextComponent`, который наделяет функциональными возможностями текстовые компоненты Swing. Для своей модели `JTextField` использует интерфейс `Document`.

Ниже показаны три конструктора `JTextField`:

```
JTextField(int cols)
JTextField(String str, int cols)
JTextField(String str)
```

Здесь *`str`* — это первоначально представляемая строка, а *`cols`* — количество столбцов в текстовом поле. Если строка не будет задана, то текстовое поле первоначально будет

пустым. Если не будет задано количество столбцов, то размер текстового поля будет выбран так, чтобы оно могло уместиться в указанной строке.

`JTextField` генерирует события в ответ на действия пользователя. Например, событие `ActionEvent` возникает, когда пользователь нажимает клавишу <Enter>. Событие `CaretEvent` возникает всякий раз, когда изменяется позиция каретки (то есть, курсора). (Событие `CaretEvent` определено в пакете `javax.swing.event`.) Возникать могут и другие события. Как правило, вашей программе не нужно будет обрабатывать эти события. Наоборот, вы будете просто получать строку, находящуюся в данный момент в текстовом поле. Для ее получения необходимо вызвать метод `getText()`.

Ниже показан пример применения `JTextField`. В этом аплете создается объект `JTextField` и добавляется в панель содержимого. Когда пользователь нажимает клавишу <Enter>, генерируется событие действия. В результате этого события отображается текст в окне состояния.

```
// Демонстрация применения 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.invokeAndWait(
        new Runnable() {
          public void run() {
            makeGUI( );
          }
        }
      );
    } catch (Exception exc) {
      System.out.println("Can't create because of " + exc);
      // System.out.println("Невозможно создать из-за " + exc);
    }
  }
  private void makeGUI() {
    // Изменение компоновки потока.
    setLayout(new FlowLayout());
    // Добавление текста в панель содержимого.
    jtf = new JTextField(15);
    add(jtf);
    jtf.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ae) {
        // Отображаем текст, когда пользователь нажимает Enter.
        showStatus(jtf.getText());
      }
    });
  }
}
```

На рис. 30.2 показан пример полученного текстового поля.

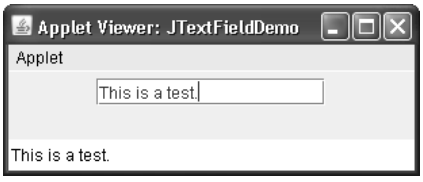

Рис. 30.2. Окно аплета `JTextFieldDemo`

Кнопки Swing

В Swing определены четыре типа кнопок: `JButton`, `JToggleButton`, `JCheckBox` и `JRadioButton`. Все они являются подклассами класса `AbstractButton`, который расширяет класс `JComponent`. Таким образом, у кнопок есть общие характерные черты.

Класс `AbstractButton` содержит множество методов, позволяющих управлять поведением кнопок. Например, вы можете определить различные значки, которые будут отображаться на месте кнопки, когда она будет отключена, нажата или выбрана. Другой значок можно использовать в качестве значка-наката (rollover), который будет отображаться при наведении указателя мыши на кнопку. Ниже показаны методы, 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 различных состояний.

Текст, связанный с кнопкой, можно прочитать и записать посредством следующих методов:

```
String getText()
void setText(String str)
```

Здесь *str* — это текст, который будет связан с кнопкой.

Модель, используемая всеми кнопками, определена интерфейсом `ButtonModel`. Кнопка генерирует событие действия, когда ее нажимает пользователь. Возможны также и другие события. Сейчас мы поговорим о каждом конкретном классе кнопки.

JButton

Класс `JButton` определяет функциональные возможности экранной кнопки. В предыдущей главе вы уже могли с ним вкратце познакомиться. Он позволяет связывать с кнопкой на экране значок, строку или оба этих элемента вместе. Ниже показаны три его конструктора:

```
JButton(Icon icon)
JButton(String str)
JButton(String str, Icon icon)
```

Здесь *icon* и *str* представляют значок и строку, которые используются для кнопки.

Когда пользователь нажимает кнопку (щелкает на ней), возникает событие `ActionEvent`. С помощью объекта `ActionEvent`, переданного методу `actionPerformed()` зарегистри-

рованного слушателя ActionListener, вы можете получить *командную строку действия*, связанную с кнопкой. По умолчанию эта строка будет отображаться внутри кнопки. Однако команду действия можно задать, вызвав метод setActionCommand() в отношении кнопки. Получить команду действия можно, вызвав метод getActionCommand() для объекта события. Он объявляется следующим образом:

```
String getActionCommand()
```

Команда действия идентифицирует кнопку. Таким образом, при использовании двух или более кнопок в одном и том же приложении команда действия дает возможность легко определить, какая кнопка была нажата.

В предыдущей главе был показан пример текстовой кнопки. В следующем примере демонстрируется кнопка со значком. В нем отображаются четыре кнопки и одна метка. Каждая кнопка отображает значок, представляющий флаг государства. Когда пользователь щелкает на кнопке, в метке появляется название государства.

```
// Пример использования кнопки JButton со значком.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
  <applet code="JButtonDemo" width=250 height=450>
  </applet>
*/
public class JButtonDemo 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);
      // System.out.println("Невозможно создать из-за " + exc);
    }
  }
private void makeGUI() {
    // Изменение компоновки потока.
    setLayout(new FlowLayout());
    // Добавление кнопок в панель содержимого.
    ImageIcon france = new ImageIcon("france.gif");
    JButton jb = new JButton(france);
    jb.setActionCommand("France");
    jb.addActionListener(this);
    add(jb);
    ImageIcon germany = new ImageIcon("germany.gif");
    jb = new JButton(germany);
    jb.setActionCommand("Germany");
    jb.addActionListener(this);
    add(jb);
```

```
    ImageIcon italy = new ImageIcon("italy.gif");
    jb = new JButton(italy);
    jb.setActionCommand("Italy");
    jb.addActionListener(this);
    add(jb);

    ImageIcon japan = new ImageIcon("japan.gif");
    jb = new JButton(japan);
    jb.setActionCommand("Japan");
    jb.addActionListener(this);
    add(jb);

    // Создаем и добавляем метку в панель содержимого.
    jlab = new JLabel("Choose a Flag");
    add(jlab);
  }

  // Обработка событий кнопки.
  public void actionPerformed(ActionEvent ae) {
     jlab.setText("You selected " + ae.getActionCommand());
     // jlab.setText("Выбрана " + ae.getActionCommand());
  }
}
```

На рис. 30.3 показан результат выполнения этого аплета.

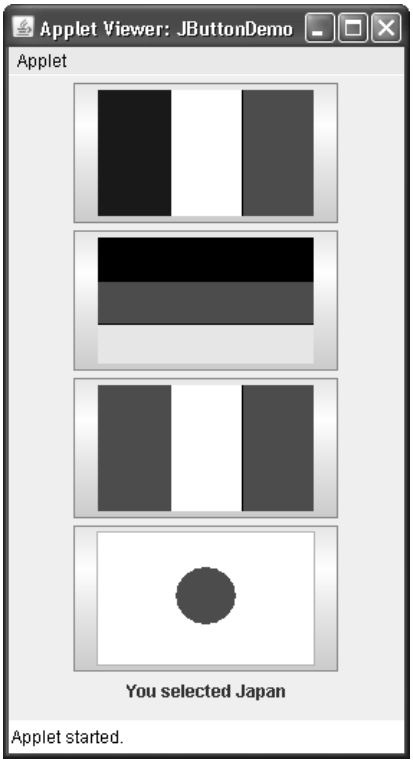

Рис. 30.3. Окно аплета `JButtonDemo`

JToggleButton

Полезной разновидностью кнопки является *тумблер* (toggle button). Тумблер выглядит подобно обычной кнопке, но действует по-другому, так как у него может быть два состояния: нажатое и отжатое. То есть, когда вы нажимаете тумблер, он остается нажатым, а не отжимается подобно обычной кнопке. Если после этого нажать тумблер еще раз, он будет отжат. Таким образом, каждый раз, когда пользователь нажимает тумблер, последний принимает одно из двух возможных состояний.

Тумблеры являются объектами класса `JToggleButton`. `JToggleButton` реализует `AbstractButton`. Кроме создания стандартных тумблеров, `JToggkeButton` является суперклассом двух других компонентов Swing, которые также представляют элементы управления, имеющие двумя состояниями. Это `JCheckBox` и `JRadioButton`. Таким образом, `JToggleButton` определяет базовые функции компонентов, обладающих двумя состояниями.

Класс `JToggleButton` определяет несколько конструкторов. Один из них, используемый в примере этого раздела, выглядит так:

```
JToggleButton(String str)
```

Он создает тумблер, содержащий текст, заданный с помощью параметра *`str`*. Стандартное состояние тумблера — отжатое. Остальные конструкторы позволяют создавать тумблеры, содержащие изображения, или изображения и текст.

Класс `JToggleButton` использует модель, определенную во вложенном классе `JToggleButton.ToggleButtonModel`. Как правило, вам не придется взаимодействовать непосредственно с моделью, чтобы использовать стандартный тумблер.

Как и `JButton`, `JToggleButton` генерирует событие действия каждый раз, когда пользователь нажимает тумблер. Однако, в отличие от `JButton`, `JToggleButton` также генерирует событие элемента. Это событие используется компонентами, которые поддерживают принцип выбора. Если тумблер `JToggleButton` нажат, он является выбранным. Если пользователь отжимает его, выбор отменяется.

Для обработки событий элемента нужно реализовать интерфейс `ItemListener`. Из главы 22 вы должны помнить, что при каждом генерировании события элемента оно передается методу `itemStateChanged()`, определенному в интерфейсе `ItemListener`. В методе `itemStateChanged()` метод `getItem()` может быть вызван в объекте `ItemEvent`, чтобы получить ссылку на экземпляр `JToggleButton`, сгенерировавший событие. Этот метод показан ниже:

```
Object getItem()
```

Он возвращает ссылку на кнопку. Вам нужно будет привести эту ссылку к классу `JToggleButton`.

Самый простой способ определить состояние тумблера предусматривает вызов метода `isSelected()` (он является наследником `AbstractButton`) для кнопки, сгенерировавшей событие. Этот метод выглядит следующим образом:

```
boolean isSelected()
```

Он возвращает значение `true`, если кнопка была выбрана; в противном случае он возвращает значение `false`.

Ниже показан пример, в котором используется тумблер. Обратите внимание на слушатель событий. Он просто вызывает метод `isSelected()`, чтобы определить состояние кнопки.

```
// Демонстрация применения 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 jtbn;
  public void init() {
    try {
      SwingUtilities.invokeAndWait(
        new Runnable() {
          public void run() {
            makeGUI();
          }
        }
      );
    } catch (Exception exc) {
      System.out.println("Невозможно создать из-за " + exc);
    }
  }
  private void makeGUI() {
    // Изменение компоновки потока.
    setLayout(new FlowLayout());
    // Создаем метку.
    jlab = new JLabel("Button is off.");
    // jlab = new JLabel("Кнопка отжата.");
    // Создаем тумблер.
    jtbn = new JToggleButton("On/Off");
    // Добавляем слушатель событий для тумблера.
    jtbn.addItemListener(new ItemListener() {
      public void itemStateChanged(ItemEvent ie) {
        if(jtbn.isSelected())
          jlab.setText("Button is on.");
          // jlab.setText("Кнопка нажата.");
        else
          jlab.setText("Button is off.");
          // jlab.setText("Кнопка отжата.");
      }
    });
    // Добавляем тумблер и метку в панель содержимого.
    add(jtbn);
    add(jlab);
  }
}
```

Результат выполнения этого аплета показан на рис. 30.4.

Рис. 30.4. Окно аплета `JToggleButtonDemo`

Флажки

Класс `JCheckBox` определяет функции флажка. Его суперклассом является `JToggleButton`, который обеспечивает поддержку кнопок с двумя состояниями (о них мы только что говорили). Класс `JCheckBox` определяет несколько конструкторов. Один из них выглядит следующим образом:

```
JCheckBox(String str)
```

Этот конструктор создает флажок с текстом, определенным с помощью параметра *`str`* в качестве метки. Остальные конструкторы позволяют определить исходное состояние выбора кнопки и задать значок.

Если пользователь отмечает флажок или снимает с него отметку, генерируется событие `ItemEvent`. Вы можете получить ссылку на флажок `JCheckBox`, сгенерировавший событие, вызвав метод `getItem()` в объекте `ItemEvent`, который передан методу `itemStateChanged()`, определенному в `ItemListener`. Определить выбранное состояние проще всего можно, вызвав метод `isSelected()` в экземпляре `JCheckBox`.

Помимо поддержки обычных операций с флажком `JCheckBox` позволяет определить значки, которые будут показывать состояния флажка, когда он будет отмечен, не отмечен или на него будет наведен указатель мыши. Здесь мы не будем использовать эти значки, однако вы смело можете их внедрять в своих программах.

В следующем аплете демонстрируется использование флажков. Аплет отображает четыре флажка и метку. Когда пользователь щелкает на флажке, возникает событие `ItemEvent`. В методе `itemStateChanged()` вызывается метод `getItem()` для получения ссылки на объект `JCheckBox`, сгенерировавший событие. После этого вызов метода `isSelected()` определяет состояние флажка: отмечен он или нет. Метод `getText()` получает текст для данного флажка и использует его для определения текста внутри метки.

```
// Демонстрация применения 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);
      // System.out.println("Невозможно создать из-за " + exc);
    }
  }

  private void makeGUI() {

    // Изменение компоновки потока.
    setLayout(new FlowLayout());

    // Добавление флажков в панель 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);

    // Создание метки и добавление ее в панель содержимого.
    jlab = new JLabel("Select languages");
    // jlab = new JLabel("Выберите языки");
    add(jlab);
  }

  // Обработка событий флажка.
  public void itemStateChanged(ItemEvent ie) {
    JCheckBox cb = (JCheckBox)ie.getItem();

    if(cb.isSelected())
      jlab.setText(cb.getText() + " is selected");
      // jlab.setText(cb.getText() + " выбран");
    else
      jlab.setText(cb.getText() + " is cleared");
      // jlab.setText(cb.getText() + " очищен");
  }
}
```

На рис. 30.5 показан результат выполнения аплета.

Рис. 30.5. Окно аплета `JCheckBoxDemo`

Переключатели

Переключатели представляют собой группы взаимоисключающих кнопок, в которых одновременно можно выбрать только одну кнопку. Они поддерживаются классом `JRadioButton`, который расширяет класс `JToggleButton`. Класс `JRadioButton` предлагает несколько конструкторов. Один из них, используемый в примере, показан ниже:

```
JRadioButton(String str)
```

Здесь *str* — это метка кнопки. Остальные конструкторы позволяют определить исходное состояние кнопки и задать значок.

Переключатели необходимо объединять в группу, в которой одновременно можно выбрать только одну из кнопок. Например, если пользователь выбирает переключатель, находящийся в группе, то все выбранные ранее переключатели в этой группе автоматически отключаются. Для создания группы кнопок предназначен класс `ButtonGroup`. Для этой цели вызывается его конструктор, используемый по умолчанию. После этого можно добавить в группу элементы с помощью следующего метода:

```
void add(AbstractButton ab)
```

Здесь *ab* представляет ссылку на кнопку, которую необходимо добавить в группу.

`JRadioButton` генерирует события действий, события элементов и события изменений каждый раз, когда меняется выбранная кнопка в переключателе. Как правило, обрабатывается событие действия, а это означает, что вы обычно будете реализовывать интерфейс `ActionListener`. Вы должны помнить, что `ActionListener` определяет только один метод — `actionPerformed()`. Узнать, какая кнопка была выбрана, в этом методе можно несколькими способами. Во-первых, можно проверить ее команду действия, вызвав метод `getActionCommand()`. По умолчанию команда действия — это то же, что и метка кнопки, однако вы можете задать вместо нее что-нибудь другое, вызвав метод `setActionCommand()` для переключателя. Во-вторых, можно вызвать метод `getSource()` в объекте `ActionEvent` и проверить ссылку относительно кнопок. Наконец, можно просто проверить каждый переключатель, чтобы узнать, какая кнопка в данный момент была выбрана, вызвав метод `isSelected()` для каждой кнопки. Помните, что каждый раз, когда возникает событие действия, оно означает, что выбранная кнопка была изменена, и что была выбрана только одна кнопка.

В следующем аплете демонстрируется использование переключателей. В нем создаются и объединяются в группу три переключателя. Как уже было сказано, это нужно для того, чтобы они взаимно исключали друг друга. При выборе переключателя генерируется событие действия, которое обрабатывается методом `actionPerformed()`. В этом обработчике метод `getActionCommand()` получает текст, который связан с переключателем, и использует его для определения текста в метке.

```
// Демонстрация применения 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);
      // System.out.println("Невозможно создать из-за " + exc);
    }
  }

  private void makeGUI() {
    // Изменение компоновки потока.
    setLayout(new FlowLayout());
    // Создание переключателей и добавление их в панель 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);

    // Определение группы кнопок.
    ButtonGroup bg = new ButtonGroup();
    bg.add(b1);
    bg.add(b2);
    bg.add(b3);

    // Создание метки и добавление ее в панель содержимого.
    jlab = new JLabel("Select One");
    // jlab = new JLabel("Выберите один из переключателей");
    add(jlab);
  }
  // Обработка выбора кнопки.
  public void actionPerformed(ActionEvent ae) {
    jlab.setText("You selected " + ae.getActionCommand());
    // jlab.setText("Выбран " + ae.getActionCommand());
  }
}
```

Результат выполнения этого аплета показан на рис. 30.6.

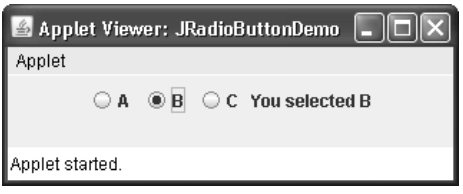

Рис. 30.6. Окно аплета `JRadioButtonDemo`

JTabbedPane

Класс `JTabbedPane` определяет *панель с вкладками*. Он управляет набором компонентов, соединяя их с помощью вкладок. При выборе вкладки компонент, связанный с ней, появляется на переднем плане. Панели с вкладками очень популярны в современных пользовательских графических интерфейсах, и вы тоже будете часто их применять в своих программах. Несмотря на сложную природу панели с вкладками, создавать и использовать их очень просто.

`JTabbedPane` определяет три конструктора. Мы будем использовать стандартный конструктор, который создает пустой элемент управления с вкладками, расположенными вдоль верхней части панели. Другие два конструктора позволяют определить месторасположение вкладок, которые можно расположить вдоль одной из четырех сторон. `JTabbedPane` использует модель `SingleSelectionModel`.

Вкладки добавляются посредством вызова метода `addTab()`. Ниже показана одна из его форм:

```
void addTab(String name, Component comp)
```

Здесь *name* — это имя вкладки, а *comp* — это компонент, который должна содержать вкладка. Обычно добавляется компонент `JPanel`, содержащий группу связанных между собой компонентов. Эта технология позволяет вкладке содержать набор компонентов.

Общая процедура использования панели с вкладками выглядит следующим образом.

1. Создается объект `JTabbedPane`.
2. Для добавления каждой вкладки нужно вызвать метод `addTab()`.
3. Панель с вкладками переносится в панель содержимого.

В следующем примере показан процесс создания панели с вкладками. Первая вкладка имеет заголовок Cities (Города) и содержит четыре кнопки. Каждая кнопка отображает название города. Вторая вкладка имеет заголовок Colors (Цвета) и содержит три флажка. Каждый флажок отображает название цвета. Третья вкладка имеет заголовок Flavors (Привкусы) и содержит одно комбинированное окно. С его помощью пользователь может выбрать один из трех привкусов.

```
// Демонстрация применения 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);
      // System.out.println("Невозможно создать из-за " + exc);
    }
  }
  private void makeGUI() {
    JTabbedPane jtp = new JTabbedPane();
    jtp.addTab("Cities", new CitiesPanel());
    jtp.addTab("Colors", new ColorsPanel());
    jtp.addTab("Flavors", new FlavorsPanel());
    add(jtp);
  }
}
// Создаем панели, которые будут добавляться в панель с вкладками.
class CitiesPanel extends JPanel {
  public CitiesPanel() {
    JButton b1 = new JButton("New York");
    add(b1);
    JButton b2 = new JButton("London");
    add(b2);
    JButton b3 = new JButton("Hong Kong");
    add(b3);
    JButton b4 = new JButton("Tokyo");
    add(b4);
  }
}
class ColorsPanel extends JPanel {
  public ColorsPanel() {
    JCheckBox cb1 = new JCheckBox("Red");
    add(cb1);
    JCheckBox cb2 = new JCheckBox("Green");
    add(cb2);
    JCheckBox cb3 = new JCheckBox("Blue");
    add(cb3);
  }
}
class FlavorsPanel extends JPanel {
  public FlavorsPanel() {
    JComboBox jcb = new JComboBox();
    jcb.addItem("Vanilla");
    jcb.addItem("Chocolate");
    jcb.addItem("Strawberry");
    add(jcb);
  }
}
```

На рис. 30.7 показаны окна этого аплета.

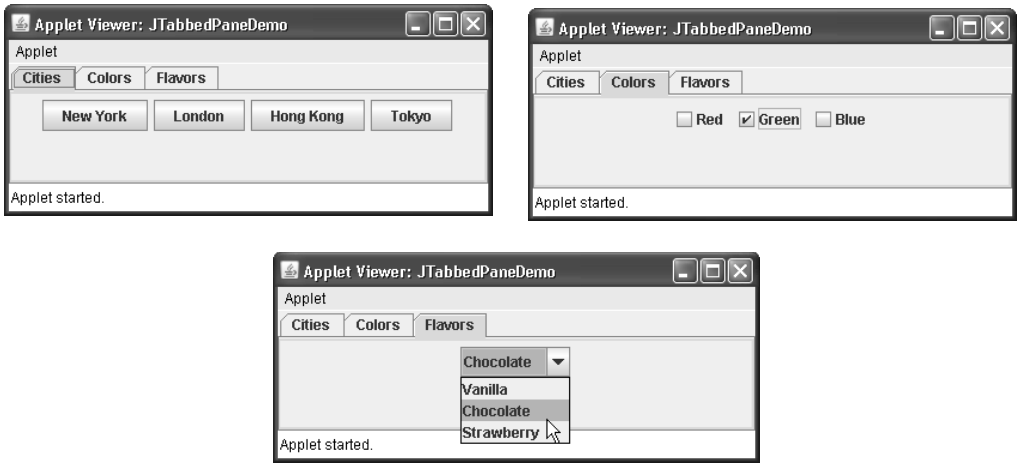

Рис. 30.7. Окна аплета `JTabbedPaneDemo`

JScrollPane

`JScrollPane` — это облегченный контейнер, который автоматически обрабатывает прокрутку другого компонента. Прокручиваемым компонентом может быть как отдельный компонент (например, таблица), так и группа компонентов, содержащихся внутри другого облегченного контейнера, такого как `JPanel`. В любом случае, если прокручиваемый объект больше области просмотра, к нему автоматически добавляется горизонтальная и/или вертикальная линейка прокрутки, что позволяет прокручивать компонент в рамках панели. Поскольку `JScrollPane` автоматизирует процесс прокрутки, он обычно исключает необходимость в управлении отдельными линейками прокрутки.

Просматриваемая область панели с линейками прокрутки называется *окном просмотра*. Это окно, в котором отображается прокручиваемый компонент. Таким образом, окно просмотра показывает видимую часть прокручиваемого компонента. Линейки прокрутки прокручивают компонент в области просмотра. По умолчанию `JScrollPane` динамически добавляет или удаляет линейку прокрутки по мере необходимости. Например, если компонент выше окна просмотра, добавляется вертикальная линейка прокрутки. Если компонент полностью умещается в окне просмотра, линейки прокрутки удаляются.

`JScrollPane` определяет несколько конструкторов. Конструктор, используемый в этой главе, выглядит так:

```
JScrollPane(Component comp)
```

Прокручиваемый компонент задается с помощью параметра *comp*. Линейки прокрутки автоматически отображаются, если содержимое панели превышает размеры окна просмотра.

Чтобы использовать панель с линейками прокрутки, нужно выполнить следующие шаги.

1. Создать компонент, который будет прокручиваться.
2. Создать экземпляр `JScrollPane`, передавая ему объект прокрутки.
3. Добавить панели с линейками прокрутки в панель содержимого.

На примере следующего аплета демонстрируется применение панели с линейками прокрутки. Сначала в нем создается объект `JPanel`, после чего в этот объект добавляется 400 кнопок, сгруппированных в 20 столбцов. Затем эта панель добавляется в панель с линейками прокрутки, а последняя добавляется в панель содержимого. Поскольку панель больше окна просмотра, то автоматически появляются вертикальная и горизонтальная линейки прокрутки. Линейки прокрутки можно использовать для того, чтобы прокручивать кнопки в данном представлении.

```
// Демонстрация применения 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);
      // System.out.println("Невозможно создать из-за " + exc);
    }
  }
  private void makeGUI() {
    // Добавление 400 кнопок в панель.
    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;
      }
    }
    // Создание панели с прокруткой.
    JScrollPane jsp = new JScrollPane(jp);

    // Добавление панели с прокруткой в панель содержимого.
    // Так как используется компоновка границ по умолчанию,
    // панель с линейками прокрутки добавляется в центр.
    add(jsp, BorderLayout.CENTER);
  }
}
```

На рис. 30.8 показан результат выполнения этого аплета.

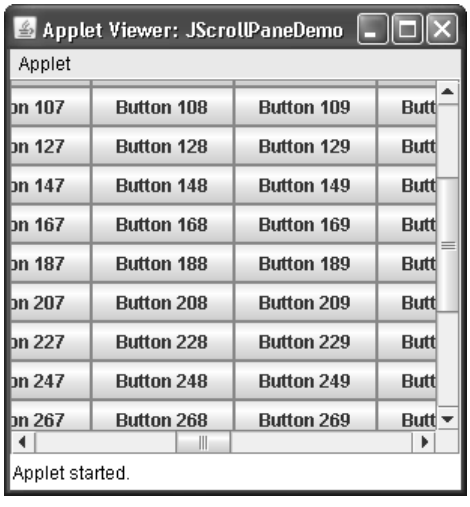

Рис. 30.8. Окно аплета `JScrollPaneDemo`

JList

В Swing базовый класс списков называется `JList`. Он поддерживает выбор одного или нескольких элементов из списка. Хотя список часто состоит из строк, можно создать список из любых объектов, которые только можно отобразить. `JList` настолько часто применяется в Java, что вы уже должны были обязательно с ним встретиться.

`JList` предлагает несколько конструкторов. Один из используемых конструкторов выглядит так:

```
JList(Object[] items)
```

Этот конструктор создает список `JList`, содержащий элементы в массиве, определяемые с помощью параметра *`items`*.

`JList` основан на двух моделях. Первая модель — `ListModel`. Этот интерфейс определяет, как достигается доступ к данным списка. Второй моделью является интерфейс `ListSelectionModel`, который определяет методы, позволяющие узнать, какой элемент (элементы) списка был выбран.

Хотя `JList` будет работать самостоятельно без проблем, чаще всего вам придется вкладывать `JList` в `JScrollPane`. Благодаря этому длинные списки автоматически будут прокручиваться, что упростит проектирование графического интерфейса. Также это позволит упростить изменение количества записей в списке, не изменяя размеры компонента `JList`.

`JList` генерирует событие `ListSelectionEvent`, когда пользователь выбирает или производит изменения в выборе элемента. Это событие генерируется также тогда, когда пользователь отменяет выбор элемента. Оно обрабатывается слушателем `ListSelectionListener`. Этот слушатель определяет только один метод — `valueChanged()`:

```
void valueChanged(ListSelectionEvent le)
```

Здесь *`le`* — ссылка на объект, который сгенерировал событие.

Хотя событие ListSelectionEvent само предлагает некоторые методы, вы часто будете использовать сам объект JList, чтобы узнать, что произошло.

