Глава 2

Автоупаковка и автораспаковка

Изучение новых функциональных возможностей, включенных в последнюю версию Java 2 5.0, начнем с так долго ожидаемых всеми программистами на языке Java автоупаковки (*autoboxing*) и автораспаковки (*auto-unboxing*). Выбор этот сделан по трем причинам. Во-первых, автоупаковка/автораспаковка сильно упрощает и рационализирует исходный код, в котором требуется объектное представление *базовых* типов языка Java, таких как *int* или *char*. Поскольку такие ситуации часто встречаются в текстах программ на Java, выигрыш от применения средств автоупаковки/распаковки получат практически все, программирующие на этом языке. Во-вторых, автоупаковка/автораспаковка во многом способствует простоте и удобству применения другого нового средства — настройки типов (*generics*). Следовательно, понимание автоупаковки и автораспаковки понадобится для последующего изучения этого механизма. В-третьих, автоупаковка/распаковка плавно изменяет наши представления о взаимосвязи объектов и данных базовых типов. Эти изменения гораздо глубже, чем может показаться на первый взгляд из-за концептуальной простоты двух описываемых здесь новых функциональных возможностей. Их влияние ощущается во всем языке Java.

Автоупаковка и автораспаковка напрямую связаны с *оболочками типов* (*type wrapper*) языка Java и со способом вставки значений в экземпляры таких оболочек и извлечения значений из них. По этой причине мы начнем с краткого обзора оболочек типов и процесса упаковки и распаковки значений для них.

Обзор оболочек типов и упаковки значений

Как вам известно, в языке Java используются базовые типы (также называемые простыми), такие как *int* или *double*, для хранения элементарных данных типов, поддерживаемых языком. Для хранения таких данных из-за более высокой производительности применяются базовые типы, а не объекты. В этом случае использование объектов добавляет неприемлемые издержки даже к простейшей вычислительной операции. Таким образом, базовые типы не являются частью иерархии объектов и не наследуют класс *Object*.

Несмотря на выигрыш в производительности, предлагаемый базовыми типами, возникают ситуации, требующие обязательного объектного представления. Например, Вы не можете передать в метод переменную базового типа как параметр по ссылке. Кроме того, много стандартных структур данных, реализованных в языке Java, оперирует объектами, и, следовательно, Вы не можете использовать эти структуры для хранения данных базовых типов. Для подобных (и других) случаев Java предоставляет оболочки типов, представляющие собой классы, которые инкапсулируют данные простого типа в объект. Далее перечислены классы оболочки типов.

```
Boolean    Byte        Character    Double
```

```
Float      Long       Integer    Short
```

Значение базового типа инкапсулируется в оболочку в момент конструирования объекта. Упакованное таким образом значение можно получить обратно с помощью вызова одного из методов, определенных в оболочке. Например, все оболочки числовых типов предлагают следующие методы:

```
byte byteValue()      double doubleValue()     float floatValue()
int intValue()        long longValue()         short shortValue()
```

Каждый метод возвращает значение заданного базового типа. Например, объект типа *Long* может вернуть значение одного из встроенных числовых типов, включая *short*, *double* или *long*.

Процесс инкапсуляции значения в объект называется *упаковкой* (*boxing*). До появления Java 2 версии 5.0 вся упаковка выполнялась программистом вручную, с помощью создания экземпляра оболочки с нужным значением. В приведенной далее строке кода значение 100 упаковывается вручную в объект типа *Integer*:

```
Integer iOb = new Integer(l00);
```

В приведенном примере новый объект типа *Integer* со значением 100 создается явно и ссылка на него присваивается переменной *iOb*.

Процесс извлечения значения из оболочки типа называется *распаковкой* (unboxing). И снова, до появления Java 2 версии 5.0 вся распаковка выполнялась вручную с помощью вызова метода оболочки для получения значения из объекта.

В следующей строке кода значение из объекта *iOb* вручную распаковывается в переменную типа *int*:

```
int i = iOb.intValue();
```

В данном случае метод `intValue()` возвращает значение типа `int` из объекта `iOb`. Как объяснялось ранее, есть и другие методы, позволяющие извлечь из объекта значение другого числового типа, такого как *byte*, *short*, *long*, *double* или *float*. Например, для получения значения типа *long* из объекта `iOb` Вам следует вызвать метод `iOb.longValue()`. Таким образом, можно распаковать значение в переменную простого типа, отличающегося от типа оболочки.

Начиная с первоначальной версии языка Java, для упаковки и распаковки вручную выполнялась одна и та же базовая процедура, приведенная в предыдущих примерах. Хотя такой способ упаковки и распаковки работает, он утомителен и подвержен ошибкам, так как требует от программиста вручную создавать подходящий объект для упаковки значения и при необходимости его распаковки явно задавать переменную соответствующего базового типа. К счастью Java 2, v5.0 коренным образом модернизирует эти важнейшие процедуры, вводя средства автоупаковки/распаковки.

Основы автоупаковки/распаковки

Автоупаковка (*autoboxing*) — это процесс автоматической инкапсуляции данных простого типа, такого как *int* или *double*, в эквивалентную ему оболочку типа, как только понадобится объект этого типа. При этом нет необходимости в явном создании объекта нужного типа. *Автораспаковка* (*auto-unboxing*) — это процесс автоматического извлечения из упакованного объекта значения, когда оно потребуется. Вызовы методов, таких как *intValue()* и *doubleValue()*, становятся ненужными.

Добавление средств автоупаковки/автораспаковки значительно упрощает кодирование ряда алгоритмов, исключая утомительные упаковку и распаковку, выполняемые вручную. Кроме того, эти новые средства программирования позволяют избежать ошибок за счет устранения возможности распаковки вручную неверного типа из оболочки. Автоупаковка также облегчает использование настраиваемых типов (*generic*s) и запоминание данных базовых типов в коллекциях.

Благодаря автоупаковке исчезает необходимость в создании вручную объекта для инкапсуляции значения простого типа. Вам нужно только присвоить это значение указателю на объект типа-оболочки. Язык Java автоматически создаст для вас этот объект. В следующей

строке приведен пример современного способа конструирования объекта типа *Integer*, хранящего значение 100:

```
Integer iOb = 100; // автоматически упаковывает значение типа int
```

Обратите внимание на то, что никакого объекта не создается явно, с помощью операции new. Язык Java выполнит это автоматически.

Для автораспаковки объекта просто присвойте ссылку на него переменной соответствующего базового типа. Например, для распаковки объекта *iOb* можно использовать следующую строку кода:

```
int i = iOb; // автораспаковка
```

Все детали выполнит для вас язык Java.

В листинге 2.1 приведена короткая программа, вобравшая в себя все приведенные ранее фрагменты и демонстрирующая основы механизма автоупаковки/распаковки,

Листинг 2.1. Демонстрация применения автоупаковки/распаковки

```
// Demonstrate autoboxing/unboxing.
class AutoBox {

  public static void main(String args[]) {

    Integer iOb = 100; ; // автоупаковка значения типа int

    int i = iOb; // автораспаковка

    System.out.println(i + " " + iOb);  // отображает на экране: 100 100

  }
}
```

Обратите внимание еще раз на то, что не нужно явно создавать объект типа *Integer* для упаковки значения 100 и нет необходимости вызывать метод *intValue()* для распаковки этого значения.

Автоупаковка и методы

Помимо простых случаев присваивания, автоупаковка выполняется автоматически каждый раз, когда данные базового типа должны быть преобразованы в объект, а автораспаковка — при необходимости преобразования объекта в значение базового типа. Следовательно, автоупаковка/распаковка может происходить, когда аргумент передается в метод или когда значение возвращается методом. Рассмотрим пример, приведенный в листинге 2.2.