Событие ListSelectionEvent и слушатель ListSelectionListener определены в пакете javax.swing.event.

По умолчанию JList позволяет пользователю выбирать несколько диапазонов элементов внутри списка, однако вы можете изменить это поведение, вызвав метод setSelectionModel(), который определен в JList. Он показан ниже:

```
void setSelectionMode(int mode)
```

Здесь *mode* — это режим выбора. Он должен быть представлен одним из следующих значений, определенных в ListSelectionModel:

```
SINGLE_SELECTION
SINGLE_INTERVAL_SELECTION
MULTIPLE_INTERVAL_SELECTION
```

По умолчанию используется последнее значение, которое позволяет пользователю выбирать множество диапазонов элементов внутри списка. Если будет задан выбор в единичном интервале (SINGLE_INTERVAL_SELECTION), пользователь сможет выбрать только один диапазон элементов. Если будет выбрано значение SINGLE_SELECTION, пользователь сможет выбрать только один элемент. Естественно, один элемент можно выбрать и в двух других режимах. Просто эти режимы позволяют также выбирать диапазон элементов.

Вы можете получить индекс первого выбранного элемента, который будет также являться индексом единственного выбранного элемента в режиме SINGLE_SELECTION, если вызовете метод getSelectedIndex(), показанный ниже:

```
int getSelectedIndex()
```

Индексация начинается с нуля. Поэтому если будет выбран первый элемент, метод вернет значение 0. Если не будет выбрано ни одного элемента, будет возвращено значение -1.

Вместо того чтобы получать индекс выбранного элемента, вы можете получить значение, связанное с выбранным элементом, вызвав метод getSelectedValue():

```
Object getSelectedValue()
```

Он возвращает ссылку на первое выбранное значение. Если не будет выбрано ни одного значения, метод вернет null.

На примере следующего аплета демонстрируется использование простого списка JList, хранящего перечень городов. Каждый раз, когда пользователь будет выбирать город в списке, будет генерироваться событие ListSelectionEvent, обработкой которого занимается метод valueChanged(), определенный в слушателе ListSelectionListener. Он получает индекс выбранного элемента и отображает имя выбранного города в метке.

```
// Демонстрация применения 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 jlst;
  JLabel jlab;
  JScrollPane jscrlp;
  // Создаем массив городов.
  String Cities[] = { "New York", "Chicago", "Houston",
                      "Denver", "Los Angeles", "Seattle",
                      "London", "Paris", "New Delhi",
                      "Hong Kong", "Tokyo", "Sydney" };
  public void init() {
    try {
      SwingUtilities.invokeAndWait(
        new Runnable() {
          public void run() {
            makeGUI();
          }
        }
      );
    } catch (Exception exc) {
      System.out.println("Can't create because of " + exc);
      // System.out.println("Невозможно создать из-за " + exc);
    }
  }
  private void makeGUI() {
    // Изменение компоновки потока.
    setLayout(new FlowLayout());
    // Создаем список JList.
    jlst = new JList(Cities);
    // Присваиваем режиму выбора значение SINGLE_SELECTION.
    jlst.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    // Добавляем список в панель с линейками прокрутки.
    jscrlp = new JScrollPane(jlst);
    // Задаем предпочтительные размеры панели с линейками прокрутки.
    jscrlp.setPreferredSize(new Dimension(120, 90));
    // Создаем метку, в которой будет отображаться выбранный город.
    jlab = new JLabel("Choose a City");
    // jlab = new JLabel("Выберите город");

    // Добавляем слушатель выбора для списка.
    jlst.addListSelectionListener(new ListSelectionListener() {
      public void valueChanged(ListSelectionEvent le) {
        // Получаем индекс измененного элемента.
        int idx = jlst.getSelectedIndex();
        // Отображаем выбор, если элемент был выбран.
       if(idx != -1)
         jlab.setText("Current selection: " + Cities[idx]);
         // jlab.setText("Текущий выбор: " + Cities[idx]);
       else // В противном случае повторно предлагаем выбрать город.
         jlab.setText("Choose a City");
         // jlab.setText("Выберите город");
      }
    });
    // Добавляем список и метку в панель содержимого.
    add(jscrlp);
    add(jlab);
  }
}
```

Результат выполнения этого аплета показан на рис. 30.9.

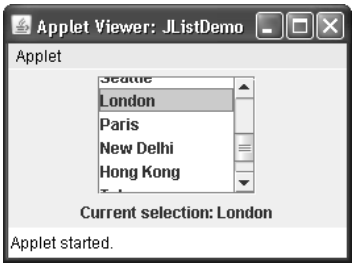

Рис. 30.9. Окно аплета `JListDemo`

JComboBox

С помощью Swing-класса `JComboBox` определяется компонент, называемый *комбинированным окном списка* (комбинация текстового поля и выпадающего списка). Как правило, комбинированное окно отображает одну запись, но оно будет также отображать и выпадающий список, позволяющий выбирать другие элементы. Вы можете создать комбинированное окно, которое позволит вводить выбираемый элемент в текстовом поле. Ниже показан конструктор `JComboBox`, используемый в этом примере:

```
JComboBox(Object[ ] items)
```

Здесь *items* — это массив, инициализирующий комбинированное окно. Кроме него доступны также и другие конструкторы.

`JComboBox` использует модель `ComboBoxModel`. Изменяемые комбинированные окна (окна, элементы которых могут изменяться) используют модель `MutableComboBoxModel`.

Кроме передачи массива элементов, которые должны быть отображены в выпадающем списке, элементы можно добавлять динамически в список посредством метода `addItem()`, показанном ниже:

```
void addItem(Object obj)
```

Здесь *obj* — это объект, который необходимо добавить в комбинированное окно. Этот метод должен использоваться только в изменяемых комбинированных окнах.

`JComboBox` генерирует событие действия, когда пользователь выбирает элемент из списка. `JComboBox` также генерирует событие элемента, когда изменяется состояние выбора, что происходит при выборе или отмене выбора элемента. Таким образом, изменение выбора означает возникновение двух событий: одно возникает для элемента, выбор которого был отменен, а другое — для выбранного элемента. Часто оказывается достаточным простого прослушивания событий действия, однако использовать можно оба типа событий.

Узнать, какой элемент списка был выбран, можно с помощью метода `getSelectedItem()`:

```
Object getSelectedItem( )
```

Вам нужно будет привести возвращенное значение к типу объекта, хранящегося в списке.

На примере следующего аплета показано использование комбинированного окна списка. В этом окне содержатся элементы “France”, “Germany”, “Italy” и “Japan”. Если поль-

зователь выберет государство, произойдет обновление метки, чтобы в ней был отображен флаг данного государства. Вы посмотрите, насколько мало кода нужно для того, чтобы использовать этот мощный компонент.

```
// Демонстрация применения JComboBox.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
  <applet code="JComboBoxDemo" width=300 height=100>
  </applet>
*/
public class JComboBoxDemo extends JApplet {
  JLabel jlab;
  ImageIcon france, germany, italy, japan;
  JComboBox jcb;

  String flags[] = { "France", "Germany", "Italy", "Japan" };

  public void init() {
    try {
      SwingUtilities.invokeAndWait(
        new Runnable() {
          public void run() {
            makeGUI();
          }
        }
      );
    } catch (Exception exc) {
      System.out.println("Can't create because of " + exc);
      // System.out.println("Невозможно создать из-за " + exc);
    }
  }

  private void makeGUI() {

    // Изменение компоновки потока.
    setLayout(new FlowLayout());

    // Создание экземпляра комбинированного окна
    // и добавление его в панель содержимого.
    jcb = new JComboBox(flags);
    add(jcb);

    // Обработка выбранных элементов.
    jcb.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ae) {
        String s = (String) jcb.getSelectedItem();
        jlab.setIcon(new ImageIcon(s + ".gif"));
      }
    });

    // Создание метки и добавление ее в панель содержимого.
    jlab = new JLabel(new ImageIcon("france.gif"));
    add(jlab);
  }
}
```

На рис. 30.10 показан результат выполнения аплета.

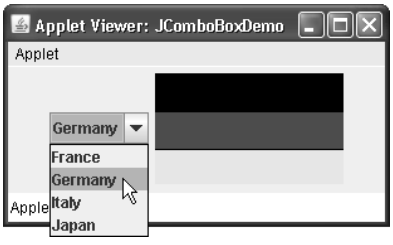

Рис. 30.10. Окно аплета `JComboBoxDemo`

Деревья

Дерево — это компонент, представляющий иерархический вид данных. Пользователь может развертывать или свертывать отдельные поддеревья в данном отображении. В Swing деревья реализуются посредством класса `JTree`. Ниже показаны некоторые из его конструкторов:

```
JTree(Object obj[ ])
JTree(Vector<?> v)
JTree(TreeNode tn)
```

В первом случае дерево создается из элементов массива `obj`. Во втором случае дерево создается из элементов вектора (параметр `v`). В третьем случае дерево определяется корневым узлом, который, в свою очередь, определяется соответствующим параметром (`tn`).

Хотя `JTree` входит в состав пакета `javax.swing`, он поддерживает классы и интерфейсы, которые определены в пакете `javax.swing.tree`. Это объясняется тем, что количество классов и интерфейсов, необходимых для поддержки `JTree`, слишком большое.

`JTree` базируется на двух моделях: `TreeModel` и `TreeSelectionModel`. `JTree` генерирует разнообразные события, однако применительно к деревьям можно выделить три из них: `TreeExpansionEvent`, `TreeSelectionEvent` и `TreeModelEvent`. Событие `TreeExpansionEvent` возникает, когда узел развертывается или свертывается. Событие `TreeSelectionEvent` возникает, когда пользователь выбирает или отменяет выбор узла в дереве. Событие `TreeModelEvent` возникает, когда изменяются данные или структура дерева. За этими событиями следят слушатели `TreeExpansionListener`, `TreeSelectionListener` и `TreeModelListener`, соответственно. Классы событий дерева и интерфейсы слушателей определены в пакете `javax.swing.event`.

В примере программы этого раздела обрабатывается событие `TreeSelectionEvent`. Чтобы прослушать это событие, нужно реализовать слушатель `TreeSelectionListener`. Он определяет только один метод, называемый `valueChanged()`, который получает объект `TreeSelectionEvent`. Вы можете получить путь к выбранному объекту, обратившись к методу `getPath()`:

```
TreePath getPath()
```

Метод возвращает объект `TreePath`, который описывает путь к измененному узлу. Класс `TreePath` инкапсулирует информацию о пути к определенному узлу в дереве. Он предлагает несколько методов и конструкторов. В этой книге используется только метод `toString()`. Он возвращает строку, описывающую путь.

Интерфейс TreeNode объявляет методы, которые получают информацию об узле дерева. Например, можно получить ссылку на родительский узел или перечень узлов-потомков. Интерфейс MutableTreeNode расширяет интерфейс TreeNode. Он объявляет методы, которые могут вставлять и удалять узлы-потомки или изменять родительский узел.

Класс DefaultMutableTreeNode реализует интерфейс MutableTreeNode. Он представляет узел в дереве. Ниже показан один из его конструкторов:

```
DefaultMutableTreeNode(Object obj)
```

Здесь *obj* — это объект, который необходимо заключить в данном узле дерева. Новый узел дерева не имеет родителя или потомка.

Чтобы создать иерархию из трех узлов, можно использовать метод add() конструктора DefaultMutableTreeNode. Ниже показана его сигнатура:

```
void add(MutableTreeNode child)
```

Здесь *child* — это изменяющийся узел дерева, который необходимо добавить в качестве потомка текущего узла.

JTree сам по себе не предлагает никаких возможностей для прокрутки. Вместо этого JTree обычно помещается внутри JScrollPane. Таким образом, большое дерево можно прокрутить в окне просмотра с меньшими размерами.

Ниже перечислены действия, которые необходимо выполнить, чтобы использовать дерево в аплете.

1. Создайте экземпляр JTree.
2. Создайте объект JScrollPane и определите дерево в качестве объекта прокрутки.
3. Добавьте дерево в панель с линейками прокрутки.
4. Добавьте панель с линейками прокрутки в панель содержимого.

В следующем примере показано, как создается дерево и обрабатывается выбор элементов. Программа создает объект DefaultMutableTreeNode с заголовком Options (Опции). Он является верхним узлом в иерархии дерева. Затем создаются дополнительные узлы дерева, и вызывается метод add() для присоединения этих узлов к дереву. Ссылку на верхний узел дерева обеспечивает параметр конструктора JTree. После этого дерево указывается в качестве параметра конструктора JScrollPane. Эта панель с линейками прокрутки добавляется в аплет. Затем создается метка и добавляется в панель содержимого. В этой метке отображается выбор в дереве. Чтобы получить события при выборе, регистрируется слушатель TreeSelectionListener. Внутри метода valueChanged() мы получаем и отображаем путь к текущему месту выбора.

```
// Демонстрация применения 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);
      // System.out.println("Невозможно создать из-за " + exc);
    }
  }

  private void makeGUI() {

    // Создаем верхний узел дерева.
    DefaultMutableTreeNode top = new DefaultMutableTreeNode("Options");

    // Создаем поддерево "A".
    DefaultMutableTreeNode a = new DefaultMutableTreeNode("A");
    top.add(a);
    DefaultMutableTreeNode a1 = new DefaultMutableTreeNode("A1");
    a.add(a1);
    DefaultMutableTreeNode a2 = new DefaultMutableTreeNode("A2");
    a.add(a2);

    // Создаем поддерево "B".
    DefaultMutableTreeNode b = new DefaultMutableTreeNode("B");
    top.add(b);
    DefaultMutableTreeNode b1 = new DefaultMutableTreeNode("B1");
    b.add(b1);
    DefaultMutableTreeNode b2 = new DefaultMutableTreeNode("B2");
    b.add(b2);
    DefaultMutableTreeNode b3 = new DefaultMutableTreeNode("B3");
    b.add(b3);

    // Создаем дерево.
    tree = new JTree(top);

    // Добавляем дерево в панель прокрутки.
    JScrollPane jsp = new JScrollPane(tree);

    // Добавляем панель с линейками прокрутки в панель содержимого.
    add(jsp);

    // Добавляем метку в панель содержимого.
    jlab = new JLabel();
    add(jlab, BorderLayout.SOUTH);

    // Обработка событий выбора в дереве.
    tree.addTreeSelectionListener(new TreeSelectionListener() {
      public void valueChanged(TreeSelectionEvent tse) {
        jlab.setText("Selection is " + tse.getPath());
        // jlab.setText("Выбрано " + tse.getPath());
      }
    });
  }
}
```

На рис. 30.11 показан результат выполнения этого аплета.

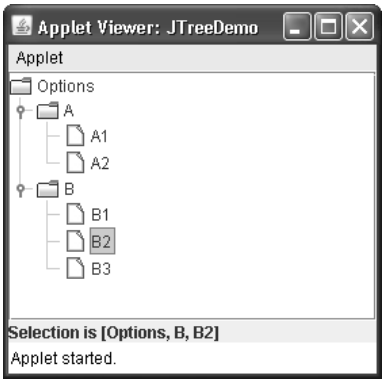

Рис. 30.11. Окно аплета `JTreeDemo`

Строка, представленная в текстовом поле, определяет путь из верхнего узла дерева к выбранному узлу.

JTable

`JTable` — это компонент, отображающий данные в виде строк и столбцов. Чтобы изменить размеры столбцов, вы можете перетаскивать их границы с помощью мыши. Кроме того, весь столбец можно перетащить в другую позицию. В зависимости от конфигурации, можно выбрать строку, столбец или ячейку в таблице, а также изменять данные в ячейке. `JTable` является сложным компонентом, предлагающим гораздо больше опций и функций, чем мы можем рассмотреть здесь. (Наверное, это наиболее сложный компонент Swing.) Однако в своей стандартной конфигурации `JTable` предлагает легкие в использовании функции — особенно если вы просто хотите использовать таблицу для представления данных в табличном формате. Краткий обзор, представленный здесь, поможет вам понять, какими возможностями обладает этот компонент.

Как и `JTree`, `JTable` обладает многими классами и интерфейсами, связанными с ним. Все они находятся в пакете `javax.swing.table`.

В самой своей основе `JTable` является очень простым. Он состоит из одного или нескольких столбцов с информацией. Вверху каждого столбца расположен заголовок. Кроме описания данных в столбце, заголовок также предлагает механизм, посредством которого пользователь может изменять размеры столбца или изменять местонахождение столбца в таблице. `JTable` не предлагает никаких возможностей прокрутки. Вместо этого вы обычно будете внедрять `JTable` в `JScrollPane`.

`JTable` предлагает несколько конструкторов. Один из них выглядит следующим образом:

```
JTable(Object data[ ][ ], Object colHeads[ ])
```

Здесь *`data`* — это двумерный массив информации, которую нужно представить, а *`colHeads`* — одномерный массив, содержащий заголовки столбцов.

`JTable` основан на трех моделях. Первой из них является модель таблицы, которая определена интерфейсом `TableModel`. Эта модель определяет все, что связано с отобра-

жением данных в двумерном формате. Вторая модель — модель столбца таблицы, представленная с помощью `TableColumnModel`. `JTable` определяется посредством столбцов — это модель `TableColumnModel`, которая определяет характеристики столбца. Эти две модели реализованы в пакете `javax.swing.table`. Третья модель определяет способ выбора элементов; она определяется с применением `ListSelectionModel` (мы уже рассматривали ее при обсуждении `JList`).

`JTable` может генерировать несколько различных событий. Двумя наиболее фундаментальными операциями являются `ListSelectionEvent` и `TableModelEvent`. `ListSelectionEvent` генерируется, когда пользователь выбирает что-нибудь в таблице. По умолчанию `JTable` позволяет выбрать одну или несколько полных строк, однако вы можете изменить это поведение, чтобы позволить пользователю выбирать один или несколько столбцов или одну или несколько отдельных ячеек. Событие `TableModelEvent` генерируется, когда данные таблицы каким-либо образом изменяются. Обработка этих событий требует чуть большего труда, чем в ранее описанных компонентах, и ее мы не сможем рассмотреть в этой книге. Однако если вам просто понадобится использовать `JTable` для отображения данных (как в следующем примере), то никакие события обрабатывать не придется.

Для создания простой таблицы `JTable`, отображающей данные, выполните следующие действия.

1. Создайте экземпляр `JTable`.
2. Создайте объект `JScrollPane`, определяя таблицу в качестве объекта прокрутки.
3. Добавьте таблицу в панель с линейками прокрутки.
4. Добавьте панель с линейками прокрутки в панель содержимого.

Ниже показан пример создания и использования простой таблицы. В этом аплете сначала создается одномерный массив строк `colHeads` для заголовков столбцов. Двумерный массив строк `data` конструируется для ячеек таблицы. Каждый элемент в массиве является массивом из трех строк. Эти массивы передаются конструктору `JTable`. Таблица добавляется в панель с линейками прокрутки, после чего последняя добавляется в панель содержимого. Таблица отображает данные в массиве `data`. Конфигурация таблицы, используемая по умолчанию, позволяет также редактировать содержимое ячеек. Все изменения будут отражены в массиве `data`.

```
// Демонстрация применения 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.invokeAndWait(
        new Runnable() {
          public void run() {
            makeGUI();
          }
        }
      );
```

```
    } catch (Exception exc) {
      System.out.println("Can't create because of " + exc);
      // System.out.println("Невозможно создать из-за " + exc);
    }
  }

  private void makeGUI() {

    // Инициализируем заголовки столбцов.
    String[] colHeads = { "Name", "Extension", "ID#" };

    // Инициализируем 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" }
    };

    // Создаем таблицу.
    JTable table = new JTable(data, colHeads);

    // Добавляем таблицу в панель с линейками прокрутки.
    JScrollPane jsp = new JScrollPane(table);

    // Добавляем панель с линейками прокрутки в панель содержимого.
    add(jsp);
  }
}
```

На рис. 30.12 показан результат выполнения этого аплета.

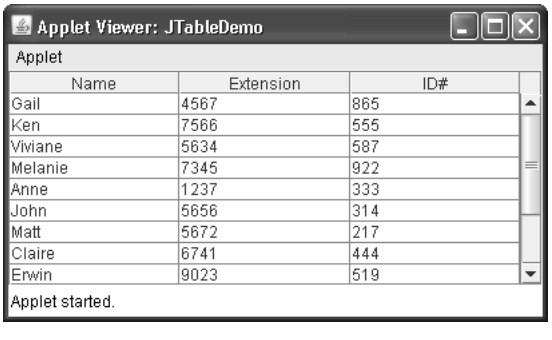

Рис. 30.12. Окно аплета `JTableDemo`

Продолжайте изучать Swing

Swing определяет очень большой набор инструментов пользовательского графического интерфейса. В ней заключены многие возможности, которые вам придется изучать самостоятельно. Например, Swing предлагает панели инструментов, контекстные подсказки и панели с индикаторами хода работ. Также Swing предлагает полную подсистему меню. Подключаемые внешние виды, которые тоже предлагает Swing, позволяют получать разные варианты внешнего вида и поведения элемента. Вы можете определять собственные модели различных компонентов, а при работе с таблицами и деревьями изменять способ редактирования и визуализации данных в ячейках. Лучший способ изучения возможностей Swing — постоянно экспериментировать с ними.

31 ГЛАВА

Сервлеты

В этой главе речь пойдет о *сервлетах*. Сервлетами (servlet) называются небольшие программы, которые выполняются на стороне сервера Web-соединения. Подобно аплетам, которые динамические расширяют функциональные возможности Web-браузера, сервлеты динамически расширяют функциональность Web-сервера. Тема сервлетов является довольно обширной и ее невозможно рассмотреть полностью в рамках одной главы. Поэтому мы сосредоточимся на рассмотрении концепций, интерфейсов и классов, а также проанализируем некоторые примеры.

Предварительные сведения

Чтобы разобраться с преимуществами сервлетов, вы должны иметь общее представление о том, как Web-браузеры и сервлеты работают сообща для предоставления содержимого пользователю. Рассмотрим запрос статической Web-страницы. Пользователь вводит в окне браузера URL-адрес (Uniform Resource Locator — унифицированный указатель информационного ресурса). Браузер генерирует HTTP-запрос к соответствующему Web-серверу. Web-сервер устанавливает соответствие между запросом и конкретным файлом. Этот файл возвращается браузеру в виде HTTP-отклика. HTTP-заголовок в отклике указывает тип содержимого. Для этой цели используется набор стандартов MIME (Multipurpose Internet Mail Extensions — многоцелевые расширения электронной почты). Например, обычный текст в формате ASCII имеет MIME-тип text/plain. Исходный код HTML (Hypertext Markup Language — язык разметки гипертекста) имеет MIME-тип text/html.

Теперь рассмотрим динамическое содержимое. Предположим, что магазин, работающий в онлайновом режиме, использует базу данных для хранения информации о своей бизнес-деятельности. База данных (БД) может включать элементы для регистрации продаж, прайс-листов, наличия товара, счетов и т.п. Руководство магазина решило сделать так, чтобы эта информация была доступна покупателям через Web-страницы. Содержимое этих Web-страниц должно генерироваться динамически, чтобы отражать самую последнюю информацию в базе данных.

На ранних этапах существования системы Web сервер мог динамически формировать страницу, создавая отдельный процесс для обработки каждого запроса клиента. Чтобы

получить необходимую информацию, процесс мог открывать соединения с одной или несколькими БД. Связь с сервером осуществлялась посредством интерфейса CGI (Common Gateway Interface — общий шлюзовой интерфейс). CGI позволял отдельным процессам считывать данные из HTTP-запроса и записывать данные в HTTP-отклик. Для написания CGI-программ применялись самые разные языки программирования. В их число входили C, C++ и Perl.

Однако у CGI имелись серьезные проблемы, связанные с производительностью. Этот интерфейс был дорогим в плане потребления ресурсов процессора и памяти, необходимых для создания отдельного процесса для каждого запроса клиента. Он был также дорогим в плане открытия и закрытия соединений с БД для каждого запроса клиента. Помимо всего этого, работа CGI-программ зависела от конкретной платформы. Потому были предложены другие технологии, к числу которых относятся и сервлеты.

По сравнению с CGI сервлеты обладают некоторыми преимуществами. Во-первых, их производительность заметно выше. Сервлеты выполняются внутри адресного пространства Web-сервера. Чтобы выполнить обработку каждого запроса клиента, не обязательно создавать отдельный процесс. Во-вторых, работа сервлетов не зависит от платформы, поскольку все они пишутся на языке Java. В-третьих, диспетчер безопасности Java на сервере реализует серию ограничений для защиты ресурсов на компьютере-сервере. И, наконец, сервлету доступны абсолютно все функциональные возможности библиотек классов Java. Сервлет может работать с аплетами, БД и другим программным обеспечением посредством сокетов и механизмов RMI, которые уже рассматривались в этой книге ранее.

Жизненный цикл сервлета

Жизненный цикл сервлета определяют три основных метода: `init()`, `service()` и `destroy()`. Они реализуются каждым сервлетом и вызываются сервером в определенное время. Сейчас мы рассмотрим обычный пользовательский сценарий, который поможет понять, когда происходит вызов этих методов.

Во-первых, предположим, что пользователь ввел в окне браузера URL-адрес. На основании этого URL-адреса браузер генерирует HTTP-запрос, посылаемый соответствующему серверу.

Во-вторых, этот HTTP-запрос принимает Web-сервер. Сервер находит соответствие между запросом и конкретным сервлетом. Сервлет динамически принимается и загружается в адресное пространство сервера.

В-третьих, сервер вызывает метод `init()` сервлета. Этот метод вызывается только тогда, когда сервлет впервые загружается в память компьютера. Сервлету можно передавать параметры инициализации, поэтому он может конфигурировать себя самостоятельно.

В-четвертых, сервер вызывает метод `service()` сервлета. Этот метод вызывается для обработки HTTP-запроса. Вы увидите, что сервлет может считывать данные, содержащиеся в HTTP-запросе. Он может также сформулировать HTTP-отклик клиенту.

Сервлет остается в адресном пространстве и является доступным для обработки любых других HTTP-запросов, полученных от клиентов. Метод `service()` вызывается для каждого HTTP-запроса.

И, наконец, сервер может принять решение загрузить сервлет из памяти. Для принятия этого решения каждый сервер использует различные алгоритмы. Для освобождения ресурсов, таких как индексы файлов, выделенных для сервлета, сервер вызывает метод `destroy()`. Важные данные могут быть сохранены на постоянном носителе. Память, отведенная для сервлета и его объектов, впоследствии может быть утилизирована в процессе сборки мусора.

Использование Tomcat для разработки сервлетов

Для создания сервлетов вам будет необходим доступ к среде разработки сервлетов. Одной из таких сред, применяемых в этой главе, является Tomcat. Tomcat представляет собой открытый продукт, который поддерживает проект Jakarta от фонда Apache Software Foundation. Этот продукт содержит библиотеки классов, документацию и среду времени выполнения, что будет необходимо для создания и тестирования сервлетов. На момент написания настоящей книги текущей версией Tomcat была 5.5.17 с поддержкой спецификации сервлетов 2.4. Вы можете загрузить Tomcat с сайта `jakarta.apache.org`.

Условием выполнения примеров из этой главы является работа в среде Windows. Стандартное размещение Tomcat в Windows выглядит как:

```
C:\Program Files\Apache Software Foundation\Tomcat 5.5\
```

Именно это размещение предполагается в примерах из этой книги. Если вы загрузите Tomcat в другой каталог, вам придется внести соответствующие изменения в примерах. Для этого, возможно, нужно будет присвоить переменной среды `JAVA_HOME` каталог верхнего уровня, в котором установлен Java Development Kit.

Чтобы приступить к использованию Tomcat, выберите пункт Configure Tomcat (Конфигурирование Tomcat) в меню Start⇨Programs (Пуск⇨Программы), а затем щелкните на кнопке Start (Пуск) в диалоговом окне Tomcat Properties (Свойства Tomcat).