Листинг 2.2. Автоупаковка/распаковка параметров метода и возвращаемых им значений

```
// Autoboxing/unboxing takes place with
// method parameters and return values.

class AutoBox2 {

  // Принимает параметр типа Integer и возвращает

  // значение типа int;

  static int m(Integer v) {
    return v ; // auto-unbox to int
  }
```

```
  public static void main(String args[]) {
     // Передает значение int в метод  m()   и присваивает возвращаемое
     // значение объекту типа Integer.     Здесь аргумент 100
    // автоупаковывается в объект типа Integer.   Возвращаемое значение
    // также автоупаковывается в тип Integer.

    Integer iOb = m(100);

     System.out.println(iOb);
  }

}
```

Программа листинга 2.2 отображает следующий ожидаемый результат:

```
100
```

В приведенной программе метод задает параметр типа *Integer* и возвращает результат типа *int*. В теле *main()* методу *m( )* передается значение 100. Поскольку ожидает объект типа *Integer*, передаваемое значение автоматически упаковывается. Далее метод то возвращает эквивалент своего аргумента, но простого типа *int*. Это приводит к автоматической распаковке в переменную *v*. Далее в методе *main()* объекту *iOb* присваивается это значение типа *int*, что вызывает его автоупаковку. Главное преимущество заключается в том, что все преобразования выполняются автоматически.

Автоупаковка/распаковка в выражениях

Вообще, автоупаковка/распаковка происходит всегда, когда требуется преобразование в объект или из объекта. Это применимо и к выражениям. В них числовой объект автоматически распаковывается. Результат выражения повторно упаковывается, если это необходимо. Рассмотрим программу, приведенную в листинге 2.3.

Листинг 2.3. Автоупаковка/распаковка внутри выражений

```
class AutoBox3 {
  public static void main(String args[]) {

    Integer iOb, iOb2;
    int i;

    iOb = 100;
    System.out.println("Original value of iOb: " + iOb);


 // Далее автоматически распаковывается объект iOb,
 // выполняется операция инкремента, затем результат
 // повторно упаковывается в объект  iOb.

    ++iOb;

    System.out.println("After ++iOb: " + iOb);


// Здесь iOb распаковывается, выражение
```

```
// вычисляется и результат повторно упаковывается
// и присваивается iOb2.

    iOb2 = iOb + (iOb / 3);

    System.out.println("iOb2 after expression: " + iOb2);

// То же самое выражение вычисляется, но результат
// повторно не упаковывается.

    i = iOb + (iOb / 3);
    System.out.println("i after expression: " + i);

  }
}
```

Далее приведен вывод программы, отображающий результаты ее работы.

```
Original value of iOb: 100
After ++iOb: 101
iOb2 after expression: 134
i after expression: 134
```

Обратите особое внимание на следующую строку программы из листинга 2.3:

```
++iOb;
```

Она вызывает увеличение на единицу значения, упакованного в объекте *iOb*. Это действие выполняется следующим образом: объект *iOb* распаковывается, значение увеличивается, и результат повторно упаковывается.

Автораспаковка позволяет смешивать в выражении числовые объекты разных типов. После того как значения распакованы, к ним применимы стандартные преобразования типов и переходы от одного к другому. Например, приведенная в листинге 2.4 программа вполне корректна.

Листинг 2.4. Обработка числовых объектов разных типов в одном выражении

```
class AutoBox4 {
  public static void main(String args[]) {

    Integer iOb = 100;
    Double dOb = 98.6;

    dOb = dOb + iOb;
    System.out.println("dOb after expression: " + dOb);
  }
}
```

Далее приведен результат работы программы из листинга 2.4.

```
iOb after expression: 198.6
```

Как видите, и объект *dOb* типа *Double*, и объект *iOb* типа *Integer* участвовали в сложении, а результат был повторно упакован и сохранен в объекте *dOb*.

Благодаря автораспаковке Вы можете использовать целочисленный объект для управления оператором *switch*. Рассмотрим следующий фрагмент программы:

```
Integer iOb = 2 ;

switch (iOb) {
    case 1: System.out.println("one");
      break;
    case 2: System.out.println("two") ;
      break;
    default: System.out.println("error") ;
```

Когда вычисляется выражение в операторе *switch*, распаковывается объект *iOb* и извлекается значение типа i*nt*.