После завершения тестирования сервлетов вы можете остановить работу Tomcat, щелкнув на кнопке Stop (Остановить) в диалоговом окне Tomcat Properties.

В каталоге

```
C:\Program Files\Apache Software Foundation\Tomcat 5.5\common\lib\
```

хранится файл `servlet-api.jar`. В этом JAR-файле содержатся классы и интерфейсы, необходимые для создания сервлетов. Чтобы сделать этот файл доступным, обновите переменную среды `CLASSPATH`, чтобы она включала следующую строку:

```
C:\Program Files\Apache Software Foundation\Tomcat 5.5\common\lib\servlet-api.jar
```

В качестве альтернативного варианта этот файл классов можно определить во время компиляции сервлетов. Например, следующая команда компилирует первый пример сервлета:

```
javac HelloServlet.java -classpath "C:\Program Files\Apache Software
  Foundation\Tomcat 5.5\common\lib\servlet-api.jar"
```

После того как вы завершите компиляцию сервлета, нужно сделать так, чтобы Tomcat нашел его. Это означает, что необходимо поместить его в подкаталог каталога Tomcat `webapps` и ввести его имя в файле `web.xml`. Для упрощения всех этих действий в примерах этой главы применяется каталог и файл `web.xml`, который Tomcat использует для своих собственных образцов сервлетов. Ниже приведена процедура, которую вам необходимо будет выполнить.

Во-первых, скопируйте файл класса сервлета в следующий каталог:

```
Program Files\Apache Software Foundation\Tomcat 5.5\webapps\
  servlets-examples\WEB-INF\classes
```

Затем добавьте имя сервлета и отображение в файл web.xml в следующий каталог:

```
Program Files\Apache Software Foundation\Tomcat 5.5\webapps\
  servlets-examples\WEB-INF
```

Например, если предположить, что первый пример будет называться HelloServlet, в раздел, описывающий сервлеты, вы добавите следующие строки:

```
<servlet>
  <servlet-name>HelloServlet</servlet-name>
  <servlet-class>HelloServlet</servlet-class>
</servlet>
```

Затем добавьте следующие строки в раздел, определяющий преобразования сервлетов:

```
<servlet-mapping>
  <servlet-name>HelloServlet</servlet-name>
  <url-pattern>/servlet/HelloServlet</url-pattern>
</servlet-mapping>
```

Выполните эти же действия для всех примеров.

Простой сервлет

Чтобы освоиться с ключевыми особенностями сервлетов, мы начнем с того, что создадим и протестируем простой сервлет. Для этого необходимо выполнить перечисленные ниже основные действия.

1. Создайте и скомпилируйте исходный код сервлета. После этого скопируйте файл классов сервлета в соответствующий каталог и добавьте имя сервлета и преобразования в соответствующий файл web.xml.
2. Запустите Tomcat.
3. Запустите Web-браузер и запросите сервлет.

Теперь давайте рассмотрим подробно каждое действие.

Создание и компиляция исходного кода сервлета

Для начала создайте файл с именем HelloServlet.java, который будет содержать следующую программу:

```
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();
  }
}
```

Взгляните на эту программу. Для начала обратите внимание, что она импортирует пакет `javax.servlet`. Этот пакет содержит классы и интерфейсы, необходимые для создания сервлетов. К ним мы еще вернемся далее в этой главе. Эта программа также определяет класс `HelloServlet` как подкласс `GenericServlet`. Класс `GenericServlet` обладает функциональными возможностями, упрощающими создание сервлета. Например, он предлагает версии методов `init()` и `destroy()`, которые можно использовать "как есть". Вам нужно будет позаботиться только о методе `service()`.

Внутри класса `HelloServlet` осуществляется переопределение метода `service()` (унаследованного от класса `GenericServlet`). Этот метод обрабатывает запросы от клиента. Обратите внимание, что первым аргументом является объект `ServletRequest`. Благодаря ему сервлет будет считывать данные, предоставленные в запросе клиента. Вторым аргументом является объект `ServletResponce`. С его помощью сервлет сможет сформировать отклик клиенту.

При вызове метода `setContentType()` определяется тип MIME для HTTP-отклика. В этой программе типом MIME является text/html. Он показывает, что браузеру необходимо интерпретировать содержимое в качестве исходного HTML-кода.

Метод `getWriter()` получает `PrintWriter`. Все, что будет передаваться в поток, будет посылаться клиенту как часть HTTP-отклика. Метод `println()` используется для записи некоторого простого исходного HTML-кода в качестве HTTP-отклика.

Скомпилируйте этот исходный код и поместите файл `HelloServlet.class` в соответствующий каталог Tomcat, руководствуясь описанием из предыдущего раздела. Кроме того, добавьте `HelloServlet` в файл `web.xml`, как было сказано ранее.

Запуск Tomcat

Запустите Tomcat, выполнив приведенные выше действия. Tomcat должен функционировать до того, как вы захотите выполнить сервлет.

Запуск Web-браузера и запрос сервлета

Запустите Web-браузер и введите следующий URL-адрес:

```
http://localhost:8080/servlets-examples/servlet/HelloServlet
```

В качестве альтернативы можно ввести такой URL-адрес:

```
http://127.0.0.1:8080/servlets-examples/servlet/HelloServlet
```

Этот URL-адрес допустим, поскольку 127.0.01 определен как IP-адрес локального компьютера.

В окне браузера вы сможете увидеть выходные данные сервлета. В нем будет содержаться строка `Hello!` в полужирном начертании.

Servlet API

Классы и интерфейсы, необходимые для построения сервлетов, содержатся в двух пакетах: `javax.servlet` и `javax.servlet.http`. Они образуют Servlet API. Имейте в виду, что эти пакеты не являются частью ключевых пакетов Java. Наоборот, они являются стандартными расширениями, предлагаемыми Tomcat. Таким образом, они не входят в состав Java SE 6.

Servlet API находится в состоянии разработки и усовершенствования. Текущей спецификацией сервлета является версия 2.4, и именно она используется в примерах из данной книги. Однако поскольку в мире Java все постоянно меняется, поинтересуйтесь, не появились ли какие-то дополнения или видоизменения. В этой главе обсуждается ядро Servlet API, которое будет доступно большинству пользователей.

Пакет `javax.servlet`

Пакет `javax.servlet` содержит множество интерфейсов и классов, формирующих каркас, в рамках которого функционируют сервлеты. В табл. 31.1 представлены ключевые интерфейсы, предлагаемые в этом пакете. Самым главным из них является интерфейс `Servlet`. Все сервлеты должны реализовывать этот интерфейс или расширять класс, реализующий этот интерфейс. Интерфейсы `ServletRequest` и `ServletResponce` также являются очень важными.

Таблица 31.1. Ключевые интерфейсы пакета `javax.servlet`

Интерфейс	Описание
`Servlet`	Объявляет жизненный цикл методов для сервлета.
`ServletConfig`	Позволяет сервлетам получать параметры инициализации.
`ServletContext`	Позволяет сервлетам регистрировать события и обращаться к информации об их среде.
`ServletRequest`	Используется для чтения данных из запроса клиента.
`ServletResponse`	Используется для записи данных в отклик клиенту.

В табл. 31.2 перечислены ключевые классы пакета `javax.servlet`.

Таблица 31.2. Ключевые классы пакета `javax.servlet`

Класс	Описание
`GenericServlet`	Реализует интерфейсы `Servlet` и `ServletConfig`.
`ServletInputStream`	Предоставляет поток ввода для чтения запросов клиента.
`ServletOutputStream`	Предоставляет поток вывода для записи откликов клиенту.
`ServletException`	Указывает на то, что произошла ошибка сервлета.
`UnavailableException`	Указывает на то, что сервлет является недоступным.

Сейчас мы поговорим об этих интерфейсах и классах более подробно.

Интерфейс `Servlet`

Все сервлеты должны реализовывать интерфейс `Servlet`. В нем объявлены методы `init()`, `service()` и `destroy()`, которые вызываются сервером во время жизненного цикла сервлета. Кроме них предлагается также метод, позволяющий сервлету получать любые параметры инициализации. Методы, определяемые интерфейсом `Servlet`, перечислены в табл. 31.3.

Таблица 31.3. Методы, определенные интерфейсом `Servlet`

Метод	Описание
`void destroy()`	Вызывается при разгрузке сервлета.
`ServletConfig getServletConfig()`	Возвращает объект `ServletConfig`, содержащий любые параметры инициализации.
`String getServletInfo()`	Возвращает строку, описывающую сервлет.
`void init(ServletConfig sc) throws ServletException`	Вызывается во время инициализации сервлета. Параметры инициализации для сервлета могут быть получены из параметра *sc*. Если сервлет невозможно инициализировать, генерируется исключение `UnavailableException`.
`void service(ServletRequest req, ServletResponse res) throws ServletException, IOException`	Вызывается для обработки запроса клиента. Запрос клиента можно прочитать из *req*. Отклик клиенту можно записать в *res*. В случае возникновения ошибок сервлета или ошибок ввода-вывода 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, фамилия автора, номер версии, дата выпуска, авторские права). Этот метод также вызывается сервером.

Интерфейс `ServletConfig`

Интерфейс `ServletConfig` позволяет сервлету получать данные о конфигурации во время его загрузки. В табл. 31.4 перечислены методы, объявляемые этим интерфейсом.

Таблица 31.4. Методы, определенные интерфейсом `ServletConfig`

Метод	Описание
`ServletContext getServletContext()`	Возвращает содержимое для данного сервлета.
`String getInitParameter(String param)`	Возвращает значение параметра инициализации (*param*).
`Enumeration getInitParameterNames()`	Возвращает перечень имен параметров инициализации.
`String getServletName()`	Возвращает имя вызывающего сервлета.

Интерфейс `ServletContext`

Интерфейс `ServletContext` позволяет сервлетам получать информацию об их среде. Некоторые его методы представлены в табл. 31.5.

Таблица 31.5. Методы, определенные интерфейсом `ServletContext`

Метод	Описание
`Object getAttribute(String attr)`	Возвращает значение атрибута сервера, указанного в параметре *attr*.
`String getMimeType(String file)`	Возвращает тип MIME файла.
`String getRealPath(String vpath)`	Возвращает реальный путь, соответствующий виртуальному пути.
`String getServerInfo()`	Возвращает информацию о сервере.
`void log(String s)`	Записывает строку, указанную в соответствующем параметре, в журнал сервлета.
`void log(String s, Throwable e)`	Записывает строку и трассировку стека для исключения в журнал сервлета.
`void setAttribute(String attr, Object val)`	Присваивает заданному атрибуту указанное значение.

Интерфейс `ServletRequest`

Интерфейс `ServletRequest` позволяет сервлетам получать информацию о запросе клиента. Некоторые его методы представлены в табл. 31.6.

Таблица 31.6. Методы, определенные интерфейсом `ServletRequest`

Метод	Описание
`Object getAttribute(String attr)`	Возвращает значение указанного атрибута.
`String getCharacterEncoding()`	Возвращает схему кодировки символов в запросе.
`int getContentLength()`	Возвращает размер запроса. Если размер невозможно определить, возвращается значение `-1`.
`String getContentType()`	Возвращает тип запроса. Если тип невозможно определить, возвращается значение `null`.
`ServletInputStream getInputStream() throws IOException`	Возвращает `ServletInputStream`, который можно использовать для чтения двоичных данных в запросе. Если метод `getReader()` уже был вызван для этого запроса, генерируется исключение `IllegalStateException`.
`String getParameter(String pname)`	Возвращает значение указанного параметра.
`Enumeration getParameterNames()`	Возвращает перечень имен параметров для данного запроса.
`String[ ] getParameterValues(String name)`	Возвращает массив, состоящий из значений, связанных с параметром *name*.
`String getProtocol()`	Возвращает описание протокола.
`BufferedReader getReader() throws IOException`	Возвращает буферизированный читатель, который можно использовать для чтения текста из запроса. Если метод `getInputStream()` уже был вызван для данного запроса, генерируется исключение `IllegalStateException`.

Окончание табл. 31.6

Метод	Описание
`String getRemoteAddr()`	Возвращает строковый эквивалент IP-адреса клиента.
`String getRemoteHost()`	Возвращает строковый эквивалент имени хоста клиента.
`String getScheme()`	Возвращает схему передачи URL-адреса, которая используется для запроса (например, “http”, “ftp”).
`String getServerName()`	Возвращает имя сервера.
`int getServerPort()`	Возвращает номер порта.

Интерфейс `ServletResponce`

Интерфейс `ServletResponce` позволяет сервлету формулировать отклик клиенту. Некоторые его методы описаны в табл. 31.7.

Таблица 31.7. Методы, определенные интерфейсом `ServletResponce`

Метод	Описание
`String getCharacterEncoding()`	Возвращает схему кодировки символов в отклике.
`ServletOutputStream getOutputStream() throws IOException`	Возвращает `ServletOutputStream`, который можно использовать для записи двоичных данных в отклик. Если метод `gerWriter()` уже был вызван для данного запроса, генерируется исключение `IllegalStateException`.
`PrintWriter getWriter() throws IOException`	Возвращает `PrintWriter`, который можно использовать для записи символьных данных в отклик. Если метод `gerOutputStream()` уже был вызван для данного запроса, генерируется исключение `IllegalStateException`.
`void setContentLength(int size)`	Задает размер содержимого для отклика.
`void setContentType(String type)`	Задает тип содержимого для отклика.

Класс `GenericServlet`

Класс `GenericServlet` предлагает реализации основных методов жизненного цикла для сервлета. Класс `GenericServlet` реализует интерфейсы `Servlet` и `ServletConfig`. Кроме того, доступен также и метод для добавления строки в журнал сервера. Ниже показаны сигнатуры этого метода:

```
void log(String s)
void log(String s, Throwable e)
```

Здесь *s* — это строка, которую необходимо добавить в журнал, а *e* — это возникшее исключение.

Класс `ServletInputStream`

Класс `ServletInputStream` расширяет класс `InputStream`. Он реализуется контейнером сервлета и предлагает входной поток, который разработчик сервлета может ис-

пользовать для чтения данных из запроса клиента. Он определяет конструктор, применяемый по умолчанию. Кроме этого, предлагается метод для чтения байтов из потока. Ниже показана его сигнатура:

```
int readLine(byte[] buffer, int offset, int size) throws IOException
```

Здесь *buffer* — это массив, в котором хранится определенное количество (*size*) байт, начиная со смещения (*offset*). Метод возвращает фактическое количество прочитанных байт или –1, если возникнет условие окончания потока.

Класс ServletOutputStream

Класс `ServletOutputStream` расширяет класс `OutputStream`. Он реализуется контейнером сервлета и предлагает выходной поток, который разработчик сервлета может применять для записи данных в отклик клиенту. Определяется конструктор, используемый по умолчанию. Он также определяет методы `print()` и `println()`, которые выводят данные в поток.

Классы ServletException

Пакет `javax.servlet` определяет два исключения. Первым из них является `ServletException`, которое свидетельствует о возникновении ошибки сервлета. Вторым является `UnavailableException`, который расширяет `ServletException`. Это исключение свидетельствует о том, что сервлет является недоступным.

Чтение параметров сервлета

Интерфейс `ServletRequest` включает методы, позволяющие считывать имена и значения параметров, включенных в запрос клиента. Мы займемся созданием сервлета, который продемонстрирует их использование. Пример содержит два файла. Web-страница определена в файле `PostParameters.htm`, а сервлет — в файле `PostParametersServlet.java`.

В следующем листинге приведен исходный HTML-код файла `PostParameters.htm`. Этот код определяет таблицу, состоящую из двух меток и двух текстовых полей. Одной из меток является Employee (Сотрудник), а другой — Phone (Телефон). Имеется также кнопка подтверждения. Обратите внимание, что параметр `action` дескриптора формы (`<form>`) определяет URL-адрес. URL-адрес идентифицирует сервлет, который будет выполнять обработку HTTP-запроса POST.

```
<html>
<body>
<center>
<form name="Form1"
  method="post"
  action="http://localhost:8080/servlets-examples/servlet/PostParametersServlet">
<table>
<tr>
  <td><B>Employee</td>
  <td><input type=textbox name="e" size="25" value=""></td>
</tr>
<tr>
  <td><B>Phone</td>
  <td><input type=textbox name="p" size="25" value=""></td>
</tr>
</table>
```

```
<input type=submit value="Submit">
</body>
</html>
```

В следующем листинге представлен исходный код файла `PostParametersServlet.java`. Метод `service()` переопределяется с целью обработки запросов клиентов. Метод `getParameterNames()` возвращает перечень имен параметров. Их обработка осуществляется в цикле. Как видите, клиенту выводится имя параметра и его значение. Получение значения параметра производится с помощью метода `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 {

    // Получаем класс PrintWriter.
    PrintWriter pw = response.getWriter();

    // Получаем перечень имен параметров.
    Enumeration e = request.getParameterNames();

    // Отображаем 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 тестирования примера.

1. Запустите Tomcat (если это еще не сделано).
2. Отобразите Web-страницу в окне браузера.
3. Введите в текстовых полях фамилию служащего и номер телефона.
4. Отправьте Web-страницу.

После этого в окне браузера будет отображен отклик, динамически сгенерированный сервлетом.

Пакет `javax.servlet.http`

Пакет `javax.servlet.http` содержит некоторое количество интерфейсов и классов, которые чаще всего используются разработчиками сервлетов. Вы увидите, что их функциональные возможности облегчают построение сервлетов, работающих с HTTP-запросами и откликами.

В табл. 31.8 перечислены основные интерфейсы, которые предлагает этот пакет.

Таблица 31.8. Основные интерфейсы пакета `javax.servlet.http`

Интерфейс	Описание
`HttpServletRequest`	Позволяет сервлетам считывать данные из HTTP-запроса.
`HttpServletResponse`	Позволяет сервлетам записывать данные в HTTP-отклик.
`HttpSession`	Позволяет считывать и записывать данные сеансов.
`HttpSessionBindingListener`	Информирует объект о том, связан ли он с сеансом или нет.

В табл. 31.9 описаны основные классы, предлагаемые в этом пакете. Наиболее важным из них является класс `HttpServlet`. Разработчики сервлетов обычно расширяют этот класс в целях обработки HTTP-запросов.

Таблица 31.9. Основные классы пакета `javax.servlet.http`

Класс	Описание
`Cookie`	Позволяет хранить информацию о состоянии на компьютере клиента.
`HttpServlet`	Предлагает методы для обработки HTTP-запросов и откликов.
`HttpSessionEvent`	Инкапсулирует событие изменения сеанса.
`HttpSessionBindingEvent`	Показывает, связан или нет слушатель со значением сеанса, или показывает, что атрибут сеанса был изменен.

Интерфейс `HttpServletRequest`

Интерфейс `HttpServletRequest` реализуется контейнером сервлета. Он позволяет сервлету получать информацию о запросе клиента. В табл. 31.10 перечислены некоторые его методы.

Таблица 31.10. Методы, определенные в интерфейсе `HttpServletRequest`

Метод	Описание
`String getAuthType()`	Возвращает схему аутентификации.
`Cookie[] getCookies()`	Возвращает массив, содержащий cookie в данном запросе.
`long getDateHeader(String field)`	Возвращает значение поля заголовка даты.
`String getHeader(String field)`	Возвращает значение поля заголовка.
`Enumeration getHeaderNames()`	Возвращает перечень имен заголовков.
`int getIntHeader(String field)`	Возвращает целочисленный (`int`) эквивалент поля заголовка.
`String getMethod()`	Возвращает HTTP-метод для запроса.
`String getPathInfo()`	Возвращает любую информацию о маршруте, который определен после маршрута сервлета и перед строкой запроса в URL-адресе.

Окончание табл. 31.10

Метод	Описание
`String getPathTranslated()`	Возвращает любую информацию о маршруте, который определен после маршрута сервлета и перед строкой запроса в URL-адресе, после перевода его в действительный маршрут.
`String getQueryString()`	Возвращает любой строковый запрос в URL-адрес.
`String getRemoteUser()`	Возвращает имя пользователя, который сгенерировал данный запрос.
`String getRequestedSessionId()`	Возвращает идентификатор сеанса.
`String getRequestURI()`	Возвращает URL-адрес.
`StringBuffer getRequestURL()`	Возвращает URL-адрес.
`String getServletPath()`	Возвращает часть URL-адреса, которая идентифицирует сервлет.
`HttpSession getSession()`	Возвращает сеанс для данного запроса. Если сеанс не существует, он создается и затем возвращается.
`HttpSession getSession(Boolean` *`new`*`)`	Если значение *`new`* равно `true` и сеанса не существует, создает и возвращает сеанс для данного запроса. В противном случае возвращает существующий сеанс для данного запроса.
`boolean isRequestedSessionIdFromCookie()`	Возвращает `true`, если cookie содержит идентификатор сеанса. В противном случае возвращает `false`.
`boolean isRequestedSessionIdFromURL()`	Возвращает `true`, если URL-адрес содержит идентификатор сеанса. В противном случае возвращает `false`.
`boolean isRequestedSessionIdValid()`	Возвращает `true`, если запрошенный идентификатор сеанса является действительным в текущем содержимом сеанса.

Интерфейс `HttpServletResponse`

Интерфейс `HttpServletResponse` позволяет сервлету сформулировать для клиента HTTP-отклик. Определяются несколько констант. Они соответствуют кодам различных состояний, которые можно назначать HTTP-отклику. Например, `SC_OK` показывает, что HTTP-отклик достиг цели, а `SC_NOT_FOUND` показывает, что запрошенный ресурс является недоступным. В табл. 31.11 приводятся некоторые методы этого интерфейса.

Таблица 31.11. Методы, определенные в интерфейсе `HttpServletResponse`

Метод	Описание
`void addCookie(Cookie` *`cookie`*`)`	Добавляет *`cookie`* в HTTP-отклик.
`boolean containsHeader(String` *`field`*`)`	Возвращает `true`, если в заголовке HTTP-отклика содержится заданное поле.

Окончание табл. 31.11

Метод	Описание
`String encodeURL(String url)`	Определяет, должен ли идентификатор сеанса быть закодированным в URL-адресе. Если да, возвращает измененную версию URL-адреса. В противном случае возвращает URL-адрес. Все URL-адреса, сгенерированные сервером, должны обрабатываться этим методом.
`String encodeRedirectURL(String url)`	Определяет, должен ли идентификатор сеанса быть закодированным в URL-адресе. Если да, возвращает измененную версию URL-адреса. В противном случае возвращает URL-адрес. Все URL-адреса, переданные методу `sendRedirect()`, должны обрабатываться этим методом.
`void sendError(int c) throws IOException`	Посылает клиенту заданный код ошибки.
`void sendError(int c, String s) throws IOException`	Посылает клиенту заданный код ошибки и строку сообщения.
`void sendRedirect(String url) throws IOException`	Переадресовывает клиента по указанному URL-адресу.
`void setDateHeader(String field, long msec)`	Добавляет *field* в заголовок со значением даты, исчисляемой в миллисекундах (*msec*). Отсчет миллисекунд ведется с полуночи 1 января 1970 года (GMT).
`void setHeader(String field, String value)`	Добавляет *field* в заголовок со значением, указанным в параметре *value*.
`void setIntHeader(String field, int value)`	Добавляет *field* в заголовок со значением, указанным в параметре *value*.
`void setStatus(int code)`	Присваивает заданный код состоянию для данного отклика.

Интерфейс `HttpSession`

Интерфейс `HttpSession` позволяет сервлету считывать и записывать информацию о состоянии, связанную с HTTP-сеансом. Некоторые из его методов перечислены в табл. 31.12. Каждый из них генерирует исключение `IllegalStateException`, если сеанс уже является недействительным.

Таблица 31.12. Методы, определенные в интерфейсе `HttpSession`

Метод	Описание
`Object getAttribute(String attr)`	Возвращает значение, связанное с именем (параметр *attr*). Возвращает `null`, если указанный атрибут не был найден.
`Enumeration getAttributeNames()`	Возвращает перечень имен атрибутов, связанных с сеансом.
`long getCreationTime()`	Возвращает время, прошедшее с момента создания сеанса. Отсчет ведется в миллисекундах, начиная с полуночи 1 января 1970 года (GMT).
`String getId()`	Возвращает идентификатор сеанса.

Окончание табл. 31.12

Метод	Описание
`long getLastAccessedTime()`	Возвращает время, прошедшее с того момента, когда клиент произвел последний запрос для данного сеанса. Отсчет времени ведется в миллисекундах, начиная с полуночи 1 января 1970 года (GMT).
`void invalidate()`	Отменяет данный сеанс и удаляет его из содержимого.
`boolean isNew()`	Возвращает `true`, если сервер создал сеанс, к которому еще не обращался клиент.
`void removeAttribute(String attr)`	Удаляет из сеанса заданный атрибут.
`void setAttribute(String attr, Object val)`	Связывает заданное значение с именем заданного атрибута.

Интерфейс `HttpSessionBindingListener`

Интерфейс `HttpSessionBindingListener` реализуется объектами, для которых необходимо уведомление о том, являются ли они связанными или не связанными с HTTP-сеансом. Ниже перечислены методы, которые вызываются, если объект связан или, наоборот, не связан:

```
void valueBound(HttpSessionBindingEvent e)
void valueUnbound(HttpSessionBindingEvent e)
```

Здесь *e* — объект события, описывающий связывание.

Класс `Cookie`

Класс `Cookie` инкапсулирует cookie-набор. cookie-набор — это строки с данными, которые хранятся на компьютере клиента и содержат информацию о состоянии. cookie-наборы полезны для отслеживания активности пользователей. Например, предположим, что пользователь посещает онлайновый магазин. Cookie-набор может хранить имя пользователя, адрес и другую информацию. Пользователю не нужно будет вводить эти данные каждый раз при посещении магазина.

Сервлет может сохранить cookie-набор на компьютере пользователя с помощью метода `addCookie()` интерфейса `HttpServletResponce`. Данные для этого cookie-набора затем включаются в заголовок HTTP-отклика, который отправляется браузеру.

Имена и значения cookie-набора хранятся на компьютере пользователя. Часть информации, сохраняемой для каждого cookie-набор, включает следующие сведения:

- имя cookie-набора;
- значение cookie-набора;
- истечение срока действия cookie-набора;
- домен и путь cookie-набора.

Истечение срока действия определяет, когда данный cookie-набор будет удален из компьютера пользователя. Если эта дата не назначена явным образом, cookie-набор уда-

ляется по завершению текущего сеанса браузера. В противном случае он сохраняется в файле на компьютере пользователя.

Домен и путь cookie-набора определяют, когда он будет включен в заголовок HTTP-запроса. Если пользователь вводит URL-адрес, чей домен и путь совпадают с этими значениями, cookie-набор назначается Web-серверу, в противном случае — нет.

Для класса `Cookie` предусмотрен один конструктор. Он имеет следующую сигнатуру:

```
Cookie(String name, String value)
```

Здесь *name* и *value* cookie-набора передаются конструктору в качестве параметров. Методы класса `Cookie` перечислены в табл. 31.13.

Таблица 31.13. Методы, определяемые классом `Cookie`

Метод	Описание
`Object clone()`	Возвращает копию этого объекта.
`String getComment()`	Возвращает комментарий.
`String getDomain()`	Возвращает домен.
`int getMaxAge()`	Возвращает максимальный возраст (в секундах).
`String getName()`	Возвращает имя.
`String getPath()`	Возвращает маршрут.
`boolean getSecure()`	Возвращает `true` для безопасного cookie-набора. В противном случае возвращает `false`.
`String getValue()`	Возвращает значение.
`int getVersion()`	Возвращает версию.
`void setComment(String c)`	Присваивает заданный комментарий.
`void setDomain(String d)`	Присваивает заданный домен.
`void setMaxAge(int secs)`	Устанавливает максимальный возраст cookie-набора в секундах. Это значение представляет собой количество секунд, по истечении которых cookie-набор будет удален.
`void setPath(String p)`	Присваивает заданный путь.
`void setSecure(Boolean secure)`	Присваивает заданный флаг безопасности.
`void setValue(String v)`	Присваивает заданное значение.
`void setVersion(int v)`	Присваивает заданную версию.

Класс `HttpServlet`

Класс `HttpServlet` расширяет класс `GenericServlet`. Обычно он используется при разработке сервлетов, получающих и обрабатывающих HTTP-запросы. Методы класса `HttpServlet` представлены в табл. 31.14.

Таблица 31.14. Методы, определяемые классом `HttpServlet`

Метод	Описание
`void doDelete(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException`	Обрабатывает HTTP-запрос DELETE.
`void doGet(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException`	Обрабатывает HTTP-запрос GET.
`void doOptions(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException`	Обрабатывает HTTP-запрос OPTIONS.
`void doPost(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException`	Обрабатывает HTTP-запрос POST.
`void doPut(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException`	Обрабатывает HTTP-запрос PUT.
`void doTrace(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException`	Обрабатывает HTTP-запрос TRACE.
`Long getLastModified(HttpServletRequest req)`	Возвращает время, измеряемое в миллисекундах после полуночи 1 января 1970 года (GMT), когда в последний раз был изменен запрошенный ресурс.
`void service(HttpServletRequest req, HttpServletResponse res) throws IOException, ServletException`	Вызывается сервером, когда для данного сервлета поступает HTTP-запрос. Параметры предусматривают доступ к HTTP-запросу и отклику, соответственно.

Класс `HttpSessionEvent`

Класс `HttpSessionEvent` инкапсулирует события сеансов. Он расширяет класс `EventObject` и генерируется во время изменений, производимых в сеансе. В `HttpSessionEvent` определен следующий конструктор:

```
HttpSessionEvent(HttpSession session)
```

Здесь *session* — это источник события.

Класс `HttpSessionEvent` определяет один метод, `getSession()`, который показан ниже:

```
HttpSession getSession()
```

Он возвращает сеанс, в котором произошло событие.

Класс `HttpSessionBindingEvent`

Класс `HttpSessionBindingEvent` расширяет класс `HttpSessionEvent`. Он генерируется в том случае, когда слушатель связан или не связан со значением в объекте `HttpSession`. Он генерируется также, если атрибут является связанным или не связанным. Ниже показаны его конструкторы:

```
HttpSessionBindingEvent(HttpSession session, String name)
HttpSessionBindingEvent(HttpSession session, String name, Object val)
```

Здесь *session* — это источник события, а *name* — имя объекта, связанного или не связанного. Если параметр связан или не связан, ему передается заданное значение.

Метод `getName()` получает имя, связанное или не связанное. Ниже показан его конструктор:

```
String getName()
```

Показанный ниже метод `getSession()` получает сеанс, с которым связан или не связан слушатель:

```
HttpSession getSession()
```

Метод `getValue()` получает значение параметра, связанного или не связанного:

```
Object getValue()
```

Обработка HTTP-запросов и откликов

Класс `HttpServlet` предлагает специализированные методы, обрабатывающие различные типы HTTP-запросов. Разработчики сервлетов обычно переопределяют один из этих методов. К этим методам относятся `doDelete()`, `doGet()`, `doHead()`, `doOptions()`, `doPost()`, `doPut()` и `doTrace()`. Привести полное описание различных типов HTTP-запросов в этой книге невозможно. Однако чаще всего при обработке форм используются запросы GET и POST, потому в этом разделе приводятся примеры этих случаев.

Обработка HTTP-запросов GET

Сейчас мы займемся разработкой сервлета, обрабатывающего HTTP-запрос GET. Сервлет вызывается при подтверждении заполнения формы на Web-странице. В примере задействованы два файла. Web-страница определена в файле `ColorGet.htm`, а сервлет — в файле `ColorGetServlet.java`. Исходный HTML-код файла `ColorGet.htm` показан в следующем листинге. Он определяет форму, содержащую элемент выбора и кнопку для оправки. Обратите внимание, что параметр `action` дескриптора `<form>` определяет URL-адрес. URL-адрес идентифицирует сервлет для обработки HTTP-запроса GET.

```
<html>
<body>
<center>
<form name="Form1"
  action="http://localhost:8080/servlets-examples/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>
```

В следующем листинге показан исходный код файла ColorGetServlet.java. Метод doGet() переопределяется для обработки любых HTTP-запросов GET, посылаемых данному сервлету. Он использует метод getParameter() интерфейса HttpServletRequest для получения выбора, произведенного пользователем. После этого формируется отклик.

```
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();
  }
}
```

Скомпилируйте сервлет. Затем скопируйте его в соответствующий каталог и обновите файл web.xml, о чем было сказано ранее. Далее выполните следующие действия для тестирования этого примера.

1. Запустите Tomcat, если он еще не работает.
2. Отобразите Web-страницу в окне браузера.
3. Выберите цвет.
4. Отправьте форму на Web-странице.

После этого в окне браузера отобразит отклик, динамически сгенерированный сервлетом.

Еще один момент: параметры для HTTP-запроса GET включены как часть URL-адреса, посылаемого Web-серверу. Предположим, что пользователь выбрал красный цвет и отправил форму. URL-адрес, который посылается серверу из браузера, будет иметь следующий вид:

```
http://localhost:8080/servlets-examples/servlet/ColorGetServlet?color=Red
```

Символы, стоящие справа от вопросительного знака, называются *строкой запроса*.

Обработка HTTP-запросов POST

Теперь перейдем к разработке сервлета, обрабатывающего HTTP-запрос POST. Сервлет вызывается при подтверждении формы на Web-странице. В примере задействовано два файла. Web-страница определена в файле ColorPost.htm, а сервлет — в файле ColorPostServlet.java.

В следующем листинге показан исходный HTML-код в файле ColorPost.htm. Он идентичен коду файла ColorGet.htm 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/servlets-examples/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>
```

В следующем листинге показан исходный код файла `ColorPostServlet.java`. Метод `doPost()` заменяется с целью обработки любых HTTP-запросов POST, отправляемых данному сервлету. Он использует метод `getParameter()` интерфейса `HttpServletRequest` для получения выбора, произведенного пользователем. После этого формируется отклик.

```
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 сервлет. Чтобы протестировать его, выполните те же действия, что и в предыдущем разделе.

На заметку! *Параметры для HTTP-запроса POST не включаются в URL-адрес, отправляемый Web-серверу. В этом примере браузер посылает серверу следующий URL-адрес:* `http://localhost:8080/servlets-examples/servlet/ColorPostServlet`. *Имена параметров и значения отправляются в теле HTTP-запроса.*

Использование cookie-наборов

Теперь давайте разработаем сервлет, который поможет проиллюстрировать процесс использования cookie-наборов. Сервлет вызывается при отправке формы на Web-странице. Пример содержит три файла, описанных в табл. 31.15.

Таблица 31.15. Файлы примера применения cookie-наборов

Файл	Описание
`AddCookie.htm`	Позволяет пользователю определять значение для cookie-набора по имени `MyCookie`.
`AddCookieServlet.java`	Обрабатывает отправку `AddCookie.htm`.
`GetCookiesServlet.java`	Отображает значения cookie-набора.

В следующем листинге показан исходный HTML-код файла `AddCookie.htm`. Эта страница содержит текстовое поле для ввода значения. На странице имеется кнопка отправки. Если щелкнуть на этой кнопке, значение в текстовом поле будет отправлено `AddCookieServlet` посредством HTTP-запроса POST.

```
<html>
<body>
<center>
<form name="Form1"
  method="post"
  action="http://localhost:8080/servlets-examples/servlet/
AddCookieServlet">
<B>Enter a value for MyCookie:</B>
<input type=textbox name="data" size=25 value="">
<input type=submit value="Submit">
</form>
</body>
</html>
```

В следующем листинге показан исходный код файла `AddCookieServlet.java`. Он получает значение параметра по имени `data`. Затем он создает объект `Cookie` с именем `MyCookie`, который содержит значение параметра `data`. После этого в заголовок HTTP-отклика добавляется cookie-набор с помощью метода `addCookie()`. Далее браузер получает сообщение обратной связи.

```
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 {

    // Получение параметра от HTTP-запроса.
    String data = request.getParameter("data");

    // Создание cookie-набора.
    Cookie cookie = new Cookie("MyCookie", data);
```

```
    // Добавление cookie-набора в HTTP-отклик.
    response.addCookie(cookie);

    // Запись вывода в браузер.
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.println("<B>MyCookie has been set to");
    pw.println(data);
    pw.close();
  }
}
```

В следующем листинге показан исходный код файла `GetCookieServlet.java`. Он вызывает метод `getCookie()` для чтения любых cookie-наборов, включенных в HTTP-запрос GET. Имена и значения этих cookie-наборов включаются в HTTP-отклик. Обратите внимание, что для получения этой информации 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 {

    // Получение cookie-наборов из заголовка HTTP-запроса.
    Cookie[] cookies = request.getCookies();

    // Отображение всех cookie-наборов.
    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 упоминалось ранее. После этого выполните следующие действия, чтобы протестировать данный пример.

1. Запустите Tomcat, если он еще не работает.
2. Отобразите `AddCookie.htm` в окне браузера.
3. Введите значение для `MyCookie`.
4. Отправьте форму на Web-странице.

После выполнения этих примеров вы увидите, что в окне браузера будет отображено сообщение обратной связи.

В строке адреса браузера введите следующий URL-адрес:

```
http://localhost:8080/servlets-examples/servlet/GetCookiesServlet
```

Обратите внимание, что в окне браузера отображаются имя и значение cookie-набора.

В этом примере не применяется метод `setMaxAge()` класса `Cookie` для явного назначения истечения срока действия cookie-набора. Поэтому срок действия cookie-набора истекает по завершении сеанса браузера. Если использовать метод `setMaxAge()`, то вы увидите, что cookie-набор будет сохранен на диске клиентского компьютера.

Отслеживание сеансов

Протокол HTTP не поддерживает состояние. Каждый запрос не зависит от предыдущего запроса. Однако в некоторых приложениях бывает необходимо сохранять информацию о состоянии, которая потом будет анализироваться после общения между собой браузера и сервера. Такой механизм предлагают сеансы.

Сеанс можно создать с помощью метода `getSession()` интерфейса `HttpServletRequest`. Возвращаемым объектом является `HttpSession`. Этот объект способен сохранять набор связей, связывающих имена с объектами. Этими связями управляют методы `setAttribute()`, `getAttribute()`, `getAttributeNames()` и `removeAttribute()` интерфейса `HttpSession`. Важно отметить, что состояние сеанса распределяется между всеми сервлетами, связанными с определенным клиентом.

Следующий сервлет иллюстрирует использование информации о состоянии сеанса. Метод `getSession()` получает текущий сеанс. Если сеанс не существует, создается новый сеанс. Метод `getAttribute()` вызывается для получения объекта, связанного с именем `date`. Этим объектом является объект `Date`, инкапсулирующий дату и время последнего доступа к данной странице. (Естественно, такой связи не будет, если доступ к странице производится в первый раз.) Затем создается объект `Date`, инкапсулирующий текущую дату и время. Метод `setAttribute()` вызывается для связывания имени `date` с этим объектом.

```
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 {

    // Получение объекта HttpSession.
    HttpSession hs = request.getSession(true);

    // Получение класса PrintWriter.
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter();
    pw.print("<B>");

    // Отображение даты/времени последнего доступа.
    Date date = (Date)hs.getAttribute("date");
    if(date != null) {
      pw.print("Last access: " + date + "<br>");
    }
```

```
      // Отображение текущей даты/времени.
      date = new Date();
      hs.setAttribute("date", date);
      pw.println("Current date: " + date);
    }
  }
```

Когда вы впервые запросите этот сервлет, браузер отобразит одну строку с информацией о текущей дате и времени. При последующем вызове будут отображаться две строки. В первой строке будет указана дата и время последнего доступа к сервлету, а во второй — текущая дата и время.

IV ЧАСТЬ

Применения Java

32
ГЛАВА

Финансовые аплеты и сервлеты

Помимо больших и сложных приложений, к числу которых относятся текстовые процессоры, базы данных и пакеты программ бухгалтерского учета, и которые доминируют в мире вычислений, существует класс программ, которые являются одновременно и популярными, и небольшими. Они предназначены для выполнения различных финансовых расчетов: регулярных платежей по ссуде, будущей стоимости вклада, остатка баланса по ссуде. Ни один из этих расчетов не является сложным и не требует множества строк кода, а получаемая с их помощью информация является очень полезной.

Как вы знаете, язык Java изначально предназначался для поддержки создания небольших переносимых программ. Сначала эти программы принимали форму аплетов, однако спустя несколько лет появились сервлеты. (Напомним, что *аплеты* выполняются на локальном компьютере, внутри браузера, а *сервлеты* функционируют на сервере.) Большинство обычных финансовых расчетов, вследствие их небольших размеров, удобно производить в сервлетах и аплетах. Более того, если финансовый аплет/сервлет добавить на Web-страницу, то пользователи наверняка сочтут это удобным. Они будут регулярно посещать страницу, на которой можно произвести необходимый расчет.

В этой главе мы займемся разработкой некоторых аплетов, производящих следующие финансовые расчеты:

- регулярные платежи по ссуде;
- остаток баланса по ссуде;
- будущая стоимость вклада;
- первоначальная сумма вклада, необходимая для достижения желаемой будущей стоимости;
- годовой доход по вкладу;
- сумма вклада, необходимая для достижения желаемого годового дохода.

В конце главы будет показан способ преобразования финансовых аплетов в сервлеты.

Расчет платежей по ссуде

Пожалуй, самым распространенным финансовым расчетом является расчет регулярных платежей по ссуде, выданной, например, на покупку автомобиля или жилого дома. Расчет платежей по ссуде производится с помощью следующей формулы:

```
Payment = (intRate * (principal / payPerYear)) /
          (1 – ((intRate / payPerYear) + 1)^-payPerYear*numYears)
```

Здесь *intRate* — процент по ссуде, *principal* — первоначальный баланс, *payPerYear* — количество платежей в течение одного года, а *numYear* — срок погашения ссуды в годах.

Следующий аплет, называемый `RegPay`, использует предыдущую формулу для расчета платежей по ссуде с учетом информации, введенной пользователем. Как и остальные аплеты в этой главе, `RegPay` основан на Swing. Это означает, что он расширяет класс `JApplet` и использует классы Swing для создания пользовательского интерфейса. Обратите внимание также на то, что этот класс реализует интерфейс `ActionListener`.

```
// Простой аплет для расчетов по ссуде.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.text.*;
/*
  <applet code="RegPay" width=320 height=200>
  </applet>
*/

public class RegPay extends JApplet
  implements ActionListener {

  JTextField amount Text, paymentText, periodText, rateText;
  JButton doIt;

  double principal;  // первоначальная сумма
  double intRate;    // процент по ссуде
  double numYears;   // срок погашения ссуды в годах

  /* Количество платежей в течение одного года. Можно сделать
     так, чтобы это значение задавал пользователь. */
  final int payPerYear = 12;

  NumberFormat nf;

  public void init() {
    try {
      SwingUtilities.invokeAndWait(new Runnable () {
        public void run() {
          makeGUI();  // инициализация GUI
        }
      });
    } catch(Exception exc) {
      System.out.println("Can't create because of "+ exc);
      // System.out.println("Невозможно создать из-за "+ exc);
    }
  }
```

```
// Устанавливаем и инициализируем GUI.
private void makeGUI() {

  // Используем сеточную компоновку.
  GridBagLayout gbag = new GridBagLayout();
  GridBagConstraints gbc = new GridBagConstraints();
  setLayout(gbag);

  JLabel heading = new
        JLabel("Compute Monthly Loan Payments");
  // JLabel heading = new
  //       JLabel("Вычисление ежемесячных платежей по ссуде");

  JLabel amountLab = new JLabel("Principal ");
  // JLabel amountLab = new JLabel("Начальный баланс ");
  JLabel periodLab = new JLabel("Years ");
  // JLabel periodLab = new JLabel("Количество лет ");
  JLabel rateLab = new JLabel("Interest Rate ");
  // JLabel rateLab = new JLabel("Процент по ссуде ");
  JLabel paymentLab = new JLabel("Monthly Payments ");
  // JLabel paymentLab = new JLabel("Ежемесячный платеж ");

  amountText = new JTextField(10);
  periodText = new JTextField(10);
  paymentText = new JTextField(10);
  rateText = new JTextField(10);

  // Поле платежа только для отображения.
  paymentText.setEditable(false);

  doIt = new JButton("Compute");
  // doIt = new JButton("Вычислить");

  // Определяем сетку.
  gbc.weighty = 1.0; // используем строку, вес которой равен 1
  gbc.gridwidth = GridBagConstraints.REMAINDER;
  gbc.anchor = GridBagConstraints.NORTH;
  gbag.setConstraints(heading, gbc);
  // Располагаем большинство компонентов справа.
  gbc.anchor = GridBagConstraints.EAST;

  gbc.gridwidth = GridBagConstraints.RELATIVE;
  gbag.setConstraints(amountLab, gbc);
  gbc.gridwidth = GridBagConstraints.REMAINDER;
  gbag.setConstraints(amountText, gbc);

  gbc.gridwidth = GridBagConstraints.RELATIVE;
  gbag.setConstraints(periodLab, gbc);
  gbc.gridwidth = GridBagConstraints.REMAINDER;
  gbag.setConstraints(periodText, gbc);

  gbc.gridwidth = GridBagConstraints.RELATIVE;
  gbag.setConstraints(rateLab, gbc);
  gbc.gridwidth = GridBagConstraints.REMAINDER;
  gbag.setConstraints(rateText, gbc);

  gbc.gridwidth = GridBagConstraints.RELATIVE;
  gbag.setConstraints(paymentLab, gbc);
  gbc.gridwidth = GridBagConstraints.REMAINDER;
  gbag.setConstraints(paymentText, gbc);

  gbc.anchor = GridBagConstraints.CENTER;
  gbag.setConstraints(doIt, gbc);
```

```
    // Добавляем все компоненты.
    add(heading);
    add(amountLab);
    add(amountText);
    add(periodLab);
    add(periodText);
    add(rateLab);
    add(rateText);
    add(paymentLab);
    add(paymentText);
    add(doIt);

    // Регистрируем на получение событий действий.
    amountText.addActionListener(this);
    periodText.addActionListener(this);
    rateText.addActionListener(this);
    doIt.addActionListener(this);

    // Создаем формат числа.
    nf = NumberFormat.getInstance();
    nf.setMinimumFractionDigits(2);
    nf.setMaximumFractionDigits(2);
  }

  /* Пользователь нажал <Enter> в текстовом поле или щелкнул
     на кнопке Compute. Отображаем результат, если все
     поля заполнены. */
  public void actionPerformed(ActionEvent ae) {
    double result = 0.0;

    String amountStr = amountText.getText();
    String periodStr = periodText.getText();
    String rateStr = rateText.getText();
    try {
      if(amountStr.length() != 0 &&
         periodStr.length() != 0 &&
         rateStr.length() != 0) {

        principal = Double.parseDouble(amountStr);
        numYears = Double.parseDouble(periodStr);
        intRate = Double.parseDouble(rateStr) / 100;

        result = compute();

        paymentText.setText(nf.format(result));
      }

      showStatus(""); // удаляем любое предыдущее сообщение об ошибке
      } catch (NumberFormatException exc) {
      showStatus("Invalid Data");
      // showStatus("Неверные данные");
      paymentText.setText("");
    }
  }
  // Расчет платежа по ссуде.
  double compute() {
    double numer;
    double denom;
    double b, e;

    numer = intRate * principal / payPerYear;
```

```
    e = -(payPerYear * numYears);
    b = (intRate / payPerYear) + 1.0;
    denom = 1.0 - Math.pow(b, e);
    return numer / denom;
  }
}
```

Аплет, созданный этой программой, показан на рис. 32.1. Чтобы проверить его в действии, введите сумму, полученную по ссуде, срок погашения ссуды и процент по ссуде. Предполагается, что платежи будут производиться ежемесячно. Как только информация будет введена, щелкните на кнопке Compute (Вычислить), чтобы рассчитать сумму ежемесячного платежа.

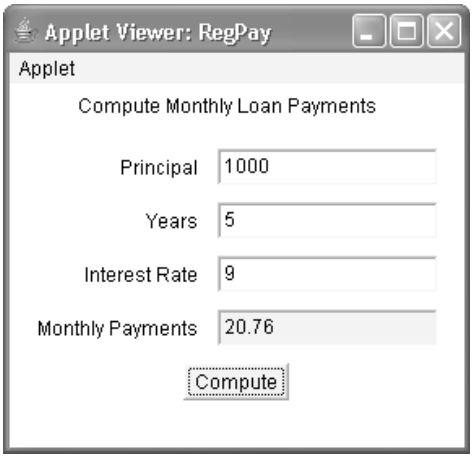

Рис. 32.1. Окно аплета `RegPay`

В следующих разделах подробно анализируется код `RegPay`. Поскольку все аплеты в этой главе используют одну и ту же базовую структуру, большая часть изложенного здесь объяснения применима и к другим аплетам в этой главе.

Поля `RegPay`

`RegPay` начинается с того, что объявляет переменные экземпляров, содержащие ссылки на текстовые поля, в которых пользователь будет вводить информацию о ссуде. Затем в нем объявляется переменная `doIt`, которая будет хранить ссылку на кнопку Compute.

Затем `RegPay` объявляет три переменных типа `double`, которые хранят значения по ссуде. Основная сумма ссуды хранится в переменной `principal`, процент по ссуде — в переменной `intRate`, а срок погашения ссуды (измеряется в годах) — в переменной `numYears`. Эти значения пользователь вводит в текстовых полях. После этого объявляется последняя `final`-переменная `payPerYear` целочисленного типа, которой присваивается значение 12. Таким образом, количество платежей в течение одного года получается жестко запрограммированным — 12 платежей, или ежемесячно в течение года, поскольку на практике большинство ссуд оформляется именно в такой манере. Как можно судить из комментариев, можно сделать так, чтобы пользователь самостоятельно мог вводить это значение, хотя для этого потребуется еще одно текстовое поле.

Последней переменной экземпляра, объявленной в RegPay, является nf, которая представляет собой ссылку на объект, имеющий тип NumberFormat, который будет описывать формат числа, используемого для вывода. NumberFormat хранится в пакете java.text. Хотя формат выходных чисел можно выбрать и другими способами (например, с помощью нового в Java класса Formatter), в данном случае удобно использовать именно NumberFormat, так как этот формат используется неоднократно, и его можно установить один раз, в самом начале программы. Хороший пример его использования можно продемонстрировать именно с помощью финансовых аплетов.

Метод init()

Как и во всех аплетах, метод init() вызывается, когда аплет впервые начинает свое выполнение. Этот метод просто вызывает метод makeGUI() в потоке диспетчеризации событий. Как было сказано в главе 29, аплеты, основанные на Swing, должны создаваться и взаимодействовать с компонентами GUI только посредством потока диспетчеризации событий.

Метод makeGUI()

Метод makeGUI() устанавливает пользовательский интерфейс для аплета. Он выполняет следующее:

1. Заменяет диспетчер компоновки на GridBagLayout.
2. Создает копии различных компонентов GUI.
3. Добавляет компоненты в сетку.
4. Добавляет слушателей событий компонентов.

Теперь давайте проанализируем строки метода makeGUI(). Он начинается следующими строками кода:

```
// Использование сеточной компоновки.
GridBagLayout gbag = new GridBagLayout();
GridBagConstraints gbc = new GridBagConstraints();
setLayout(gbag);
```

В этом фрагменте кода создается диспетчер компоновки GridBagLayout, который будет использоваться аплетом. (Более подробно о применении GridBagLayout рассказывалось в главе 24.) Использование диспетчера GridBagLayout объясняется тем, что с его помощью можно очень точно управлять размещением элементов управления в аплете.

Затем метод makeGUI() создает метки, текстовые поля и кнопку Compute, как показано ниже:

```
JLabel heading = new
     JLabel("Compute Monthly Loan Payments");

JLabel amountLab = new JLabel("Principal ");
JLabel periodLab = new JLabel("Years ");
JLabel rateLab = new JLabel("Interest Rate ");
JLabel paymentLab = new JLabel("Monthly Payments ");
amountText = new JTextField(10);
periodText = new JTextField(10);
paymentText = new JTextField(10);
rateText = new JTextField(10);
```

```
// Поле платежа только для отображения.
paymentText.setEditable(false);

doIt = new JButton("Compute");
```

Обратите внимание, что текстовому полю, отображающему ежемесячный платеж, устанавливается свойство, доступное только для чтения, посредством вызова `setEditable(false)`. В результате поле окрашивается серым цветом, и пользователь не может ввести в нем текст. Однако содержимое текстового поля можно установить с помощью вызова метода `setText()`. Таким образом, если редактирование запрещено в `JTextField`, поле можно использовать для отображения текста, и пользователь не сможет его изменить.

В следующем фрагменте кода определяются ограничения сетки для каждого компонента:

```
// Определяем сетку.
gbc.weighty = 1.0; // используем строку, вес которой равен 1
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.anchor = GridBagConstraints.NORTH;
gbag.setConstraints(heading, gbc);

// Располагаем большинство компонентов справа.
gbc.anchor = GridBagConstraints.EAST;

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(amountLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(amountText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(periodLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(periodText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(rateLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(rateText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(paymentLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(paymentText, gbc);

gbc.anchor = GridBagConstraints.CENTER;
gbag.setConstraints(doIt, gbc);
```

Хотя на первый взгляд этот фрагмент кода может показаться сложным, на самом деле это не так. Вы должны просто иметь в виду, что каждая строка в сетке определяется отдельно. Сейчас мы проанализируем работу этого фрагмента. Для начала вес каждой строки, содержащейся в `gbc.weighty`, получает значение 1. Это позволит равномерно распределить дополнительное пространство в сетке там, где имеется больше интервалов между строками, чем это необходимо для хранения компонента. Затем `gbc.gridwidth` присваивается `REMAINDER`, а `gbc.anchor` присваивается `NORTH`. Метка, на которую ссылается `heading`, добавляется в `gbag` посредством вызова метода `setConstraints()`. В этом фрагменте устанавливается положение заголовка `heading` в верхней части сетки (`NORTH` — север) и для него отводится оставшаяся часть строки. Таким образом, после выполнения этого фрагмента кода заголовок будет находиться в верхней части окна, оставаясь в то же время в строке.

Затем добавляются четыре текстовых поля и их метки. Сначала gbc.anchor присваивается EAST. В результате каждый компонент будет выровнен вправо.

Далее gbc.gridWidth присваивается RELATIVE и добавляется метка. После этого gbc.gridWidth присваивается REMAINDER и добавляется текстовое поле. Таким образом, каждая пара текстового поля и метки будет занимать одну строку. Этот процесс повторяется до тех пор, пока не будут добавлены все четыре пары текстового поля и метки. После всего этого кнопка Compute размещается в центре.

После того как будут определены ограничители сетки, следующий фрагмент кода добавляет в окно компоненты:

```
// Добавляем все компоненты.
add(heading);
add(amountLab);
add(amountText);
add(periodLab);
add(periodText);
add(rateLab);
add(rateText);
add(paymentLab);
add(paymentText);
add(doIt);
```

Затем регистрируются слушатели событий для трех текстовых полей ввода и кнопки Compute, как показано ниже:

```
// Регистрируем на получение событий действий.
amountText.addActionListener(this);
periodText.addActionListener(this);
rateText.addActionListener(this);
doIt.addActionListener(this);
```

В завершение мы получаем объект NumberFormat, а в качестве формата выбираются две десятичные цифры:

```
// Создаем формат числа.
nf = NumberFormat.getInstance();
nf.setMinimumFractionDigits(2);
nf.setMaximumFractionDigits(2);
```

При вызове фабричного метода getInstance() мы получаем объект NumberFormat, пригодный для размещения по умолчанию.

При вызове методов setMinimumFractionDigits() и setMaximumFractionDigits() определяются минимальное и максимальное количество отображаемых десятичных цифр. Так как минимальное и максимальное количество составляет 2, то видимыми всегда будут два десятичных знака.