Приведенные примеры программ показывают, что наличие автоупаковки/распаковки делает использование числовых объектов в выражении легким и интуитивно понятным. В прошлом в этот код пришлось бы вставлять вызовы методов, подобных *intValue()*.

Автоупаковка/распаковка логических и символьных значений

Кроме оболочек для числовых типов язык Java также предоставляет оболочки для данных типов *boolean* и char. Они называются *Boolean* и *Character* соответственно. К ним также применимы автоупаковка/распаковка. Рассмотрим программу, приведенную в листинге 2.5.

Листинг 2.5. Автоупаковка/распаковка типов Boolean и Character

```
class AutoBox5 {
  public static void main(String args[]) {

    // Автоупаковка/распаковка логических переменных.
    Boolean b = true;


    // Далее объект b автоматически распаковывается, когда используется
    // в условном выражении оператора, такого как if.
    if(b) System.out.println("b is true");

    // Автоупаковка/распаковка символьных переменных.


    Character ch = 'x'; // box a char
    char ch2 = ch; // unbox a char

    System.out.println("ch2 is " + ch2);
  }
}
```

Далее приведен вывод программы из листинга 2.5, отображающий результаты ее работы:

```
b is true
ch2 is x
```

Наиболее важной в программе из листинга 2.5 является автораспаковка объекта *b* внутри условного выражения в операторе *if*. Как вы должны помнить, условное выражение, управляющее выполнением оператора *if*, следует вычислять как значение типа *boolean*. Благодаря наличию автораспаковки логическое значение, содержащееся в объекте *b*, автоматически распаковывается при вычислении условного выражения. Таким образом, с

появлением Java 2 v5.0 стало возможным использование объекта типа *Boolean* для управления оператором *if*.

Более того, теперь объект типа *Boolean* можно применять для управления любыми операторами цикла языка Java. Когда объект типа *Boolean* используется как условное выражение в циклах *while*, *for*, do/while, он автоматически распаковывается в эквивалент простого типа *boolean*. Например, приведенный далее фрагмент теперь абсолютно корректен.

```
Boolean b;
//
while (b)  { //
```

Помощь автоупаковки/распаковки в предупреждении ошибок

Кроме удобства, которое предоставляет механизм автоупаковки/распаковки, он может помочь в предупреждении ошибок. Рассмотрим программу, приведенную в листинге 2.6.

Листинг 2.6. Ошибка, возникшая при распаковке вручную

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


    Integer iOb = 1000;

    // автоматически упаковывает значение 1000

    int i = iOb.byteValue();// вручную распаковывается как тип byte !!!

    System.out.println(i);// не отображает значение 1000
  }
}
```

Программа из листинга 2.6 отображает число -24 вместо ожидаемого значения 1000! Причина заключается в том, что значение, хранящееся в объекте *iOb*, распаковывается вручную с помощью вызова метода *byteValue()* который приводит к усечению этого значения, равного 1000. В результате переменной *i* присваивается число -24, так называемый "мусор". Автораспаковка препятствует возникновению ошибок этого типа, потому что она преобразует значение, хранящееся в *iOb*, в величину базового типа, сопоставимого с типом *int*.

Вообще говоря, поскольку автоупаковка всегда создает правильный объект, а автораспаковка всегда извлекает надлежащее значение, не возникает ситуаций для формирования неверного типа объекта или значения. В редких случаях, когда Вам нужен тип, отличающийся от созданного автоматическим процессом, Вы и сейчас можете упаковывать и распаковывать значения вручную так, как делали это раньше. Конечно, при этом теряются преимущества автоупаковки/распаковки. Как правило, во вновь разрабатываемом коде должны применяться эти механизмы, так как они соответствуют современному стилю программирования на языке Java.

Предостережения

Теперь, когда в язык Java включены средства автоупаковки/распаковки, может появиться желание использовать только числовые объекты типа *Integer* или *Double*, полностью отказавшись от данных простых типов. Например, благодаря наличию автоупаковки/распаковки теперь можно написать код, подобный приведенному далее.