Метод actionPerformed()

Метод actionPerformed() вызывается каждый раз при нажатии клавиши <ENTER>, когда курсор находится в текстовом поле, или при щелчке по кнопке Compute. Этот метод выполняет три основных функции: он получает введенную пользователем информацию о ссуде, вызывает метод compute() для поиска платежей по ссуде и отображает результат. Теперь давайте рассмотрим строки метода actionPerformed().

После объявления переменной result метод paint() получает строки из трех полей, в которых пользователь вводит свои данные, с помощью следующей последовательности:

```
String amountStr = amountText.getText();
String periodStr = periodText.getText();
String rateStr = rateText.getText();
```

Затем начинается блок try, в котором проверяется, действительно ли все три поля содержат информацию:

```
try {
  if(amountStr.length() != 0 &&
    periodStr.length() != 0 &&
    rateStr.length() != 0) {
```

Напомним, что пользователь должен ввести исходную сумму выданной ссуды, срок погашения ссуды в годах и процент по ссуде. Если каждое из этих полей будет содержать информацию, то длина каждой строки будет больше нуля.

Если пользователь ввел все данные по ссуде, то числовые значения, соответствующие этим строкам, извлекаются и сохраняются в соответствующей переменной экземпляра. Затем вызывается метод compute() для расчета платежа по ссуде, и результат отображается в текстовом поле, доступном только для чтения, на которое ссылается paymentText, как показано ниже:

```
principal = Double.parseDouble(amountStr);
numYears = Double.parseDouble(periodStr);
intRate = Double.parseDouble(rateStr) / 100;

result = compute();

paymentText.setText(nf.format(result));
```

Обратите внимание на вызов метода nf.format(result). Благодаря этому методу значение в переменной result будет иметь определенный ранее формат (с двумя десятичными цифрами) и будет возвращена результирующая строка. Эта строка впоследствии будет использоваться для установки текста в TextField, определяемого с помощью paymentText.

Если в одном из текстовых полей пользователь введет нечисловое значение, то метод Double.parseDouble() сгенерирует исключение NumberFormatException. Если это произойдет, в строке состояния будет отображено сообщение об ошибке, и текстовое поле Payment (Платеж) окажется пустым, как показано ниже:

```
    showStatus(""); // удаляем любое предыдущее сообщение об ошибке
  } catch (NumberFormatException exc) {
    showStatus("Invalid Data");
    paymentText.setText("");
    }
  }
```

В противном случае будет удалено любое отображенное ранее сообщение.

Метод compute()

Расчет платежа по ссуде производится с помощью метода compute(). Он реализует приведенную ранее формулу и работает со значениями переменных principal, intRate, numYears и payPerYear. Метод возвращает результат расчета.

На заметку! *Базовая структура, используемая в* `RegPay`*, применяется во всех аплетах, представленных в этой главе.*

Расчет будущей стоимости вклада

Еще один важный финансовый расчет позволяет определить будущую стоимость вклада на основании данных о первоначальной сумме вклада, норме прибыли, количестве периодов начисления сложного процента в течение одного года и количестве лет, на которые рассчитан вклад. Допустим, вы хотите узнать, сколько денег будет на вашем пенсионном счете через 12 лет, если на данный момент на нем имеется $98 000, а средняя годовая норма прибыли составляет 6 процентов. Для этих целей предназначен аплет `FutVal`.

Для расчета будущей стоимости вклада используется следующая формула:

$$\text{Future Value} = principal * ((rateOfRet / compPerYear) + 1)^{compPerYear * numYears}$$

Здесь *`rateOfRet`* определяет норму прибыли, *`principal`* — первоначальную сумму вклада, *`compPerYear`* — количество периодов начисления сложного процента в течение одного года, а *`numYears`* — срок вклада в годах. Если вы используете годовую норму прибыли для *`rateOfRet`*, то количество периодов начисления сложного процента в течение одного года составляет 1.

Аплет `FutVal` использует предыдущую формулу для расчета будущей стоимости вклада. Аплет, созданный этой программой, показан на рис. 32.2. Помимо различий в расчетах в методе `compute()`, этот аплет похож на аплет `RegPay`, который мы рассматривали в предыдущем разделе.

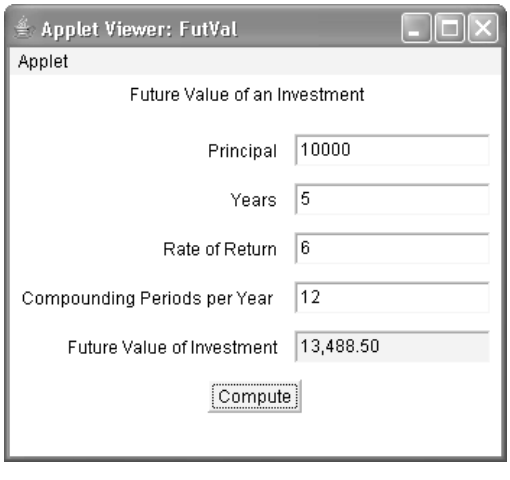

Рис. 32.2. Окно аплета `FutVal`

```
// Расчет первоначальной суммы вклада, необходимой для
// того, чтобы в будущем достичь требуемой стоимости вклада.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.text.*;
/*
```

```
  <applet code="FutVal" width=380 height=240>
  </applet>
*/

public class FutVal extends JApplet
  implements ActionListener {

  JTextField amountText, futvalText, periodText,
          rateText, compText;
  JButton doIt;

  double principal; // первоначальная стоимость
  double rateOfRet; // норма прибыли
  double numYears;  // срок вклада в годах
  int compPerYear;  // количество периодов начисления сложного
                // процента в течение одного года

  NumberFormat nf;

  public void init() {
    try {
      SwingUtilities.invokeAndWait(new Runnable () {
        public void run() {
          makeGUI(); // инициализация GUI
        }
      });
    } catch(Exception exc) {
      System.out.println("Can't create because of "+ exc);
      // System.out.println("Невозможно создать из-за "+ exc);
    }
  }

  // Устанавливаем и инициализируем GUI.
  private void makeGUI() {

    // Используем сеточную компоновку.
    GridBagLayout gbag = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(gbag);

    JLabel heading = new
          JLabel("Future Value of an Investment");
    // JLabel heading = new
    //       JLabel("Расчет будущей стоимости вклада");

    JLabel amountLab = new JLabel("Principal ");
    // JLabel amountLab = new JLabel("Начальная сумма ");
    JLabel periodLab = new JLabel("Years ");
    // JLabel periodLab = new JLabel("Количество лет ");
    JLabel rateLab = new JLabel("Rate of Return ");
    // JLabel rateLab = new JLabel("Норма прибыли ");
    JLabel futvalLab =
           new JLabel("Future Value of Investment ");
    // JLabel futvalLab =
    //        new JLabel("Будущая стоимость вклада ");
    JLabel compLab =
           new JLabel("Compounding Periods per Year ");
    // JLabel compLab =
    //    new JLabel("Количество периодов начисления сложного процента в год ");
```

```
amountText = new JTextField(10);
periodText = new JTextField(10);
futvalText = new JTextField(10);
rateText = new JTextField(10);
compText = new JTextField(10);

// Поле будущей стоимости, только для отображения.
futvalText.setEditable(false);

doIt = new JButton("Compute");
// doIt = new JButton("Вычислить");

// Определяем сетку.
gbc.weighty = 1.0; // используем строку, вес которой равен 1
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.anchor = GridBagConstraints.NORTH;
gbag.setConstraints(heading, gbc);

// Размещаем большинство компонентов справа.
gbc.anchor = GridBagConstraints.EAST;

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(amountLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(amountText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(periodLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(periodText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(rateLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(rateText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(compLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(compText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(futvalLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(futvalText, gbc);

gbc.anchor = GridBagConstraints.CENTER;
gbag.setConstraints(doIt, gbc);

add(heading);
add(amountLab);
add(amountText);
add(periodLab);
add(periodText);
add(rateLab);
add(rateText);
add(compLab);
add(compText);
add(futvalLab);
add(futvalText);
add(doIt);
```

```
    // Регистрируем на получение событий действий.
    amountText.addActionListener(this);
    periodText.addActionListener(this);
    rateText.addActionListener(this);
    compText.addActionListener(this);
    doIt.addActionListener(this);

    // Создаем формат числа.
    nf = NumberFormat.getInstance();
    nf.setMinimumFractionDigits(2);
    nf.setMaximumFractionDigits(2);
  }

  /* Пользователь нажал <Enter> в текстовом поле или
     щелкнул на кнопке Compute. Отображаем результат,
     если все поля заполнены. */
  public void actionPerformed(ActionEvent ae) {
    double result = 0.0;

    String amountStr = amountText.getText();
    String periodStr = periodText.getText();
    String rateStr = rateText.getText();
    String compStr = compText.getText();

    try {
      if(amountStr.length() != 0 &&
        periodStr.length() != 0 &&
        rateStr.length() != 0 &&
        compStr.length() != 0) {

        principal = Double.parseDouble(amountStr);
        numYears = Double.parseDouble(periodStr);
        rateOfRet = Double.parseDouble(rateStr) / 100;
        compPerYear = Integer.parseInt(compStr);

        result = compute();

        futvalText.setText(nf.format(result));
      }

      showStatus(""); // удаляем любое предыдущее сообщение об ошибке
    } catch (NumberFormatException exc) {
      showStatus("Invalid Data");
      // showStatus("Неверные данные");
      futvalText.setText("");
    }
  }

  // Расчет будущей стоимости.
  double compute() {
    double b, e;

    b = (1 + rateOfRet/compPerYear);
    e = compPerYear * numYears;

    return principal * Math.pow(b, e);
  }
}
```

Расчет первоначальной суммы вклада, необходимой для достижения будущей стоимости

Иногда бывает необходимо узнать о том, сколько денег нужно положить в банк, чтобы в будущем достичь определенной стоимости вклада. Например, если вы хотите с помощью вклада заработать деньги на обучение своих детей, и вы знаете, что для этого через 5 лет вам понадобится $75 000, то вам нужно рассчитать, сколько денег необходимо внести на счет сейчас, при условии, что процентная ставка составляет 7% в год. Для этого предназначен аплет `InitInv`.

Формула для расчета первоначальной суммы вклада выглядит следующим образом:

$$\text{Initial Investment} = targetValue / (((rateOfRet / compPerYear) + 1)^{compPerYear * numYears})$$

Здесь *`rateOfRet`* определяет норму прибыли, *`targetValue`* — первоначальный баланс, *`compPerYear`* — количество периодов начисления сложного процента в течение одного года, а *`numYears`* — срок вклада. Если вы используете годовую норму прибыли для *`rateOfRet`*, то количество периодов начисления сложного процента в течение одного года будет равно 1.

Следующий аплет, называемый `InitInv`, использует показанную выше формулу для расчета первоначальной суммы вклада, необходимой для того, чтобы в будущем достичь требуемой стоимости вклада. Аплет, созданный этой программой, показан на рис. 32.3.

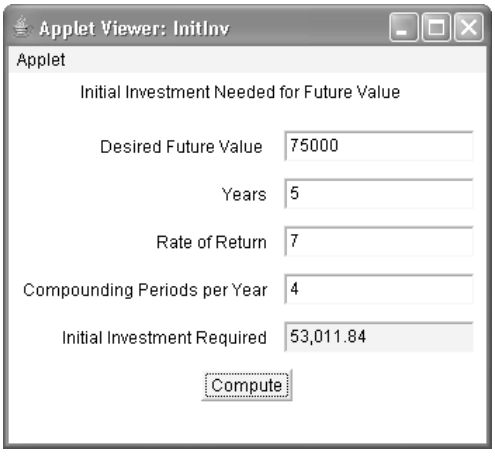

Рис. 32.3. Окно аплета `InitInv`

```
/* Расчет первоначальной суммы вклада, необходимой для
   получения желаемого ежегодного дохода. Другими словами,
   нужно рассчитать первоначальную сумму вклада, необходимую
   для того чтобы регулярно получать доход в виде процентов
   в течение определенного периода времени. */
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.text.*;
```

```
/*
  <applet code="InitInv" width=340 height=240>
  </applet>
*/
public class InitInv extends JApplet
  implements ActionListener {

  JTextField targetText, initialText, periodText,
           rateText, compText;
  JButton doIt;

  double targetValue;    // первоначальное значение targetValue
  double rateOfRet;      // норма прибыли
  double numYears;       // срок вклада
  int compPerYear;       // количество периодов начисления сложного
                         // процента в течение одного года

  NumberFormat nf;
  public void init() {
    try {
      SwingUtilities.invokeAndWait(new Runnable () {
        public void run() {
          makeGUI();       // инициализация GUI
        }
      });
    } catch(Exception exc) {
      System.out.println("Can't create because of "+ exc);
      // System.out.println("Невозможно создать из-за "+ exc);
    }
  }

  // Устанавливаем и инициализируем GUI.
  private void makeGUI() {
    // Используем сеточную компоновку.
    GridBagLayout gbag = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(gbag);

    JLabel heading = new
          JLabel("Initial Investment Needed for " +
                "Future Value");
    // JLabel heading = new
    //       JLabel("Расчет первоначальной суммы вклада " +
    //             "для достижения будущей стоимости");

    JLabel targetLab = new JLabel("Desired Future Value ");
    // JLabel targetLab = new JLabel("Желаемая будущая сумма ");
    JLabel periodLab = new JLabel("Years ");
    // JLabel periodLab = new JLabel("Количество лет ");
    JLabel rateLab = new JLabel("Rate of Return ");
    // JLabel rateLab = new JLabel("Норма прибыли ");
    JLabel compLab =
           new JLabel("Compounding Periods per Year ");
    // JLabel compLab =
    //    new JLabel("Количество периодов начисления сложного процента в год ");
    JLabel initialLab =
           new JLabel("Initial Investment Required ");
    // JLabel initialLab =
    //        new JLabel("Требуемая первоначальная сумма вклада ");
```

```
targetText = new JTextField(10);
periodText = new JTextField(10);
initialText = new JTextField(10);
rateText = new JTextField(10);
compText = new JTextField(10);

// Поле исходного значения, только для отображения.
initialText.setEditable(false);

doIt = new JButton("Compute");
// doIt = new JButton("Вычислить");

// Определяем сетку.
gbc.weighty = 1.0; // используем строку, вес которой равен 1

gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.anchor = GridBagConstraints.NORTH;
gbag.setConstraints(heading, gbc);

// Размещаем большинство компонентов справа.
gbc.anchor = GridBagConstraints.EAST;

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(targetLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(targetText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(periodLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(periodText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(rateLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(rateText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(compLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(compText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(initialLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(initialText, gbc);

gbc.anchor = GridBagConstraints.CENTER;
gbag.setConstraints(doIt, gbc);

// Добавляем все компоненты.
add(heading);
add(targetLab);
add(targetText);
add(periodLab);
add(periodText);
add(rateLab);
add(rateText);
add(compLab);
add(compText);
add(initialLab);
add(initialText);
add(doIt);
```

```
    // Регистрируем на получение событий.
    targetText.addActionListener(this);
    periodText.addActionListener(this);
    rateText.addActionListener(this);
    compText.addActionListener(this);
    doIt.addActionListener(this);

    // Создаем формат числа.
    nf = NumberFormat.getInstance();
    nf.setMinimumFractionDigits(2);
    nf.setMaximumFractionDigits(2);
  }

  /* Пользователь нажал <Enter> в текстовом поле или
     щелкнул на кнопке Compute. Отображаем результат,
     если заполнены все поля. */
  public void actionPerformed(ActionEvent ae) {
    double result = 0.0;

    String targetStr = targetText.getText();
    String periodStr = periodText.getText();
    String rateStr = rateText.getText();
    String compStr = compText.getText();

    try {
      if(targetStr.length() != 0 &&
         periodStr.length() != 0 &&
         rateStr.length() != 0 &&
         compStr.length() != 0) {

        targetValue = Double.parseDouble(targetStr);
        numYears = Double.parseDouble(periodStr);
        rateOfRet = Double.parseDouble(rateStr) / 100;
        compPerYear = Integer.parseInt(compStr);

        result = compute();

        initialText.setText(nf.format(result));
       }

       showStatus(""); // удаляем любое предыдущее сообщение об ошибке
     } catch (NumberFormatException exc) {
       showStatus("Invalid Data");
       // showStatus("Неверные данные");
       initialText.setText("");
     }
   }

   // Расчет требуемого первоначального вклада.
   double compute() {
     double b, e;

     b = (1 + rateOfRet/compPerYear);
     e = compPerYear * numYears;

     return targetValue / Math.pow(b, e);
   }
}
```

Расчет первоначальной суммы вклада, необходимой для получения желаемого годового дохода

Еще один распространенный финансовый расчет используется для определения денежной суммы, которую необходимо внести на счет, чтобы получать определенный годовой процент. Например, вы хотите, чтобы после выхода на пенсию вы получали по $5000 каждый месяц, и чтобы эти деньги можно было получать в течение 20 лет. Вопрос заключается в том, сколько денег необходимо внести на счет сейчас, чтобы обеспечить будущие процентные выплаты? Ответ можно получить с помощью следующей формулы:

```
Initial Investment = ((regWD * wdPerYear) / rateOfRet) *
                     (1 – (1 / (rateOfRet / wdPerYear + 1)^(wdPerYear * numYears)))
```

Здесь *rateOfRet* определяет норму прибыли, *regWD* — требуемый регулярный доход в виде процентов, *wdPerYear* — сколько раз в году вы будете получать доход в виде процентов, а *numYears* — в течение скольких лет вы планируете получать эти деньги.

Приведенный ниже код аплета `Annuity` рассчитывает сумму первоначального вклада, необходимую для получения желаемого ежегодного дохода. Аплет, созданный с помощью этой программы, показан на рис. 32.4.

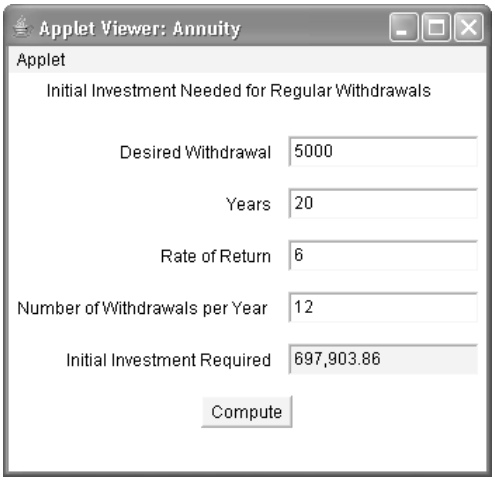

Рис. 32.4. Окно аплета `Annuity`

```
/* Расчет первоначальной суммы вклада, необходимой для
   получения желаемого ежегодного дохода. Другими словами,
   нужно рассчитать первоначальную сумму вклада, необходимую
   для регулярного получения дохода в виде процентов в течение
   определенного периода времени. */
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.text.*;
/*
```

```
  <applet code="Annuity" width=340 height=260>
  </applet>
*/

public class Annuity extends JApplet
  implements ActionListener {

  JTextField regWDText, initialText, periodText,
          rateText, numWDText;
  JButton doIt;
  double regWDAmount;    // доход, получаемый в виде процентов
  double rateOfRet;      // норма прибыли
  double numYears;       // срок вклада
  int numPerYear;        // сколько раз в году будет выплачиваться доход

  NumberFormat nf;

  public void init() {
    try {
      SwingUtilities.invokeAndWait(new Runnable () {
        public void run() {
          makeGUI();       // инициализация GUI
        }
      });
    } catch(Exception exc) {
      System.out.println("Can't create because of "+ exc);
      // System.out.println("Невозможно создать из-за "+ exc);
    }
  }

  // Устанавливаем и инициализируем GUI.
  private void makeGUI() {

    // Использование сеточной компоновки.
    GridBagLayout gbag = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(gbag);

    JLabel heading = new
         JLabel("Initial Investment Needed for " +
               "Regular Withdrawals");
    // JLabel heading = new
    //      JLabel("Расчет первоначальной суммы вклада для " +
    //            "желаемого ежегодного дохода");

    JLabel regWDLab = new JLabel("Desired Withdrawal ");
    // JLabel regWDLab = new JLabel("Желаемый ежегодный доход ");
    JLabel periodLab = new JLabel("Years ");
    // JLabel periodLab = new JLabel("Количество лет ");
    JLabel rateLab = new JLabel("Rate of Return ");
    // JLabel rateLab = new JLabel("Норма прибыли ");
    JLabel numWDLab =
           new JLabel("Number of Withdrawals per Year ");
    // JLabel numWDLab =
    //        new JLabel("Количество получений дохода в год ");
    JLabel initialLab =
           new JLabel("Initial Investment Required ");
    // JLabel initialLab =
    //        new JLabel("Требуемая первоначальная сумма вклада ");

    regWDText = new JTextField(10);
```

```
periodText = new JTextField(10);
initialText = new JTextField(10);
rateText = new JTextField(10);
numWDText = new JTextField(10);

// Поле первоначального вклада, только для отображения.
initialText.setEditable(false);

doIt = new JButton("Compute");
// doIt = new JButton("Вычислить");

// Определяем сетку.

gbc.weighty = 1.0; // используем строку, вес которой равен 1

gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.anchor = GridBagConstraints.NORTH;
gbag.setConstraints(heading, gbc);

// Размещаем большинство компонентов справа.
gbc.anchor = GridBagConstraints.EAST;

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(regWDLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(regWDText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(periodLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(periodText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(rateLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(rateText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(numWDLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(numWDText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(initialLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(initialText, gbc);

gbc.anchor = GridBagConstraints.CENTER;
gbag.setConstraints(doIt, gbc);

// Добавляем все компоненты.
add(heading);
add(regWDLab);
add(regWDText);
add(periodLab);
add(periodText);
add(rateLab);
add(rateText);
add(numWDLab);
add(numWDText);
add(initialLab);
add(initialText);
add(doIt);
```

```
    // Регистрируем на получение событий действий текстового поля.
    regWDText.addActionListener(this);
    periodText.addActionListener(this);
    rateText.addActionListener(this);
    numWDText.addActionListener(this);
    doIt.addActionListener(this);

    // Создание формата числа.
    nf = NumberFormat.getInstance();
    nf.setMinimumFractionDigits(2);
    nf.setMaximumFractionDigits(2);
  }

  /* Пользователь нажал <Enter> в текстовом поле или
     щелкнул на кнопке Compute. Отображаем результат,
     если заполнены все поля. */
  public void actionPerformed(ActionEvent ae) {
    double result = 0.0;

    String regWDStr = regWDText.getText();
    String periodStr = periodText.getText();
    String rateStr = rateText.getText();
    String numWDStr = numWDText.getText();

    try {
      if(regWDStr.length() != 0 &&
         periodStr.length() != 0 &&
         rateStr.length() != 0 &&
         numWDStr.length() != 0) {

        regWDAmount = Double.parseDouble(regWDStr);
        numYears = Double.parseDouble(periodStr);
        rateOfRet = Double.parseDouble(rateStr) / 100;
        numPerYear = Integer.parseInt(numWDStr);

        result = compute();

        initialText.setText(nf.format(result));
      }

      showStatus(""); // удаляем любое предыдущее сообщение об ошибке
    } catch (NumberFormatException exc) {
      showStatus("Invalid Data");
      // showStatus("Неверные данные");
      initialText.setText("");
    }
  }

  // Расчет требуемой суммы первоначального вклада.
  double compute() {
    double b, e;
    double t1, t2;

    t1 = (regWDAmount * numPerYear) / rateOfRet;

    b = (1 + rateOfRet/numPerYear);
    e = numPerYear * numYears;

    t2 = 1 - (1 / Math.pow(b, e));

    return t1 * t2;
  }
}
```

Нахождение максимального годового дохода для данной суммы вклада

Этот расчет используется для определения максимального годового дохода (регулярные выплаты в виде процентов), который можно получать для данной суммы вклада в течение определенного периода времени. Например, если на ваш пенсионный счет внесено $500 000, то нужно узнать, сколько денег вы сможете получать из этой суммы ежемесячно в течение 20 лет, если процентная ставка составляет 6%. Формула для вычисления максимального дохода показана ниже:

```
Maximum Withdrawal = principal * (((rateOfRet / wdPerYear) /
                     (-1 + ((rateOfRet / wdPerYear) + 1)^(wdPerYear * numYears))) +
                     (rateOfRet / wdPerYear))
```

Здесь *rateOfRet* определяет норму прибыли, *principal* — сумму первоначального вклада, *wdPerYear* — сколько раз в году будет осуществляться выплата по процентам, а *numYears* — на какой срок в годах рассчитан вклад.

Приведенный ниже код аплета `MaxWD` рассчитывает максимальные периодические суммы дохода в виде процентов, которые можно получать в течение указанного периода времени с учетом определенной нормы прибыли. Аплет, созданный этой программой, показан на рис. 32.5.