```
//Пример плохого использования автоупаковки/распаковки
Double a,b,c;

a = 10.0;

b = 4.0;

c = Math.sqrt(a*a + b*b);

System.out.println("Hypotenuse is " + c);
```

В приведенном примере объекты типа *Double* содержат значения, которые используются для вычисления гипотенузы прямоугольного треугольника. Хотя этот код технически корректен и будет выполняться правильно, он служит образцом очень плохого применения автоупаковки/распаковки. Гораздо эффективнее использовать данные простого типа *double* для подобных вычислений, т. к. каждая автоупаковка и автораспаковка вносят дополнительные затраты, которых лишены вычисления с применением базовых типов данных.

Вообще говоря, следует ограничить использование оболочек типов только теми случаями, для которых требуется объектное представление данных простых типов. Автоупаковка/автораспаковка включены в язык таким образом, чтобы не ограничивать применение простых типов данных.

Глава 3

Настраиваемые типы

Среди множества расширений языка, включенных в Java 2 версии 5.0, средства настройки типов (*generics*) оказали на язык наиболее глубокое влияние. Они не только добавили новый синтаксический элемент в язык Java, но и вызвали изменения во многих классах и методах *API* (*Application Programming Interface*, Интерфейс прикладного программирования) ядра. Благодаря применению *настраиваемых типов* стало возможным создавать классы, интерфейсы и методы, работающие с различными типами данных, при этом обеспечивая безопасность типов. Многие алгоритмы логически идентичны вне зависимости от используемых типов данных. Например, механизм поддержки стека одинаков для стеков, хранящих элементы типа *Integer, String, Object* или *Thread*. С помощью настраиваемых типов Вы можете определить один алгоритм независимо от конкретного типа данных и затем применять его без дополнительной доработки к различным типам данных. Функциональные возможности настраиваемых типов коренным образом меняют подход к написанию программ на языке Java.

Пожалуй, включение в язык настраиваемых типов оказало наибольшее влияние на средства работы с группами объектов (*Collections Framework*). Как известно, в подсистеме *Collections Framework* определено несколько классов, таких как списки (*list*) и отображения (*map*), которые управляют обработкой, коллекций. Классы, описывающие коллекции, могут использоваться для любого типа объекта. Введение настраиваемых типов обеспечивает этим классам полную *типовую*

безопасность (*type safety*). Таким образом, кроме включения в язык новой мощной функциональной возможности, применение настраиваемых типов позволяет существенно улучшить использование средств, уже существующих в языке. Именно поэтому настраиваемые типы представляют собой столь важное расширение языка Java.

Что такое настраиваемые типы

По существу настраиваемые типы — это *параметризованные типы* (*parameterized type*). Они очень важны, так как позволяют Вам создавать классы,интерфейсы и методы, в которых тип обрабатываемых данных задается как параметр. С помощью настраиваемых типов можно, например, создать один класс, автоматически оперирующий разными типами данных. Класс, интерфейс или метод, работающие с параметризованными типами, называются настраиваемыми (*generic*).

Важно помнить, что в языке Java, всегда существовала возможность создавать обобщенные или универсальные (*generalized*) классы, интерфейсы и методы с помощью ссылки на тип Object. Поскольку тип *Object* — это суперкласс для всех остальных классов, ссылка на него, по сути, позволяет ссылаться на объект любого типа. Таким образом, в исходных текстах программ, существовавших до появления настраиваемых типов, обобщенные классы, интерфейсы и методы использовали ссылки на тип *Object* для работы с различными типами данных. Проблема заключалась в том, что подобная обработка не обеспечивала типовой безопасности.

Средства настройки типов обеспечили недостающую типовую безопасность. Они также модернизировали процесс, потому что отпала необходимость в явном приведении типов (*Type cast*) при переходе от типа *Object* к конкретному обрабатываемому типу. Благодаря введению настраиваемых типов все приведения выполняются автоматически и скрыто. Таким образом, применение настраиваемых типов расширяет возможности повторного использования кода и делает этот процесс легким и безопасным.

Предупреждение

> Предупреждение программистам на языке C++: несмотря на то, что настраиваемые типы подобны шаблонам в C++, это не одно и то же. Существуют принципиальные различия между двумя подходами к настраиваемым типам (*generic type*) в этих языках. Если Вы знаете язык C++, важно не спешить с выводами по поводу принципов работы настраиваемых типов в Java.

Простой пример применения настраиваемых типов

Давайте начнем с простого примера, настраиваемого или полиморфного класса. В приведенной в листинге 3.1 программе определены два класса. Первый — настраиваемый класс *Gen*, второй — *GenDemo*, использующий класс *Gen*.

Листинг 3.1. Пример простого настраиваемого класса