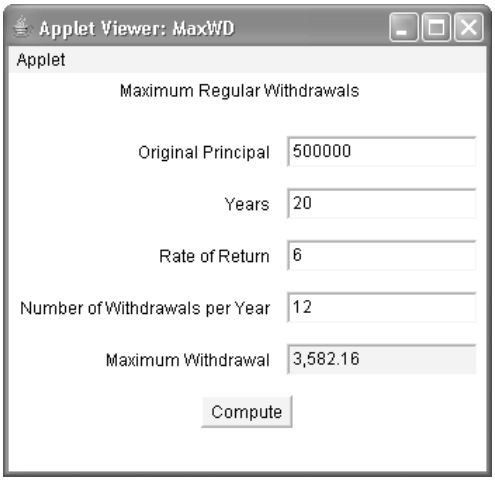

Рис. 32.5. Окно аплета `MaxWD`

```
/* Расчет максимального годового дохода, который можно получать
   в виде процентов по вкладу в течение определенного
   периода времени.  */
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.text.*;
/*
  <applet code="MaxWD" width=340 height=260>
```

```
  </applet>
*/
public class MaxWD extends JApplet
  implements ActionListener {

  JTextField maxWDText, orgPText, periodText,
           rateText, numWDText;
  JButton doIt;

  double principal;  // первоначальная сумма вклада
  double rateOfRet;  // годовая норма прибыли
  double numYears;   // срок вклада в годах
  int numPerYear;    // сколько раз в году будет производиться выплата

  NumberFormat nf;

  public void init() {
    try {
      SwingUtilities.invokeAndWait(new Runnable () {
        public void run() {
          makeGUI();  // инициализация GUI
        }
      });
    } catch(Exception exc) {
      System.out.println("Can't create because of "+ exc);
      // System.out.println("Невозможно создать из-за "+ exc);
    }
  }

  // Устанавливаем и инициализируем GUI.
  private void makeGUI() {

    // Использование сеточной компоновки.
    GridBagLayout gbag = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(gbag);

    JLabel heading = new
          JLabel("Maximum Regular Withdrawals");
    // JLabel heading = new
    //       JLabel("Расчет максимального годового дохода");

    JLabel orgPLab = new JLabel("Original Principal ");
    // JLabel orgPLab = new JLabel("Первоначальная сумма вклада ");
    JLabel periodLab = new JLabel("Years ");
    // JLabel periodLab = new JLabel("Количество лет ");
    JLabel rateLab = new JLabel("Rate of Return ");
    // JLabel rateLab = new JLabel("Норма прибыли ");
    JLabel numWDLab =
          new JLabel("Number of Withdrawals per Year ");
    // JLabel numWDLab =
    //       new JLabel("Количество получений дохода в год ");
    JLabel maxWDLab = new JLabel("Maximum Withdrawal ");
    // JLabel maxWDLab = new JLabel("Максимальный годовой доход ");

    maxWDText = new JTextField(10);
    periodText = new JTextField(10);
    orgPText = new JTextField(10);
    rateText = new JTextField(10);
    numWDText = new JTextField(10);
```

```
// Поле максимальной суммы дохода, только отображение.
maxWDText.setEditable(false);

doIt = new JButton("Compute");
// doIt = new JButton("Вычислить");

// Определяем сетку.
gbc.weighty = 1.0; // используем строку, вес которой равен 1

gbc.gridwidth = GridBagConstraints.REMAINDER;
gbc.anchor = GridBagConstraints.NORTH;
gbag.setConstraints(heading, gbc);

// Размещаем большинство компонентов справа.
gbc.anchor = GridBagConstraints.EAST;

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(orgPLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(orgPText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(periodLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(periodText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(rateLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(rateText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(numWDLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(numWDText, gbc);

gbc.gridwidth = GridBagConstraints.RELATIVE;
gbag.setConstraints(maxWDLab, gbc);
gbc.gridwidth = GridBagConstraints.REMAINDER;
gbag.setConstraints(maxWDText, gbc);

gbc.anchor = GridBagConstraints.CENTER;
gbag.setConstraints(doIt, gbc);

// Добавляем все компоненты.
add(heading);
add(orgPLab);
add(orgPText);
add(periodLab);
add(periodText);
add(rateLab);
add(rateText);
add(numWDLab);
add(numWDText);
add(maxWDLab);
add(maxWDText);
add(doIt);

// Регистрируем на получение событий действий.
orgPText.addActionListener(this);
periodText.addActionListener(this);
rateText.addActionListener(this);
```

```
    numWDText.addActionListener(this);
    doIt.addActionListener(this);

    // Создаем числовой формат.
    nf = NumberFormat.getInstance();
    nf.setMinimumFractionDigits(2);
    nf.setMaximumFractionDigits(2);
  }

  /* Пользователь нажал <Enter> в текстовом поле или
     щелкнул на кнопке Compute. Отображаем результат,
     если все поля заполнены. */
  public void actionPerformed(ActionEvent ae) {
    double result = 0.0;

    String orgPStr = orgPText.getText();
    String periodStr = periodText.getText();
    String rateStr = rateText.getText();
    String numWDStr = numWDText.getText();

    try {
      if(orgPStr.length() != 0 &&
        periodStr.length() != 0 &&
        rateStr.length() != 0 &&
        numWDStr.length() != 0) {

        principal = Double.parseDouble(orgPStr);
        numYears = Double.parseDouble(periodStr);
        rateOfRet = Double.parseDouble(rateStr) / 100;
        numPerYear = Integer.parseInt(numWDStr);

        result = compute();

        maxWDText.setText(nf.format(result));
      }

      showStatus(""); // удаляем любое предыдущее сообщение об ошибке
    } catch (NumberFormatException exc) {
      showStatus("Invalid Data");
      // showStatus("Неверные данные");
      maxWDText.setText("");
    }
  }

  // Расчет максимальной регулярной суммы дохода.
  double compute() {
    double b, e;
    double t1, t2;

    t1 = rateOfRet / numPerYear;

    b = (1 + t1);
    e = numPerYear * numYears;

    t2 = Math.pow(b, e) - 1;

    return principal * (t1/t2 + t1);
  }
}
```

Нахождение остатка баланса по ссуде

Нередко бывает необходимо узнать остаток баланса по ссуде. Рассчитать его можно очень просто, если знать первоначальную сумму, процентную ставку, срок ссуды и количество произведенных платежей. Чтобы найти остаток баланса, необходимо сложить платежи, вычитая из каждого платежа сумму, выделенную под проценты, и полученный результат вычесть из исходной суммы.

Приведенный ниже код аплета `RemBal` находит остаток баланса по ссуде. Аплет, созданный этой программой, показан на рис. 32.6.

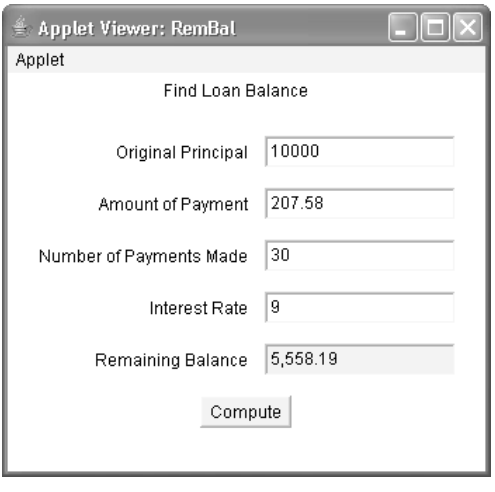

Рис. 32.6. Окно аплета `RemBal`

```
// Нахождение остатка баланса по ссуде.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.text.*;
/*
  <applet code="RemBal" width=340 height=260>
  </applet>
*/

public class RemBal extends JApplet
  implements ActionListener {

  JTextField orgPText, paymentText, remBalText,
             rateText, numPayText;
  JButton doIt;

  double orgPrincipal;   // исходная сумма
  double intRate;        // процент по ссуде
  double payment;        // сумма каждого платежа
  double numPayments;    // количество произведенных платежей

  /* Количество платежей в течение одного года. Можно сделать
     так, чтобы это значение вводил пользователь. */
  final int payPerYear = 12;
```

```
NumberFormat nf;

public void init() {
  try {
    SwingUtilities.invokeAndWait(new Runnable () {
      public void run() {
        makeGUI();  // инициализация GUI
      }
    });
  } catch(Exception exc) {
    System.out.println("Can't create because of "+ exc);
    // System.out.println("Невозможно создать из-за "+ exc);
  }
}

// Устанавливаем и инициализируем GUI.
private void makeGUI() {

  // Используем сеточную компоновку.
  GridBagLayout gbag = new GridBagLayout();
  GridBagConstraints gbc = new GridBagConstraints();
  setLayout(gbag);

  JLabel heading = new
        JLabel("Find Loan Balance ");
  // JLabel heading = new
  //       JLabel("Find Loan Balance ");

  JLabel orgPLab = new JLabel("Original Principal ");
  // JLabel orgPLab = new JLabel("Первоначальная сумма вклада ");
  JLabel paymentLab = new JLabel("Amount of Payment ");
  // JLabel paymentLab = new JLabel("Сумма платежа ");
  JLabel numPayLab = new JLabel("Number of Payments Made ");
  // JLabel numPayLab = new JLabel("Количество сделанных платежей ");
  JLabel rateLab = new JLabel("Interest Rate ");
  // JLabel rateLab = new JLabel("Процент по ссуде ");
  JLabel remBalLab = new JLabel("Remaining Balance ");
  // JLabel remBalLab = new JLabel("Остаточный баланс ");

  orgPText = new JTextField(10);
  paymentText = new JTextField(10);
  remBalText = new JTextField(10);
  rateText = new JTextField(10);
  numPayText = new JTextField(10);

  // Поле платежей, только для отображения.
  remBalText.setEditable(false);

  doIt = new JButton("Compute");
  // doIt = new JButton("Вычислить");

  // Определяем сетку.
  gbc.weighty = 1.0; // используем строку, вес которой равен 1

  gbc.gridwidth = GridBagConstraints.REMAINDER;
  gbc.anchor = GridBagConstraints.NORTH;
  gbag.setConstraints(heading, gbc);

  // Размещаем большинство компонентов справа.
  gbc.anchor = GridBagConstraints.EAST;

  gbc.gridwidth = GridBagConstraints.RELATIVE;
```

```
    gbag.setConstraints(orgPLab, gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(orgPText, gbc);

    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(paymentLab, gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(paymentText, gbc);

    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(rateLab, gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(rateText, gbc);

    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(numPayLab, gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(numPayText, gbc);

    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(remBalLab, gbc);
    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(remBalText, gbc);

    gbc.anchor = GridBagConstraints.CENTER;
    gbag.setConstraints(doIt, gbc);

    // Добавляем все компоненты.
    add(heading);
    add(orgPLab);
    add(orgPText);
    add(paymentLab);
    add(paymentText);
    add(numPayLab);
    add(numPayText);
    add(rateLab);
    add(rateText);
    add(remBalLab);
    add(remBalText);
    add(doIt);

    // Регистрируем на получение событий действия.
    orgPText.addActionListener(this);
    numPayText.addActionListener(this);
    rateText.addActionListener(this);
    paymentText.addActionListener(this);
    doIt.addActionListener(this);

    // Создаем числовой формат.
    nf = NumberFormat.getInstance();
    nf.setMinimumFractionDigits(2);
    nf.setMaximumFractionDigits(2);
  }

  /* Пользователь нажал <Enter> в текстовом поле
     или щелкнул на кнопке Compute. Отображаем результат,
     если заполнены все поля. */
public void actionPerformed(ActionEvent ae) {
  double result = 0.0;
```

```
    String orgPStr = orgPText.getText();
    String numPayStr = numPayText.getText();
    String rateStr = rateText.getText();
    String payStr = paymentText.getText();

    try {
      if(orgPStr.length() != 0 &&
        numPayStr.length() != 0 &&
        rateStr.length() != 0 &&
        payStr.length() != 0) {

        orgPrincipal = Double.parseDouble(orgPStr);
        numPayments = Double.parseDouble(numPayStr);
        intRate = Double.parseDouble(rateStr) / 100;
        payment = Double.parseDouble(payStr);

        result = compute();

        remBalText.setText(nf.format(result));
      }

        showStatus(""); // удаляем любое предыдущее сообщение об ошибке
      } catch (NumberFormatException exc) {
        showStatus("Invalid Data");
        // showStatus("Неверные данные");
        remBalText.setText("");
      }
    }

    // Расчет баланса по ссуде.
    double compute() {

      double bal = orgPrincipal;
      double rate = intRate / payPerYear;

      for(int i = 0; i < numPayments; i++)
        bal -= payment - (bal * rate);

      return bal;
    }
  }
```

Создание финансовых сервлетов

Несмотря на простоту создания и использования аплетов, они составляют всего лишь половину “уравнения” Java Internet. Другая половина остается за сервлетами. Во время соединения сервлеты выполняются на стороне сервера, и для некоторых приложений более подходящим вариантом являются именно сервлеты. Поскольку многим читателям может понадобиться использовать в своих коммерческих приложениях сервлеты вместо аплетов, в оставшейся части этой главы будет показан способ преобразования финансовых аплетов в сервлеты.

Поскольку все финансовые аплеты построены на одной и той же базовой структуре, то для преобразования мы выберем один аплет: `RegPay`. Предложенный базовый процесс вы сможете применять для преобразования любого аплета в сервлет. Вы сможете убедиться в том, что делать это совсем не трудно.

На заметку! *Сведения по созданию, проверке и запуску сервлетов можно получить в главе 31.*

Преобразование аплета RegPay в сервлет

Преобразовать аплет расчета ссуды RegPay в сервлет несложно. Для начала сервлет должен импортировать пакеты javax.servlet и javax.servlet.http. Он должен также расширять класс HttpServlet, а не JApplet. Кроме этого, потребуется удалить весь код GUI. После этого необходимо добавить HTML-код для получения параметров, передаваемых сервлету, который будет вызывать сервлет. Также сервлет должен посылать HTML-код, отображающий результаты. Базовые финансовые расчеты остаются теми же. Изменяется лишь способ получения данных и их отображение.

Сервлет RegPay

Следующий класс RegPayS является сервлет-версией аплета RegPay. Предполагается, что RegPayS.class будет храниться в каталоге примеров сервлетов Tomcat (см. главу 31). Не забудьте ввести его имя в файл web.xml (также см. главу 31).

```
// Простой сервлет для расчета ссуды.
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.text.*;

public class RegPayS extends HttpServlet {
  double principal;  // исходная сумма
  double intRate;    // процент по ссуде
  double numYears;   // срок, на который выдана ссуда

  /* Количество платежей в году. Можно сделать так,
     чтобы это значение вводил пользователь. */
  final int payPerYear = 12;

  NumberFormat nf;

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
    throws ServletException, IOException {
    String payStr = "";

    // Создаем числовой формат.
    nf = NumberFormat.getInstance();
    nf.setMinimumFractionDigits(2);
    nf.setMaximumFractionDigits(2);

    // Получаем параметры.
    String amountStr = request.getParameter("amount");
    String periodStr = request.getParameter("period");
    String rateStr = request.getParameter("rate");

    try {
      if(amountStr != null && periodStr != null &&
         rateStr != null) {
        principal = Double.parseDouble(amountStr);
        numYears = Double.parseDouble(periodStr);
        intRate = Double.parseDouble(rateStr) / 100;

        payStr = nf.format(compute());
      }
      else { // пропущен один или более параметров
        amountStr = "";
```

```
      periodStr = "";
      rateStr = "";
    }
  } catch (NumberFormatException exc) {
    // Предпринимаем соответствующее действие.
  }

  // Задаем тип содержимого.
  response.setContentType("text/html");

  // Получаем поток выходных данных.
  PrintWriter pw = response.getWriter();

  // Отображаем необходимый HTML-код.
  // Введите начальный баланс
  pw.print("<html><body> <left>" +
         "<form name=\"Form1\"" +
         " action=\"http://localhost:8080/" +
         "servlets-examples/servlet/RegPayS\">" +
         "<B>Enter amount to finance:</B>" +
         " <input type=textbox name=\"amount\"" +
         " size=12 value=\"");
  pw.print(amountStr + "\">");
  // Введите количество лет
  pw.print("<BR><B>Enter term in years:</B>" +
         " <input type=textbox name=\"period\""+
         " size=12 value=\"");
  pw.println(periodStr + "\">");
  // Введите процент по ссуде
  pw.print("<BR><B>Enter interest rate:</B>" +
         " <input type=textbox name=\"rate\"" +
         " size=12 value=\"");
  pw.print(rateStr + "\">");
  // Ежемесячный платеж
  pw.print("<BR><B>Monthly Payment:</B>" +
         " <input READONLY type=textbox" +
         " name=\"payment\" size=12 value=\"");
  pw.print(payStr + "\">");
  // Отправить
  pw.print("<BR><P><input type=submit value=\"Submit\">");
  pw.println("</form> </body> </html>");
 }

 // Расчет платежа по ссуде.
 double compute() {
  double numer;
  double denom;
  double b, e;

  numer = intRate * principal / payPerYear;
  e = -(payPerYear * numYears);
  b = (intRate / payPerYear) + 1.0;

  denom = 1.0 - Math.pow(b, e);

  return numer / denom;
 }
}
```

Первое, на что следует обратить внимание в сервлете `RegPayS` — это то, что он имеет только два метода: `doGet()` и `compute()`. Метод `compute()` такой же, как и в аплете. Метод `doGet()` определяется в классе `HttpServlet`, который расширяет сервлет `RegPayS`. Этот метод вызывается сервером, когда сервлет должен ответить на запрос GET. Заметьте, что он передает ссылку объектам `HttpServletRequest` и `HttpServletResponse`, связанным с запросом.

Из параметра `request` сервлет получает аргументы, связанные с запросом. Для этого он вызывает метод `getParameter()`. Параметр возвращается в строковом виде. Поэтому числовое значение необходимо вручную преобразовывать в двоичный формат. Если ни один из параметров не доступен, возвращается `null`.

Из объекта `response` сервлет получает поток, в который можно записать информацию отклика. Затем отклик возвращается браузеру посредством вывода в данный поток. Прежде чем получить `PrintWriter` в потоке отклика, в качестве типа вывода необходимо определить text/html, вызвав для этого метод `setContentType()`.

Сервлет `RegPayS` можно вызывать как с параметрами, так и без них. При вызове без параметров сервлет отвечает HTML-кодом, отображающим пустую форму калькулятора ссуд. В противном случае при вызове со всеми необходимыми параметрами `RegPayS` рассчитывает платеж по ссуде и повторно отображает форму, на этот раз с заполненным полем платежа. На рис. 32.7 показан сервлет в действии.

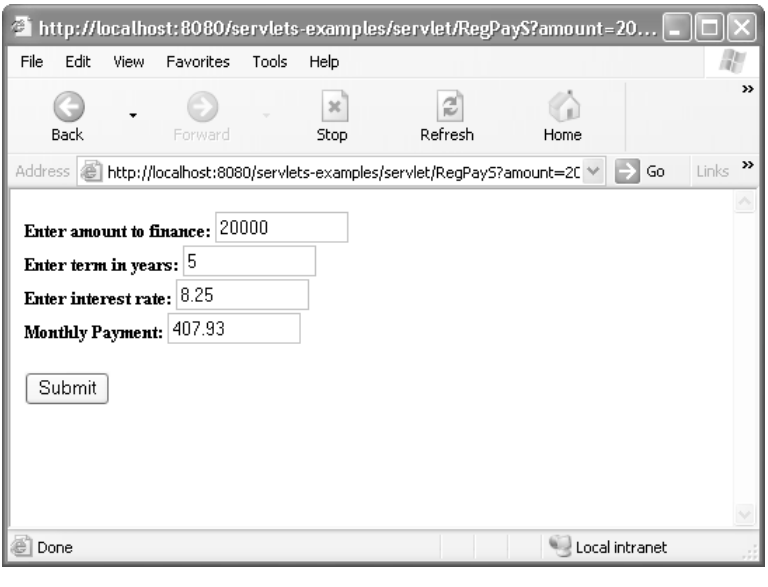

Рис. 32.7. Сервлет `RegPayS` в действии

Вызвать сервлет `RegPayS` проще всего можно, если связаться с его URL-адресом, не передавая никаких параметров. Например, если предположить, что вы используете Tomcat, можно использовать следующую строку:

```
<A HREF = "http://localhost:8080/servlets-examples/servlet/RegPayS">
Калькулятор ссуд</A>
```

В результате будет отображена ссылка “Калькулятор ссуд” на сервлет RegPayS, расположенный в каталоге примеров сервлетов Tomcat. Обратите внимание, что при этом никакие параметры не передаются. Сервлет RegPayS будет возвращать полный HTML-код, отображающий пустую страницу калькулятора ссуд.

Сервлет RegPayS можно вызвать и по-другому, если сначала отобразить вручную пустую форму. Этот прием показан ниже, и тоже с использованием каталога примеров сервлетов Tomcat.

```
<html>
<body>
<form name="Form1"
  action="http://localhost:8080/servlets-examples/servlet/RegPayS">
<B>Enter amount to finance:</B>
<input type=textbox name="amount" size=12 value="">
<BR>
<B>Enter term in years:</B>
<input type=textbox name="period" size=12 value="">
<BR>
<B>Enter interest rate:</B>
<input type=textbox name="rate" size=12 value="">
<BR>
<B>Monthly Payment:</B>
<input READONLY type=textbox name="payment"
  size=12 value="">
<BR><P>
<input type=submit value="Submit">
</form>
</body>
</html>
```

Самостоятельная работа

Первое, что вы можете попробовать сделать — преобразовать другие финансовые аплеты в сервлеты. Поскольку все финансовые аплеты имеют одну и ту же структуру, вам достаточно будет повторить те же действия, что и для сервлета RegPayS. Существует множество других финансовых расчетов, которые вы найдете полезными для реализации в качестве аплетов или сервлетов (например, расчет нормы прибыли по вкладу или расчет суммы обычного вклада, которая необходима для того, чтобы в будущем достичь определенной стоимости). Можно также вывести график погашения ссуды. Попытайтесь создать крупное приложение для производства расчетов, описанных в этой главе, позволяя пользователю выбирать необходимый расчет из меню.

33
ГЛАВА

Создание утилиты загрузки на Java

Сталкивались ли вы когда-нибудь с внезапным прекращением загрузки данных из Internet, в результате чего загрузку приходилось начинать с начала? Если вы подключаетесь к Internet посредством коммутируемого соединения, то, скорее всего, вам приходилось переживать подобные досадные моменты. Прекращение процесса загрузки может быть вызвано чем угодно, начиная с отключений во время ожидания звонка, и заканчивая сбоем самой операционной системы. Если начинать процесс загрузки снова и снова, для этого понадобиться очень много времени и нервов, что в итоге может привести вас в полное уныние.

Пользователи нередко забывают о том факте, что во многих случаях при внезапном прекращении процесса загрузки его можно возобновить с того места, в каком он был прекращен, не начиная все с самого начала. В этой главе мы займемся созданием инструментального средства под названием Download Manager (Диспетчер загрузки), которое будет управлять процессами загрузки данных из Internet и позволит возобновлять прерванную загрузку данных. Это средство позволит приостанавливать и продолжать загрузку данных, а также управлять несколькими процессами загрузки одновременно.

Самым главным достоинством утилиты Download Manager является то, что с ее помощью можно загружать только определенные порции файла. В классическом сценарии загружается весь файл целиком. Если передача файла по какой-либо причине будет внезапно прекращена, то попытка завершить загрузку файла не увенчается успехом. А утилита Download Manager может возобновить загрузку файла с того места, в котором произошел сбой, и загрузить оставшийся фрагмент файла. Следует учесть, однако, что не все процессы загрузки создаются одинаковыми, и некоторые из них вообще невозможно возобновить после внезапного прекращения. В следующем разделе вы узнаете о том, для каких файлов можно возобновлять процесс загрузки, а для каких — нет.

Download Manager — это не только полезная утилита, но и превосходный пример мощности и лаконичности встроенных Java API — особенно в случае применения для работы с Internet. Поскольку создатели Java заботились, прежде всего, о возможностях работы с Internet, то неудивительно, что сетевые возможности Java являются непревзойденными. Например, если вы попытаетесь создать утилиту загрузки на другом языке программирования (например, C++), то для этого вам придется приложить гораздо больше усилий, и вы обязательно столкнетесь с рядом трудностей.

Загрузка данных из Internet

Чтобы разобраться с тем, как работает Download Manager, необходимо выяснить, как на самом деле производится загрузка данных из Internet.

Процессы загрузки данных из Internet в самом простом виде представляют собой обыкновенные транзакции между клиентом и сервером. Клиент, в качестве которого выступает ваш браузер, просит загрузить файл, хранящийся на сервере в Internet. Сервер отвечает ему, отправляя запрошенный файл браузеру. Чтобы клиенты могли взаимодействовать с серверами, они должны иметь установленный протокол. Самыми распространенными протоколами для загрузки файлов являются FTP (File Transfer Protocol — протокол передачи файлов) и HTTP (Hypertext Transfer Protocol — протокол передачи гипертекста). Протокол FTP обычно используется для обмена файлов между компьютерами, а HTTP — для передачи Web-страниц и связанных с ними файлов (графических изображений, звуковых файлов и тому подобного). Со временем, когда система World Wide Web приобрела большую популярность, протокол HTTP стал доминирующим протоколом для загрузки файлов из Internet. Это, однако, не означает, что роль протокола FTP перестала быть важной.

Утилита Download Manager, созданием которой мы займемся в этой главе, будет поддерживать загрузку только по протоколу HTTP — это ограничение продиктовано лишь краткостью изложения материала по данной теме. Однако если вы добавите поддержку протокола FTP, это послужит вам хорошим уроком по расширению кода. Процесс загрузки с помощью HTTP может происходить в двух вариантах: с возможностью возобновления (HTTP 1.1) и без возможности возобновления (HTTP 1.0). Они отличаются между собой способом запроса файлов, хранящихся на серверах. Если используется устаревший вариант HTTP 1.0, то клиент может выслать серверу запрос, чтобы он отправил ему файл, тогда как с помощью HTTP 1.1 клиент может попросить сервер выслать как весь файл целиком, так и *только определенную порцию файла*. Именно на этой особенности и будет построена наша утилита Download Manager.

Обзор утилиты Download Manager

Утилита Download Manager использует простой но, тем не менее, эффективный GUI-интерфейс, построенный с помощью библиотек Java Swing. Окно Download Manager показано на рис. 33.1. Использование Swing позволяет придать интерфейсу живой и современный внешний вид.

Графический интерфейс поддерживает список процессов загрузки, управляемых в данный момент времени. Для каждого процесса загрузки в списке указывается его URL-адрес, размер файла в байтах, количество загруженных данных (в процентах) и текущее состояние. Процесс загрузки может иметь одно из следующих состояний: Downloading (обозначает непосредственно сам процесс загрузки), Paused (процесс загрузки приостановлен), Complete (процесс загрузки завершен), Error (ошибка процесса загрузки) и Cancelled (процесс загрузки отменен). Графический интерфейс позволяет также добавлять процессы загрузки в список и изменять состояние каждого процесса, присутствующего в списке. Если в списке выбрать какой-нибудь процесс загрузки, то в зависимости от его текущего состояния его можно приостановить, возобновить, отменить или вообще удалить из списка.

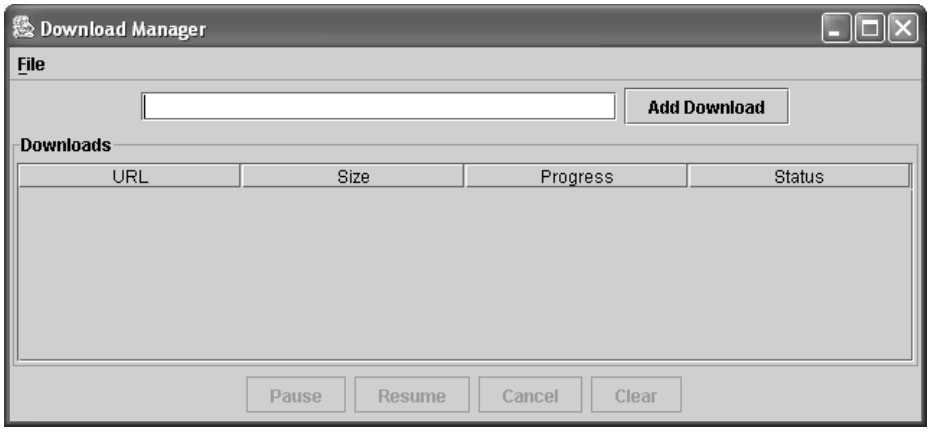

Рис. 33.1. Графический интерфейс утилиты Download Manager

Утилита Download Manager содержит несколько классов, позволяющих естественным образом разделить функциональные компоненты. Это классы `Download`, `DownloadsTableModel`, `ProgressRenderer` и `DownloadManager`.

Класс `DownloadManager` отвечает за графический пользовательский интерфейс и использует классы `DownloadsTableModel` и `ProgressRenderer` для отображения текущего списка процессов загрузки. Класс `Download` представляет "управляемый" процесс загрузки и отвечает за непосредственную загрузку файла. В следующих разделах мы рассмотрим каждый из этих классов подробно, уделяя особое внимание их внутренней работе и объясняя, как они связаны друг с другом.

Класс `Download`

Класс `Download` является главным классом утилиты Download Manager (или ее "рабочей лошадкой"). Его основная задача заключается в загрузке файла и сохранении его содержимого на диске. При каждом добавлении нового процесса загрузки в утилиту Download Manager создается новый объект `Download`, который и будет заниматься обслуживанием данного процесса.

Утилита Download Manager может загружать несколько файлов одновременно. Для этого необходимо, чтобы каждый процесс выполнялся независимо от остальных процессов. Кроме того, нужно иметь возможность управлять состоянием каждого отдельного процесса, что позволит отображать его в интерфейсе. Это можно сделать с помощью класса `Download`.

Ниже представлен полный код класса `Download`. Обратите внимание, что он расширяет класс `Observable` и реализует интерфейс `Runnable`. Каждую часть мы проанализируем детально в следующих разделах.

```
import java.io.*;
import java.net.*;
import java.util.*;

// Этот класс загружает файл, на который указывает URL-адрес.
class Download extends Observable implements Runnable {
```

```
// Максимальный размер буфера загрузки.
private static final int MAX_BUFFER_SIZE = 1024;

// Названия состояний.
public static final String STATUSES[] = {"Downloading",
  "Paused", "Complete", "Cancelled", "Error"};

// Это коды состояний.
public static final int DOWNLOADING = 0;
public static final int PAUSED = 1;
public static final int COMPLETE = 2;
public static final int CANCELLED = 3;
public static final int ERROR = 4;
private URL url; // загрузка URL-адреса
private int size; // размер загружаемых данных в байтах
private int downloaded; // количество загруженных байтов
private int status; // текущее состояние процесса загрузки

// Конструктор для класса Download.
public Download(URL url) {
  this.url = url;
  size = -1;
  downloaded = 0;
  status = DOWNLOADING;

  // Начало процесса загрузки.
  download();
}

// Получаем URL-адрес для данного процесса загрузки.
public String getUrl() {
  return url.toString();
}

// Определяем размер загружаемых данных.
public int getSize() {
  return size;
}

// Получаем информацию о ходе данного процесса загрузки.
public float getProgress() {
  return ((float) downloaded / size) * 100;
}

// Получаем сведения о состоянии данного процесса загрузки.
public int getStatus() {
  return status;
}

// Приостанавливаем данный процесс загрузки.
public void pause() {
  status = PAUSED;
  stateChanged();
}

// Возобновляем данный процесс загрузки.
public void resume() {
  status = DOWNLOADING;
  stateChanged();
  download();
}
```

```
// Отменяем данный процесс загрузки.
public void cancel() {
  status = CANCELLED;
  stateChanged();
}

// В процессе загрузки возникла ошибка.
private void error() {
  status = ERROR;
  stateChanged();
}

// Начинаем или возобновляем процесс загрузки.
private void download() {
  Thread thread = new Thread(this);
  thread.start();
}

// Извлекаем имя файла из URL-адреса.
private String getFileName(URL url) {
  String fileName = url.getFile();
  return fileName.substring(fileName.lastIndexOf('/') + 1);
}

// Загружаем файл.
public void run() {
  RandomAccessFile file = null;
  InputStream stream = null;

  try {
    // Открываем соединение с данным URL-адресом.
    HttpURLConnection connection =
      (HttpURLConnection) url.openConnection();

    // Определяем, какую часть файла нужно загрузить.
    connection.setRequestProperty("Range",
      "bytes=" + downloaded + "-");

    // Соединяемся с сервером.
    connection.connect();

    // Убеждаемся в том, что код отклика находится в диапазоне 200.
    if (connection.getResponseCode() / 100 != 2) {
      error();
    }

    // Проверяем, имеет ли содержимое допустимую длину.
    int contentLength = connection.getContentLength();
    if (contentLength < 1) {
      error();
    }

    /* Задаем размер для данного процесса загрузки,
       если он еще не был задан. */
    if (size == -1) {
      size = contentLength;
      stateChanged();
    }

    // Открываем файл и ищем конец файла.
    file = new RandomAccessFile(getFileName(url), "rw");
    file.seek(downloaded);
```

```
        stream = connection.getInputStream();
        while (status == DOWNLOADING) {
          /* Задаем размер буфера так, чтобы загрузить
             оставшуюся часть файла. */
          byte buffer[];
          if (size - downloaded > MAX_BUFFER_SIZE) {
            buffer = new byte[MAX_BUFFER_SIZE];
          } else {
            buffer = new byte[size - downloaded];
          }

          // Производим чтение из сервера в буфер.
          int read = stream.read(buffer);
          if (read == -1)
            break;

          // Записываем содержимое буфера в файл.
          file.write(buffer, 0, read);
          downloaded += read;
          stateChanged();
        }

        /* Определяем состояние как завершенное, если была
           достигнута эта точка, поскольку в ней загрузка завершена. */
        if (status == DOWNLOADING) {
          status = COMPLETE;
          stateChanged();
        }
      } catch (Exception e) {
        error();
      } finally {
        // Закрываем файл.
        if (file != null) {
          try {
            file.close();
          } catch (Exception e) {}
        }

        // Закрываем соединение с сервером.
        if (stream != null) {
          try {
            stream.close();
          } catch (Exception e) {}
        }
      }
    }

    // Уведомляем наблюдателей об изменении состояния данного
       процесса загрузки.
    private void stateChanged() {
      setChanged();
      notifyObservers();
    }
  }
```

Переменные класса Download

Класс Download начинается с объявления нескольких переменных static final, определяющих различные константы, используемые в классе. Затем объявляются четыре переменных экземпляра. Переменная url хранит URL-адрес файла, который необходимо загрузить, переменная size — размер этого файла в байтах, переменная download — количество байт, загруженных на данный момент, а переменная status показывает текущее состояние процесса загрузки.

Конструктор класса Download

Конструктор класса Download передает ссылку на URL-адрес для загрузки в виде объекта URL, который присваивается переменной экземпляра url. Затем он присваивает остальным переменным экземпляра их исходные значения и вызывает метод download(). Обратите внимание на то, что переменная size имеет значение -1, которое показывает, что размер еще не определен.

Метод download()

Метод download() создает новый объект Thread, передавая ему ссылку для вызова экземпляра Download. Как уже было сказано ранее, необходимо, чтобы каждый процесс загрузки выполнялся отдельно от остальных. Чтобы класс Download действовал самостоятельно, он должен выполняться в своем собственном потоке. Java имеет превосходную встроенную поддержку потоков и позволяет использовать их мгновенно. Чтобы использовать потоки, класс Download реализует интерфейс Runnable, переопределяя метод run(). После того как метод download() создаст новый экземпляр Thread, передавая его конструктору Runnable Download, он вызывает метод start() потока. Вызов метода start() приводит к выполнению метода run() экземпляра Runnable (класса Download).

Метод run()

При выполнении метода run() начинается процесс загрузки данных. Вследствие того, что этот метод имеет большой размер и такое же большое значение, мы детально проанализируем все его строки. Метод run() начинается следующим образом:

```
RandomAccessFile file = null;
InputStream stream = null;

try {
  // Открываем соединение с данным URL-адресом.
  HttpURLConnection connection =
    (HttpURLConnection) url.openConnection();
```

Вначале метод run() определяет переменные для сетевого потока, из которого будет считываться содержимое, и определяет файл для записи этого содержимого. Затем открывается соединение с URL-адресом посредством метода url.openConnection(). Поскольку известно, что утилита Download Manager поддерживает загрузку только по протоколу HTTP, то соединение приводится к типу HttpURLConnection. Приведение соединения к типу HttpURLConnection позволяет воспользоваться преимуществами функциональных возможностей HTTP-соединения (например, методом getResponseCode()). Обратите внимание, что при вызове метода url.openConnection() на самом деле не

устанавливается соединение с сервером, на который указывает URL-адрес. Он просто создает новый экземпляр `URLConnection`, связанный с URL-адресом, который позже будет использоваться для соединения с сервером.

После того как будет создано соединение `HttpURLConnection`, посредством вызова метода `connection.setRequestProperty()` определяется свойство запроса на соединение, как показано ниже:

```
// Определяем, какую часть файла нужно загрузить.
connection.setRequestProperty("Range",
  "bytes=" + downloaded + "-");
```

Если определить свойства запроса, то серверу, с которого будет производиться загрузка, можно будет посылать некоторую дополнительную информацию о запросе. В данном случае определяется свойство `"Range"`. Это очень важное свойство, поскольку оно определяет диапазон в байтах, которые будут запрошены для загрузки с сервера. Обычно загружаются сразу все байты файла. Однако если процесс загрузки будет прерван или приостановлен, получить нужно будет только оставшиеся байты. Определение свойства `"Range"` является основой для работы утилиты Download Manager.

Свойство `"Range"` определяется следующим образом:

```
начальный_байт - конечный_байт
```

Например, “0 - 12345”. Однако конечный байт диапазона указывать необязательно. Если конечный байт не будет указан, диапазон завершится в конце файла. Метод `run()` никогда не определяет конечный байт, поскольку процессы загрузки должны выполняться до тех пор, пока не будет загружен весь диапазон, если только процесс не будет прерван или приостановлен.

Ниже показано несколько строк кода:

```
// Соединяемся с сервером.
connection.connect();

// Убеждаемся в том, что код отклика находится в диапазоне 200.
if (connection.getResponseCode() / 100 != 2) {
  error();
}

// Проверяем, имеет ли содержимое допустимую длину.
int contentLength = connection.getContentLength();
if (contentLength < 1) {
  error();
}
```

Метод `connection.connect()` вызывается для того, чтобы установить соединение с сервером, с которого будет производиться загрузка данных. Затем осуществляется проверка кода отклика, возвращаемого сервером. Протокол HTTP имеет список кодов откликов, которые показывают отклик сервера на запрос. Коды HTTP-отклика организованы в числовые диапазоны, кратные 100, а диапазон 200 указывает на успешный отклик. Чтобы проверить, находится ли код отклика в диапазоне 200, вызывается метод `connection.getResponseCode()` и возвращенный им результат делится на 100. Если результат этого деления будет равен 2, соединение считается успешным.

После этого метод `run()` получает длину содержимого вызовом метода `connection.getContentLength()`. Длина содержимого представляет количество байт в требуемом файле. Если длина содержимого меньше 1, вызывается метод `error()`. Метод `error()`

изменяет состояние процесса загрузки на ERROR, а затем вызывает метод stateChanged(). Мы еще вернемся к методу stateChanged().

После того как длина содержимого будет получена, следующий код проверяет, присвоено ли это значение переменной size:

```
/* Задаем размер для данного процесса загрузки,
   если он еще не был задан. */
if (size == -1) {
  size = contentLength;
  stateChanged();
}
```

Как видите, вместо того чтобы безусловно присваивать длину содержимого переменной size, она присваивается только в том случае, если переменная еще не имеет значения. Дело в том, что длина содержимого отражает количество байтов, которые будет посылать сервер. Если не будет указано ничего кроме нулевого начального диапазона, длина содержимого будет представлять только часть размера файла. Переменной size необходимо присвоить полный размер загружаемого файла.

В следующих строках кода создается новый файл RandomAccessFile, для чего используется часть имени файла из URL-адреса для загрузки, которая была получена с помощью метода getFileName():

```
// Открываем файл и ищем конец файла.
file = new RandomAccessFile(getFileName(url), "rw");
file.seek(downloaded);
```

RandomAccessFile открывается в режиме "rw", который определяет, что файл можно считывать и записывать. После того как файл будет открыт, метод run() ищет конец файла с помощью метода file.seek(), присваивая значение переменной downloaded. В результате файл будет позиционирован на количество загруженных байтов, т.е. на конец файла. Позиционирование файла в конец необходимо на случай возобновления загрузки. При возобновлении загрузки в файл будут добавляться только новые байты, которые не будут перезаписывать ранее загруженные байты. После того как выходной файл будет подготовлен, метод connection.getInputStream() получает метку сетевого потока для открытого соединения с сервером, как показано ниже:

```
stream = connection.getInputStream();
```

Само действие начинается в цикле while:

```
while (status == DOWNLOADING) {
  /* Задаем размер буфера так, чтобы загрузить
     оставшуюся часть файла. */
  byte buffer[];
  if (size - downloaded > MAX_BUFFER_SIZE) {
    buffer = new byte[MAX_BUFFER_SIZE];
  } else {
    buffer = new byte[size - downloaded];
  }

  // Производим чтение из сервера в буфер.
  int read = stream.read(buffer);
  if (read == -1)
    break;
```

```
    // Записываем содержимое буфера в файл.
    file.write(buffer, 0, read);
    downloaded += read;
    stateChanged();
  }
```

Цикл выполняется до тех пор, пока значением переменной `status` не будет `DOWNLOADING`. Внутри цикла создается буферный массив `byte` для хранения загружаемых байтов. Размер буфера определяется в соответствии с тем, какое количество данных осталось загрузить. Если загрузить осталось больше чем `MAX_BUFFER_SIZE`, для определения размера буфера используется `MAX_BUFFER_SIZE`. В противном случае размер буфера определяется в точности по количеству байт, которые осталось загрузить. После того как размер буфера будет соответствующим образом определен, вызовом метода `stream.read()` начинается процесс загрузки. Этот метод считывает байты с сервера и переносит их в буфер, возвращая подсчет количества прочитанных байтов. Если количество прочитанных байтов равно `-1`, это означает, что загрузка завершена, и работа цикла заканчивается. В противном случае загрузка будет продолжаться, и считываемые байты будут записываться на диск с помощью метода `file.write()`. Затем переменная `downloaded` обновляется для отображения количества байтов, загруженных на данный момент. Наконец, внутри цикла вызывается метод `stateChanged()`. Этот метод мы рассмотрим чуть позже.

После завершения работы цикла следующий код проверяет, по какой причине была завершена работа цикла:

```
/* Определяем состояние как завершенное, если была
   достигнута эта точка, поскольку в ней загрузка завершена. */
if (status == DOWNLOADING) {
  status = COMPLETE;
  stateChanged();
}
```

Если состоянием процесса загрузки остается `DOWNLOADING`, это означает, что работа цикла была завершена вследствие завершения процесса загрузки. В противном случае работа цикла была завершена по причине изменения состояния процесса загрузки.

Метод `run()` завершается блоками `catch` и `finally`, показанными ниже:

```
} catch (Exception e) {
  error();
} finally {

  // Закрываем файл.
  if (file != null) {
    try {
      file.close();
    } catch (Exception e) {}
  }

  // Закрываем соединение с сервером.
  if (stream != null) {
    try {
      stream.close();
    } catch (Exception e) {}
  }
}
```

Если в процессе загрузки возникнет исключение, блок `catch` перехватит его и вызовет метод `error()`. Блок `finally` гарантирует, что если соединения `file` и `stream` были открыты, они будут закрыты вне зависимости от того, возникало исключение или нет.

Метод `stateChanged()`

Чтобы утилита Download Manager могла отображать самую последнюю информацию о каждом управляемом ним процессе загрузки, она должна знать обо всех изменениях информации о процессе загрузки. Для этой цели используется программный шаблонный проект Observer. Шаблон Observer аналогичен почтовой рассылке, при которой каждый пользователь регистрируется на получение уведомлений. При появлении нового уведомления каждый человек, указанный в списке, получает сообщение с уведомлением. В случае шаблона Observer используется специальный класс, посредством которого классы наблюдения могут зарегистрироваться на получение уведомлений об изменениях.

Класс `Download` использует шаблон Observer, расширяя встроенный в Java служебный класс `Observable`. Расширение класса `Observable` позволяет классам, реализующим интерфейс Java `Observer`, регистрироваться с помощью класса `Download` на получение уведомлений об изменениях. Каждый раз, когда классу `Download` необходимо уведомить своих зарегистрированных наблюдателей (`Observer`) об изменении, вызывается метод `stateChanged()`. Метод `stateChanged()` сначала вызывает метод `setChanged()` класса `Observable`, чтобы отметить класс как такой, что был изменен. Затем метод `stateChanged()` вызывает метод `notifyObservers()` класса `Observable`, который распространяет уведомление об изменении зарегистрированным наблюдателям.

Методы действия и средства доступа

Класс `Download` имеет многочисленные методы действия и средства доступа для управления процессом загрузки и получения из него данных. Методы `pause()`, `resume()` и `cancel()`, соответственно, приостанавливают, возобновляют или отменяют процесс загрузки. Точно так же метод `error()` отмечает процесс загрузки как содержащий ошибку. Методы средства доступа `getURL()`, `getSize()`, `getProgress()` и `getStatus()` возвращают их текущие соответствующие значения.

Класс `ProgressRenderer`

Класс `ProgressRenderer` представляет собой небольшой служебный класс, который используется для визуализации текущего процесса загрузки, указанного в экземпляре `JTable` “Downloads” графического интерфейса. Обычно экземпляр `JTable` визуализирует данные в каждой ячейке в виде текста. Однако нередко приходится визуализировать данные в ячейке в другом виде, отличном от текстового. В случае с утилитой Download Manager нам нужно визуализировать каждую ячейку столбца Progress таблицы в виде индикатора хода работ. Это можно сделать с помощью класса `ProgressRenderer`, представленного ниже. Обратите внимание, что он расширяет класс `JProgressBar` и реализует интерфейс `TableCellRenderer`.

```
import java.awt.*;
import javax.swing.*;
import javax.swing.table.*;
// Этот класс визуализирует JProgressBar в ячейке таблицы.
class ProgressRenderer extends JProgressBar implements TableCellRenderer
{
```

```
  // Конструктор для ProgressRenderer.
  public ProgressRenderer(int min, int max) {
    super(min, max);
  }

  /* Возвращает JProgressBar в качестве визуализатора
     для данной ячейки таблицы. */
  public Component getTableCellRendererComponent(
    JTable table, Object value, boolean isSelected,
    boolean hasFocus, int row, int column)
  {
    // Определяет процент выполнения JProgressBar.
    setValue((int) ((Float) value).floatValue());
    return this;
  }
}
```

Класс `ProgressRenderer` использует преимущество того факта, что класс `JTable` из коллекции Swing имеет систему визуализации, которая может принимать "подключаемые модули" для визуализации ячеек таблицы. Чтобы воспользоваться этой системой визуализации, нужно чтобы, во-первых, класс `ProgressRenderer` реализовал интерфейс `TableCellRenderer` из коллекции Swing. Во-вторых, экземпляр `ProgressRenderer` необходимо зарегистрировать с экземпляром `JTable`; так можно сообщить экземпляру `JTable` о том, какие ячейки необходимо визуализировать с помощью "подключаемого модуля".

Для реализации интерфейса `TableCellRenderer` необходимо, чтобы класс переопределял метод `getTableCellRendererComponent()`.

Метод `getTableCellRendererComponent()` вызывается каждый раз, когда экземпляр `JTable` приступает к визуализации ячейки, для которой зарегистрирован этот класс. Этот метод передает несколько переменных, хотя в данном случае используется только переменная `value`. Переменная `value` хранит данные для визуализируемой ячейки и передается методу `setValue()` класса `JProgressBar`. В результате выполнения метода `getTableCellRendererComponent()` возвращается ссылка на его класс. Это возможно благодаря тому, что класс `ProgressRenderer` является подклассом `JProgressBar`, который представляет собой наследника AWT-класса `Component`.

Класс `DownloadsTableModel`

Класс `DownloadsTableModel` содержит список процессов загрузки утилиты Download Manager и является резервным источником данных для экземпляра `JTable` "Downloads" графического интерфейса.

Ниже представлен класс `DownloadsTableModel`. Обратите внимание, что он расширяет класс `AbstractTableModel` и реализует интерфейс `Observer`.

```
import java.util.*;
import javax.swing.*;
import javax.swing.table.*;

// Этот класс управляет данными таблицы загрузки.
class DownloadsTableModel extends AbstractTableModel
  implements Observer
{
```

```
// Имена столбцов таблицы.
  private static final String[] columnNames = {"URL", "Size",
    "Progress", "Status"};

// Классы для значений каждого столбца.
private static final Class[] columnClasses = {String.class,
  String.class, JProgressBar.class, String.class};

// Табличный список процессов загрузки.
private ArrayList<Download> downloadList = new ArrayList<Download>();

// Добавление нового процесса загрузки в таблицу.
public void addDownload(Download download) {
  // Регистрация на получение уведомления об изменениях
  // в процессе загрузки.
  download.addObserver(this);

  downloadList.add(download);

  // Генерация для таблицы уведомления о вставке строки.
  fireTableRowsInserted(getRowCount() - 1, getRowCount() - 1);
}

// Получение процесса загрузки для определенной строки.
public Download getDownload(int row) {
  return downloadList.get(row);
}

// Удаление процесса загрузки из списка.
public void clearDownload(int row) {
  downloadList.remove(row);

  // Генерация для таблицы уведомления об удалении строки.
  fireTableRowsDeleted(row, row);
}

// Получение подсчитанного количества столбцов таблицы.
public int getColumnCount() {
  return columnNames.length;
}

// Получение имени столбца.
public String getColumnName(int col) {
  return columnNames[col];
}

// Получение класса столбца.
public Class getColumnClass(int col) {
  return columnClasses[col];
}

// Получение подсчитанного количества строк таблицы.
public int getRowCount() {
  return downloadList.size();
}

// Получение значения для определенной комбинации строки и столбца.
public Object getValueAt(int row, int col) {
  Download download = downloadList.get(row);
  switch (col) {
    case 0: // URL-адрес
      return download.getUrl();
```

```
        case 1: // Размер
          int size = download.getSize();
          return (size == -1) ? "" : Integer.toString(size);
        case 2: // Ход выполнения
          return new Float(download.getProgress());
        case 3: // Состояние
          return Download.STATUSES[download.getStatus()];
      }
      return "";
    }

    /* Вызываем обновление, если процесс загрузки сообщает
       своим наблюдателям о каких-либо изменениях */
    public void update(Observable o, Object arg) {
      int index = downloadList.indexOf(o);

      // Генерация для таблицы уведомления об обновлении строки.
      fireTableRowsUpdated(index, index);
    }
  }
```

Класс `DownloadsTableModel`, по сути, является служебным классом, используемым экземпляром `JTable` "Downloads" для управления данными в таблице. После того как экземпляр `JTable` будет инициализирован, он передается экземпляру класса `DownloadsTableModel`. Затем `JTable` вызывает несколько методов в экземпляре `DownloadsTableModel` для заполнения таблицы. Метод `getColumnCount()` вызывается для получения количества столбцов в таблице. Подобно ему, метод `getRowCount()` используется для получения количества строк в таблице. Метод `getColumnName()` возвращает имя столбца на основании его идентификатора. Метод `getDownload()` принимает идентификатор строки и возвращает связанный объект `Download` из списка. Об остальных методах класса `DownloadsTableModel`, которые являются более сложными, мы поговорим подробно в следующих далее разделах.

Метод `addDownload()`

Метод `addDownload()`, показанный ниже, добавляет новый объект `Download` в список управляемых процессов загрузки и, следовательно, строку в таблицу.

```
// Добавление нового процесса загрузки в таблицу.
public void addDownload(Download download) {
  // Регистрация на получение уведомления об изменениях
  // в процессе загрузки.
  download.addObserver(this);

  downloadList.add(download);

  // Генерация для таблицы уведомления о вставке строки.
  fireTableRowsInserted(getRowCount() - 1, getRowCount() - 1);
}
```

Этот метод сначала регистрируется с новым классом `Download` в качестве наблюдателя `Observer` на получение уведомлений об изменениях. Затем объект `Download` добавляется во внутренний список управляемых процессов загрузки. В конце метода генерируется уведомление о событии вставки строки в таблицу, чтобы оповестить таблицу о добавлении новой строки.

Метод clearDownload()

Метод clearDownload(), показанный ниже, удаляет объект Download из списка управляемых процессов загрузки:

```
// Удаление процесса загрузки из списка.
public void clearDownload(int row) {
  downloadList.remove(row);

  // Генерация для таблицы уведомления об удалении строки.
  fireTableRowsDeleted(row, row);
}
```

После удаления объекта Download из внутреннего списка генерируется уведомление о событии удаления строки таблицы, чтобы оповестить таблицу об удалении строки.

Метод getColumnClass()

Метод getColumnClass(), показанный ниже, возвращает тип класса для данных, отображаемых в определенном столбце:

```
// Получение класса столбца.
public Class getColumnClass(int col) {
  return columnClasses[col];
}
```

Все столбцы отображаются в текстовом виде (т.е. как объекты String) за исключением столбца Progress, который отображается в виде индикатора хода работ (который является объектом типа JProgressBar).

Метод getValueAt()

Метод getValueAt(), показанный ниже, вызывается для получения текущего значения, которое необходимо отобразить в каждой ячейке таблицы.

```
// Получение значения для определенной комбинации строки и столбца.
public Object getValueAt(int row, int col) {
  Download download = downloadList.get(row);
  switch (col) {
    case 0: // URL-адрес
      return download.getUrl();
    case 1: // Размер
      int size = download.getSize();
      return (size == -1) ? "" : Integer.toString(size);
    case 2: // Ход выполнения
      return new Float(download.getProgress());
    case 3: // Состояние
      return Download.STATUSES[download.getStatus()];
  }
  return "";
}
```

Этот метод сначала ищет объект Download, соответствующий указанной строке. Затем указанный столбец используется для того, чтобы определить, какое из значений свойств объекта Download необходимо вернуть.

Метод update()

Метод update(), показанный ниже, заполняет контракт интерфейса Observer, позволяя классу DownloadsTableModel получать уведомления от объектов Download в случае их изменения.

```
/* Вызываем обновление, если процесс загрузки сообщает
   своим наблюдателям о каких-либо изменениях */
public void update(Observable o, Object arg) {
  int index = downloadList.indexOf(o);

  // Генерация для таблицы уведомления об обновлении строки.
  fireTableRowsUpdated(index, index);
}
```

Этот метод передает ссылку на измененный объект Download в виде объекта Observable. Затем в списке процессов загрузки осуществляется поиск индекса этого процесса загрузки. Впоследствии этот индекс будет использоваться для генерации уведомления о событии обновления строки таблицы, которое известит таблицу об обновлении данной строки. После этого таблица повторно визуализирует строку с данным индексом, отображая ее новые значения.

Класс DownloadManager

Теперь, после того как вы смогли детально ознакомиться с вспомогательными классами утилиты Download Manager, можно переходить к рассмотрению класса DownloadManager. Класс DownloadManager отвечает за создание и запуск графического интерфейса Download Manager. Этот класс объявляет метод main(), поэтому при выполнении он будет вызываться первым. Метод main() реализует новый экземпляр класса DownloadManager, а затем вызывает его метод show(), который и приводит к его отображению.

Класс DownloadManager показан ниже. Обратите внимание, что он расширяет класс JFrame и реализует интерфейс Observer. В следующих далее разделах мы проанализируем его более детально.

```
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;

// Класс утилиты Download Manager.
public class DownloadManager extends JFrame
  implements Observer
{

// Добавление текстового поля процесса загрузки.
private JTextField addTextField;

// Модель данных таблицы процесса загрузки.
private DownloadsTableModel tableModel;

// Таблица с перечислением процессов загрузки.
private JTable table;
```

```
// Кнопки для управления выделенным процессом загрузки.
private JButton pauseButton, resumeButton;
private JButton cancelButton, clearButton;
// Выделенный на данный момент процесс загрузки.
private Download selectedDownload;
// Флаг, обозначающий, очищается или нет выбор таблицы.
private boolean clearing;
// Конструктор для Download Manager.
public DownloadManager()
{
  // Определение заголовка приложения.
  setTitle("Download Manager");

  // Определение размеров окна.
  setSize(640, 480);

  // Обработка событий закрытия окна.
  addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
      actionExit();
    }
  });
  // Настройка меню файлов.
  JMenuBar menuBar = new JMenuBar();
  JMenu fileMenu = new JMenu("File");
  fileMenu.setMnemonic(KeyEvent.VK_F);
  JMenuItem fileExitMenuItem = new JMenuItem("Exit",
    KeyEvent.VK_X);
  fileExitMenuItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      actionExit();
    }
  });
  fileMenu.add(fileExitMenuItem);
  menuBar.add(fileMenu);
  setJMenuBar(menuBar);

  // Настройка панели добавления.
  JPanel addPanel = new JPanel();
  addTextField = new JTextField(30);
  addPanel.add(addTextField);
  JButton addButton = new JButton("Add Download");
  addButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      actionAdd();
    }
  });
  addPanel.add(addButton);

  // Настройка таблицы Downloads.
  tableModel = new DownloadsTableModel();
  table = new JTable(tableModel);
  table.getSelectionModel().addListSelectionListener(new
    ListSelectionListener() {
    public void valueChanged(ListSelectionEvent e) {
      tableSelectionChanged();
    }
  });
```

```
// Разрешает выбрать одновременно только одну строку.
table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

// Настройка ProgressBar в качестве визуализатора для столбца
// хода выполнения.
ProgressRenderer renderer = new ProgressRenderer(0, 100);
renderer.setStringPainted(true); // показывает текст хода выполнения
table.setDefaultRenderer(JProgressBar.class, renderer);

// Определение высоты строки таблицы, достаточной для того,
// чтобы уместить JProgressBar.
table.setRowHeight(
  (int) renderer.getPreferredSize().getHeight());

// Настройка панели процессов загрузки.
JPanel downloadsPanel = new JPanel();
downloadsPanel.setBorder(
  BorderFactory.createTitledBorder("Downloads"));
downloadsPanel.setLayout(new BorderLayout());
downloadsPanel.add(new JScrollPane(table),
  BorderLayout.CENTER);

// Настройка панели кнопок.
JPanel buttonsPanel = new JPanel();
pauseButton = new JButton("Pause");
pauseButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    actionPause();
  }
});
pauseButton.setEnabled(false);
buttonsPanel.add(pauseButton);
resumeButton = new JButton("Resume");
resumeButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    actionResume();
  }
});
resumeButton.setEnabled(false);
buttonsPanel.add(resumeButton);
cancelButton = new JButton("Cancel");
cancelButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    actionCancel();
  }
});
cancelButton.setEnabled(false);
buttonsPanel.add(cancelButton);
clearButton = new JButton("Clear");
clearButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    actionClear();
  }
});
clearButton.setEnabled(false);
buttonsPanel.add(clearButton);
```

```
  // Добавление панелей для отображения.
  getContentPane().setLayout(new BorderLayout());
  getContentPane().add(addPanel, BorderLayout.NORTH);
  getContentPane().add(downloadsPanel, BorderLayout.CENTER);
  getContentPane().add(buttonsPanel, BorderLayout.SOUTH);
}

// Выход из программы.
private void actionExit() {
  System.exit(0);
}

// Добавление нового процесса загрузки.
private void actionAdd() {
  URL verifiedUrl = verifyUrl(addTextField.getText());
  if (verifiedUrl != null) {
    tableModel.addDownload(new Download(verifiedUrl));
    addTextField.setText(""); // сброс добавления текстового поля
  } else {
    JOptionPane.showMessageDialog(this,
      "Invalid Download URL", "Error",
      JOptionPane.ERROR_MESSAGE);
  }
}

// Проверка URL-адреса для загрузки.
private URL verifyUrl(String url) {
  // Разрешаем только те URL-адреса, которые включают HTTP.
  if (!url.toLowerCase().startsWith("http://"))
    return null;

  // Проверка формата URL-адреса.
  URL verifiedUrl = null;
  try {
    verifiedUrl = new URL(url);
  } catch (Exception e) {
    return null;
  }

  // Проверяем, содержит ли URL-адрес файл.
  if (verifiedUrl.getFile().length() < 2)
    return null;

  return verifiedUrl;
}

// Вызывается в том случае, если изменяется выбор строки таблицы.
private void tableSelectionChanged() {
  /* Отказ от регистрации на получение уведомлений
     от последнего выбранного процесса загрузки. */
  if (selectedDownload != null)
    selectedDownload.deleteObserver(DownloadManager.this);

  /* Если не происходит очистка процесса загрузки, задаем
     выбранный процесс загрузки и регистрируемся на получение
     от него уведомлений. */
  if (!clearing && table.getSelectedRow() > -1) {
    selectedDownload =
      tableModel.getDownload(table.getSelectedRow());
```

```
      selectedDownload.addObserver(DownloadManager.this);
      updateButtons();
    }
  }

  // Приостановка выбранного процесса загрузки.
  private void actionPause() {
    selectedDownload.pause();
    updateButtons();
  }

  // Возобновление выбранного процесса загрузки.
  private void actionResume() {
    selectedDownload.resume();
    updateButtons();
  }

  // Отмена выбранного процесса загрузки.
  private void actionCancel() {
    selectedDownload.cancel();
    updateButtons();
  }

  // Очистка выбранного процесса загрузки.
  private void actionClear() {
    clearing = true;
    tableModel.clearDownload(table.getSelectedRow());
    clearing = false;
    selectedDownload = null;
    updateButtons();
  }

  /* Обновление состояния каждой кнопки на основании состояния
     выбранного на данный момент процесса загрузки. */
  private void updateButtons() {
    if (selectedDownload != null) {
      int status = selectedDownload.getStatus();
      switch (status) {
        case Download.DOWNLOADING:
          pauseButton.setEnabled(true);
          resumeButton.setEnabled(false);
          cancelButton.setEnabled(true);
          clearButton.setEnabled(false);
          break;
        case Download.PAUSED:
          pauseButton.setEnabled(false);
          resumeButton.setEnabled(true);
          cancelButton.setEnabled(true);
          clearButton.setEnabled(false);
          break;
        case Download.ERROR:
          pauseButton.setEnabled(false);
          resumeButton.setEnabled(true);
          cancelButton.setEnabled(false);
          clearButton.setEnabled(true);
          break;
        default: // COMPLETE или CANCELLED
          pauseButton.setEnabled(false);
```

```
        resumeButton.setEnabled(false);
        cancelButton.setEnabled(false);
        clearButton.setEnabled(true);
      }
    } else {
      // В таблице не выбрано ни одного процесса загрузки.
      pauseButton.setEnabled(false);
      resumeButton.setEnabled(false);
      cancelButton.setEnabled(false);
      clearButton.setEnabled(false);
    }
  }

  /* Вызывается обновление, когда объект Download уведомляет
     своих наблюдателей о каких-либо изменениях. */
  public void update(Observable o, Object arg) {
    // Обновление кнопок в случае изменения выбранного процесса загрузки.
    if (selectedDownload != null && selectedDownload.equals(o))
      updateButtons();
  }

  // Запуск утилиты Download Manager.
    public static void main(String[] args) {
      SwingUtilities.invokeLater(new Runnable() {
        public void run() {
          DownloadManager manager = new DownloadManager();
          manager.setVisible(true);
        }
      });
    }
  }
```

Переменные класса `DownloadManager`

Класс `DownloadManager` начинается с объявления нескольких переменных экземпляра, большинство из которых хранит ссылки на элементы управления графического интерфейса. Переменная `selectedDownload` хранит ссылку на объект `Download`, представляющую выбранную строку в таблице. Переменная экземпляра `clearing` является флагом булевского типа, который отмечает, очищается или нет процесс загрузки из таблицы Downloads.

Конструктор `DownloadManager`

В процессе реализации класса `DownloadManager` все элементы управления графического интерфейса инициализируются внутри его конструктора. Хотя конструктор и содержит большую часть кода, на самом деле он является простым. Сейчас вы в этом убедитесь.

Во-первых, с помощью метода `setTitle()` определяется заголовок окна. При вызове метода `setSize()` задается высота и ширина окна в пикселях. После этого добавляется слушатель окна вызовом метода `addWindowListener()` и передачей ему объекта `WindowAdapter`, в котором переопределен обработчик событий `windowClosing()`. Этот обработчик вызывает метод `actionExit()` во время закрытия окна приложения. В окно приложения добавляется строка меню, содержащая меню File (Файл). Затем устанав-

ливается панель Add (Добавить), которая содержит поле Add Text (Добавить текст) и кнопку. Для кнопки Add Download (Добавить процесс загрузки) добавляется слушатель ActionListener, чтобы можно было вызывать метод actionAdd() при каждом щелчке на кнопке.

После этого создается таблица процессов загрузки. В эту таблицу добавляется слушатель ListSelectionListener, благодаря которому при выборе строки в таблице будет вызываться метод tableSelectionChanged(). Режим выбора таблицы также изменяется на ListSelectionModel.SINGLE_SELECTION, поэтому в таблице можно выделить за один раз только одну строку. Выделение одновременно только одной строки упрощает логику, определяющую, какие кнопки необходимо сделать активными в интерфейсе при выделении строки в таблице процесса загрузки. Затем создается экземпляр класса ProgressRenderer и регистрируется с таблицей для обработки столбца "Progress". Высота строки таблицы обновляется до высоты ProgressRenderer посредством метода table.setRowHeight(). После того как таблица будет готова, с помощью класса JScrollPane в нее добавляются полосы прокрутки, после чего она помещается в панель.

В завершение создается панель кнопок. На ней имеются кнопки Pause (Приостановить), Resume (Возобновить), Cancel (Отменить) и Clear (Очистить). Каждая кнопка добавляет слушатель ActionListener, который при щелчке на данной кнопке вызывает соответствующий метод действия. После создания панели кнопок все созданные панели добавляются в окно.

Метод verifyUrl()

Метод verifyUrl() вызывается методом actionAdd() при добавлении процесса загрузки в утилиту Download Manager. Метод verifyUrl() показан ниже:

```
// Проверка URL-адреса для загрузки.
private URL verifyUrl(String url) {
  // Разрешаем только те URL-адреса, которые включают HTTP.
  if (!url.toLowerCase().startsWith("http://"))
    return null;

  // Проверка формата URL-адреса.
  URL verifiedUrl = null;
  try {
    verifiedUrl = new URL(url);
  } catch (Exception e) {
    return null;
  }

  // Проверяем, содержит ли URL-адрес файл.
  if (verifiedUrl.getFile().length() < 2)
    return null;

  return verifiedUrl;
}
```

Этот метод сначала проверяет, соответствует ли введенный URL-адрес протоколу HTTP, поскольку из всех протоколов поддерживается только он. После проверки URL-адрес используется для создания нового экземпляра класса URL. Если URL-адрес составлен неправильно, конструктор класса URL генерирует исключение. В завершение этот метод проверяет, содержит ли URL-адрес файл.

Метод tableSelectionChanged()

Метод tableSelectionChanged(), показанный ниже, вызывается каждый раз при выделении строки в таблице процессов загрузки.

```
// Вызывается в том случае, если изменяется выбор строки таблицы.
private void tableSelectionChanged() {
  /* Отказ от регистрации на получение уведомлений
     от последнего выбранного процесса загрузки. */
  if (selectedDownload != null)
    selectedDownload.deleteObserver(DownloadManager.this);

  /* Если не происходит очистка процесса загрузки, задаем
     выбранный процесс загрузки и регистрируемся на получение
     от него уведомлений. */
  if (!clearing && table.getSelectedRow() > -1) {
    selectedDownload =
      tableModel.getDownload(table.getSelectedRow());
    selectedDownload.addObserver(DownloadManager.this);
    updateButtons();
  }
}
```

Работа метода начинается с того, что он проверяет, выделена ли в данный момент какая-либо строка. Для этой цели он проверяет, не имеет ли переменная selectedDownload значение null. Если переменная selectedDownload не равна null, то DownloadManager больше не будет получать уведомления в качестве наблюдателя. Затем проверяется флаг clearing. Если таблица не является пустой, и флаг clearing равен false, то переменной selectedDownload присваивается объект Download, соответствующий выделенной строке. После этого DownloadManager регистрируется как наблюдатель Observer с новым выбранным объектом Download. В конце вызывается метод updateButtons() для обновления состояний кнопок на основании выбранного состояния Download.

Метод updateButtons()

Метод updateButtons() обновляет состояние всех кнопок на панели кнопок на основе состояния выделенного процесса загрузки. Метод updateButtons() показан ниже.

```
/* Обновление состояния каждой кнопки на основании состояния
   выбранного на данный момент процесса загрузки. */
private void updateButtons() {
  if (selectedDownload != null) {
    int status = selectedDownload.getStatus();
    switch (status) {
      case Download.DOWNLOADING:
        pauseButton.setEnabled(true);
        resumeButton.setEnabled(false);
        cancelButton.setEnabled(true);
        clearButton.setEnabled(false);
        break;
      case Download.PAUSED:
        pauseButton.setEnabled(false);
        resumeButton.setEnabled(true);
        cancelButton.setEnabled(true);
        clearButton.setEnabled(false);
        break;
```

```
        case Download.ERROR:
          pauseButton.setEnabled(false);
          resumeButton.setEnabled(true);
          cancelButton.setEnabled(false);
          clearButton.setEnabled(true);
          break;
        default: // COMPLETE или CANCELLED
          pauseButton.setEnabled(false);
          resumeButton.setEnabled(false);
          cancelButton.setEnabled(false);
          clearButton.setEnabled(true);
      }
    } else {
      // В таблице не выбрано ни одного процесса загрузки.
      pauseButton.setEnabled(false);
      resumeButton.setEnabled(false);
      cancelButton.setEnabled(false);
      clearButton.setEnabled(false);
    }
  }
```

Если в таблице не выбрано ни одного процесса загрузки, все кнопки становятся неактивными и окрашиваются серым цветом. Если процесс загрузки будет выделен, то состояние каждой кнопки будет обновлено на основе состояния объекта `Download`: `DOWNLOADING`, `PAUSED`, `ERROR`, `COMPLETE` или `CANCELLED`.

Обработка событий действий

Каждый элемент управления графического интерфейса `DownloadManager` регистрирует слушатель `ActionListener`, который вызывает соответствующий метод действия. Слушатели `ActionListener` запускаются каждый раз, когда происходит событие действия в элементе управления графического интерфейса. Например, если пользователь щелкает на кнопке, генерируется событие `ActionEvent`, о котором уведомляется каждый зарегистрированный слушатель кнопки `ActionListener`. Вы могли заметить сходство в работе слушателей `ActionListener` и рассмотренного ранее шаблона `Observer`. Это объясняется тем, что они представляют собой один и тот же шаблон, но имеют разные схемы именования.

Компиляция и запуск утилиты Download Manager

Компиляция `DownloadManager` выполняется следующим образом:

```
javac DownloadManager.java DownloadsTableModel.java ProgressRenderer.java
Download.java
```

Запуск `DownloadManager` производится так, как показано ниже:

```
javaw DownloadManager
```

Использовать утилиту Download Manager несложно. Для начала в текстовом поле в верхней части экрана нужно ввести URL-адрес файла, который требуется загрузить.

Например, чтобы загрузить файл `0072229713_code.zip`, хранящийся на сайте `www.osborne.com`, нужно ввести следующее:

```
http://www.osborne.com/products/0072229713/0072229713_code.zip
```

Это файл, в котором содержится код для моей книги *The Art of Java*, которую я написал в соавторстве с Джеймсом Холмсом (James Holmes).

После того как процесс загрузки будет добавлен в утилиту Download Manager, вы сможете управлять им, выделяя его в таблице. После того как процесс будет выделен, вы можете приостановить его, отменить, возобновить и очистить. На рис. 33.2 показана утилита Download Manager в действии.

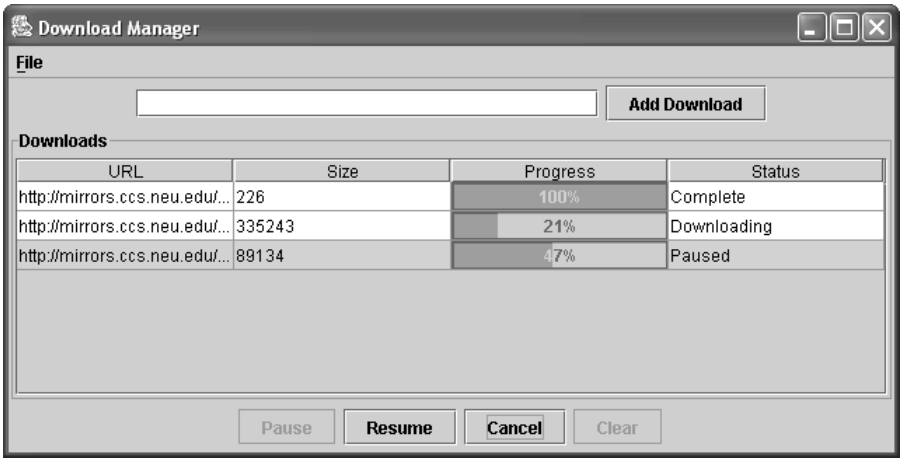

Рис. 33.2. Утилита Download Manager в действии

Расширение утилиты Download Manager

Утилита Download Manager представляет собой полнофункциональное средство, позволяющее приостанавливать и возобновлять процессы загрузки, а также загружать несколько файлов одновременно; тем не менее, вы можете расширить ее возможности и сделать ее еще лучше. Смотрите, что еще можно сделать: добавить поддержку прокси-серверов, FTP и HTTPS, а также поддержку технологии перетаскивания. Интересной особенностью является планирование, благодаря которому вы сможете запланировать процесс загрузки на определенное время (например, на час ночи), когда можно использовать большое количество свободных системных ресурсов.

Учтите, что продемонстрированные в этой главе методы не ограничены загрузкой файлов в обычном смысле. Этому коду можно найти еще множество применений. Например, многие программы, распространяемые через Internet, поступают в двух частях. Первая часть имеет небольшой размер и представляет собой компактное приложение, которое можно загрузить очень быстро. Это приложение содержит мини-диспетчер загрузки для загрузки второй части, которая обычно имеет гораздо большие размеры. Такой подход удобно использовать для больших приложений, при загрузке которых возрастает вероятность прерывания. Возможно, вы решите адаптировать утилиту Download Manager для этих целей.

A

ПРИЛОЖЕНИЕ

Использование комментариев документации

Как было сказано в первой части книги, Java поддерживает три типа комментариев. Первые два — это комментарии `//` и `/* */`. Третий тип называется *комментарием документации*. Такой комментарий начинается с последовательности символов `/**` и заканчивается последовательностью `*/`. Комментарии документации позволяют добавлять в программу информацию о ней самой. Позже с помощью утилиты `javadoc` (входящей в состав JDK) эту информацию можно будет извлекать и помещать в HTML-файл. Комментарии документации делают удобным процесс написания документации к разрабатываемым программам. Вы должны были встречать документацию, сгенерированную утилитой `javadoc`, поскольку корпорация Sun использовала именно эту утилиту для документирования библиотеки Java API.

Дескрипторы утилиты `javadoc`

Утилита `javadoc` распознает дескрипторы, перечисленные в табл. А.1.

Таблица А.1. Дескрипторы утилиты `javadoc`

Дескриптор	Назначение
`@author`	Идентифицирует автора класса.
`{@code}`	Отображает информацию “как есть”, без обработки HTML-стилей, используя шрифт кода.
`@deprecated`	Определяет, что класс или член класса является устаревшим.
`{@docRoot}`	Определяет путь к корневому каталогу текущей документации.
`@exception`	Идентифицирует исключение, сгенерированное методом.
`{@inheritDoc}`	Наследует комментарий от непосредственного суперкласса.
`{@link}`	Вставляет встроенную ссылку на другую тему.

Окончание табл. А.1

Дескриптор	Назначение
`{@linkplain}`	Вставляет встроенную ссылку на другую тему, при этом ссылка отображается с помощью открытой формы шрифта.
`{@literal}`	Отображает информацию “как есть”, без обработки HTML-стилей.
`@param`	Документирует параметр метода.
`@return`	Документирует возвращаемое значение метода.
`@see`	Определяет ссылку на другую тему.
`@serial`	Документирует поле, сериализируемое по умолчанию.
`@serialData`	Документирует данные, записанные методами `writeObject()` и `writeExternal()`.
`@serialField`	Документирует компонент `ObjectStreamField`.
`@since`	Показывает, в каком выпуске было введено определенное изменение.
`@throws`	То же, что и `@exception`.
`{@value}`	Отображает значение константы, которая должна быть полем `static`.
`@version`	Определяет версию класса.

Дескрипторы `javadoc`, начинающиеся со знака `@`, называются автономными и должны использоваться в своей собственной строке. Дескрипторы, начинающиеся с фигурной скобки, например `{@code}`, называются встроенными и могут применяться внутри большего описания. В комментариях документации можно использовать и другие, стандартные HTML-дескрипторы. Однако некоторые дескрипторы, такие как заголовки, нельзя использовать, потому что они нарушают вид HTML-файла, сформированный утилитой `javadoc`.

Комментарии документации можно применять для документирования классов, интерфейсов, полей, конструкторов и методов. В каждом из случаев комментарий документации должен стоять перед документируемым элементом. Для документирования переменной можно использовать следующие дескрипторы: `@see`, `@serial`, `@serialField`, `{@value}` и `@deprecated`. Для классов и интерфейсов можно использовать дескрипторы `@see`, `@author`, `@deprecated`, `@param` и `@version`. Методы можно документировать с помощью дескрипторов `@see`, `@return`, `@param`, `@deprecated`, `@throws`, `@serialData`, `{@inheritDoc}` и `@exception`. Дескрипторы `{@link}`, `{@docRoot}`, `{@code}`, `{@literal}`, `@since` или `{@linkplain}` могут применяться где угодно. Сейчас мы рассмотрим каждый из этих дескрипторов.

$author

Дескриптор `@author` документирует автора класса. Он имеет следующий синтаксис:

```
@author описание
```

Здесь *описание* обычно представляет фамилию человека, написавшего класс. При выполнении утилиты `javadoc` вам нужно будет задать опцию `-author`, чтобы поле `@author` включить в HTML-документацию.

{@code}

Дескриптор {@code} позволяет встраивать в комментарий текст (например, фрагмент кода). Этот текст будет отображаться с помощью шрифта кода без последующей обработки (например, без HTML-визуализации). Он имеет следующий синтаксис:

```
{@code фрагмент_кода}
```

@deprecated

Дескриптор @deprecated определяет, что класс, интерфейс или член класса является устаревшим. Рекомендуется включать дескрипторы @see или {@link} для того, чтобы информировать программиста о доступных альтернативных вариантах. Синтаксис этого дескриптора выглядит следующим образом:

```
@deprecated описание
```

Здесь *описание* — это сообщение, описывающее исключение. Дескриптор @deprecated может использоваться для документирования переменных, методов и классов.

{@docRoot}

Дескриптор {@docRoot} определяет путь к корневому каталогу текущей документации.

@exception

Дескриптор @exception описывает исключение для данного метода. Он имеет следующий синтаксис:

```
@exception имя_исключения пояснение
```

Здесь *имя_исключения* указывает полное имя исключения, а *пояснение* представляет строку, которая описывает, в каких случаях может возникнуть данное исключение. Дескриптор @exception может использоваться только для документирования методов.

{@inheritDoc}

Этот дескриптор наследует комментарий от непосредственного суперкласса.

{@link}

Дескриптор {@link} предлагает встроенную ссылку на дополнительную информацию. Он имеет следующий синтаксис:

```
{@link пакет.класс#член текст}
```

Здесь *пакет.класс#член* определяет имя класса или метода, на который добавляется ссылка, а *текст* представляет отображаемую строку.

{@linkplain}

Вставляет встроенную ссылку на другую тему. Ссылка отображается в открытой форме шрифта. В противном случае дескриптор аналогичен {@link}.

{@literal}

Дескриптор {@literal} позволяет встраивать текст в комментарий. Этот текст отображается "как есть" без последующей обработки (например, без HTML-визуализации). Он имеет следующий синтаксис:

```
{@literal описание}
```

Здесь *описание* представляет встраиваемый текст.

@param

Дескриптор @param документирует параметр для метода или параметр типа для класса или интерфейса. Он имеет следующий синтаксис:

```
@param имя_параметра пояснение
```

Здесь *имя_параметра* представляет имя параметра. Назначение этого параметра определяется соответствующим пояснением. Дескриптор @param может использоваться только для документирования метода, конструктора или обобщенного класса или интерфейса.

@return

Дескриптор @return описывает возвращаемое значение метода. Он имеет следующий синтаксис:

```
@return пояснение
```

Здесь *пояснение* описывает тип и смысл значения, возвращаемого методом. Дескриптор @return может использоваться только для документирования метода.

@see

Дескриптор @see обеспечивает ссылку на дополнительную информацию. Далее показаны наиболее распространенные его формы:

```
@see привязка
@see пакет.класс#член текст
```

В первой форме *привязка* представляет ссылку на абсолютный или относительный URL-адрес. Во второй форме *пакет.класс#член* определяет имя элемента, а *текст* представляет отображаемый для данного элемента текст. Текстовый параметр необязателен и если не используется, то отображается элемент, указанный в параметре *пакет.класс#член*. Имя члена тоже является необязательным. Таким образом, вы можете определить ссылку на модуль, класс или интерфейс в дополнение к ссылке на определенный метод или поле. Имя может быть определено полностью или частично. Однако точку, стоящую перед именем члена (если таковой существует), необходимо заменить символом #.

@serial

Дескриптор @serial определяет комментарий для поля, сериализируемого по умолчанию. Он имеет следующий синтаксис:

```
@serial описание
```

Здесь *описание* определяет комментарий для данного поля.

@serialData

Дескриптор `@serialData` документирует данные, записанные с помощью методов `writeObject()` и `writeExternal()`. Он имеет следующий синтаксис:

```
@serialData описание
```

Здесь *описание* определяет комментарий для этих данных.

@serialField

Для класса, реализующего `Serializable`, дескриптор `@serialField` предлагает комментарии для компонента `ObjectStreamField`. Он имеет следующий синтаксис:

```
@serialField имя тип описание
```

Здесь *имя* представляет имя поля, *тип* представляет его тип, а *описание* — комментарий для данного поля.

@since

Дескриптор `@since` показывает, что класс или член класса был впервые представлен в определенном выпуске. Он имеет следующий синтаксис:

```
@since выпуск
```

Здесь *выпуск* представляет строку, в которой указан выпуск или версия, начиная с которого эта особенность стала доступной.

@throws

Дескриптор `@throws` имеет то же назначение, что и дескриптор `@exception`.

{@value}

Дескриптор `{@value}` имеет две формы. Первая отображает значение следующей за ним константы, которой должно являться поле `static`. Его форма показана ниже:

```
{@value}
```

Вторая форма отображает значение определенного поля `static`. В этом случае дескриптор имеет следующую форму:

```
{@value пакет.класс#поле}
```

Здесь *пакет.класс#поле* определяет имя `static`-поля.

@version

Дескриптор `@version` определяет версию класса. Он имеет следующий синтаксис:

```
@version информация
```

Здесь *информация* представляет строку, содержащую информацию о версии (как правило, номер версии, например 2.2). При выполнении утилиты `javadoc` вам нужно будет указать опцию `-version`, чтобы поле `@version` включить в HTML-документацию.

Общая форма комментариев документации

После начальной комбинации символов /** первая строка или строки становятся главным описанием вашего класса, переменной или метода. После них можно включать один или более различных дескрипторов @. Каждый дескриптор @ должен стоять в начале новой строки или следовать за одним или несколькими символами звездочки (*), находящимися в начале строки. Несколько дескрипторов одного и того же типа необходимо группировать вместе. Например, если у вас имеется три дескриптора @see, их следует поместить друг за другом. Встроенные дескрипторы (они начинаются с фигурной скобки) можно помещать внутри любого описания.

Ниже показан пример комментария документации для класса:

```
/**
* Этот класс рисует секторную диаграмму.
* @author Герберт Шилдт
* @version 3.2
*/
```

Вывод утилиты javadoc

Утилита javadoc в качестве входных данных принимает файл с исходным кодом вашей Java-программы и выводит несколько HTML-файлов, содержащих документацию по этой программе. Информация о каждом классе будет содержаться в его собственном HTML-файле. Утилита javadoc выводит также дерево индексов и иерархии. Могут быть сгенерированы и другие HTML-файлы.

Пример использования комментариев документации

Ниже приведена простая программа, в которой используются комментарии документации. Обратите внимание, что каждый комментарий стоит непосредственно перед тем элементом, который он описывает. После того как документация по классу SquareNum будет обработана утилитой javadoc, ее можно будет найти в файле SquareNum.html.

```
import java.io.*;

/**
 * Этот класс демонстрирует применение комментариев документации.
 * @author Герберт Шилдт (Herbert Schildt)
 * @version 1.2
 */
public class SquareNum {
  /**
   * Этот метод возвращает квадрат числа.
   * Это многострочное описание. Вы можете использовать
   * столько строк, сколько будет необходимо.
   * @param num Значение, которое необходимо возвести в квадрат.
   * @return num Значение, возведенное в квадрат.
  */
  public double square(double num) {
    return num * num;
  }
```

```
  /**
   * Этот метод вводит число, полученное от пользователя.
   * @return Введенное значение в виде double.
   * @exception В случае ошибки ввода генерируется исключение IOException.
   * @see IOException
   */
  public double getNumber() throws IOException {
    // Создает BufferedReader с помощью 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 square().
   * @param args Не используется.
   * @exception В случае ошибки ввода генерируется исключение 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("Значение в квадрате равно " + val);
  }
}
```

Предметный указатель

K

Л

М

П

Научно-популярное издание

Герберт Шилдт

Полный справочник по Java 7-е издание

Верстка *Т.Н. Артеменко*
Художественный редактор *В.Г. Павлютин*

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

Подписано в печать 09.09.2008. Формат 70×100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 83,85. Уч.-изд. л. 61,20.
Доп. тираж 1000 экз. Заказ № 0000.

Отпечатано по технологии CtP
в ОАО “Печатный двор” им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.