```
//T – тип, объявленный как параметр, и
// будет заменен реальным типом
// при создании объекта типа
class Gen<T> {
  T ob; // объявляет объект типа T
  // Передает конструктору ссылку на
  // объект типа
  Gen(T o) {
    ob = o;
  }
  // Возвращает объект
  T getob() {
    return ob;
  }
```

```
  // Показывает тип T.
  void showType() {
    System.out.println("Type of T is " + ob.getClass().getName());
  }
}

// Демонстрирует настраиваемый класс.
class GenDemo {
   public static void main(String args[]) {
     // Создает ссылку на Gen для объектов типа Integer.
     Gen<Integer> iOb;

     // Создает объект типа Gen<Integer> и присваивает
     // ссылку на него iOb.  Обратите внимание на использование
     // автоупаковки (autoboxing)
     // для инкапсуляции значения 88 в объекте типа Integer.

     iOb = new Gen<Integer>(88);

      // Показывает тип данных iOb.
      iOb.showType();

      // Получает значение, хранящееся в iOb. Обратите внимание на то,
      // что не требуется никакого приведения типов.
     int v = iOb.getob();
     System.out.println("value: " + v);

     System.out.println();

     // Создает объект типа Gen для строк.
     Gen<String> strOb = new Gen<String>("Generics Test");

     // Показывает тип данных переменной strOb.
     strOb.showType();

     // Получает значение, хранящееся в strOb. И опять
     // никакого приведения типов не требуется.
     String str = strOb.getob();
     System.out.println("value: " + str);
  }
}
```

Далее приведен вывод, формируемый программой.

```
Type of T is java.lang.Integer
value: 88

Type of T is java.lang.String
value: Generics Test
```

Давайте подробно обсудим программу из листинга 3.1.

Прежде всего, обратите внимание на приведенное в следующей строке объявление типа *Gen*.

```
Class Gen <T>   {
```

Здесь *T — тип, объявленный как параметр*, или *параметр типа* (*type parameter*). Это имя используется как заменитель действительного типа, передаваемого в класс Gen при создании объекта. Таким образом, имя *T* при описании класса используется везде, где требуется параметр для типа. Обратите внимание на то, что имя *T* обрамляется символами: <>. Эту синтаксическую конструкцию можно обобщить: всегда при объявлении типа как параметра его имя заключается в угловые скобки. Поскольку класс *Gen* использует тип, объявленный как параметр, он является настраиваемым классом или параметризованным типом.

В приведенной далее строке тип *T* применяется для объявления объекта, названного *ob*.

```
T ob; // объявляет объект типа T
```

Как уже объяснялось, имя *T* — заменитель названия реального типа, который будет задан при создании объекта типа *Gen*. Таким образом, объект *ob* получит тип, переданный параметром *T* Например, если в параметре *T* передается тип *String*, у объекта *ob* будет тип *String*.

Теперь рассмотрим конструктор класса *Gen*.

```
Gen(T o)   {
  ob = o;
}
```

Обратите внимание на то, что *у* параметра o тип *T*. Это означает, что действительный тип переменной o определяется типом, передаваемым в параметрре *T* в момент создания объекта класса *Gen*. Кроме того, поскольку для параметра o и переменной-члена *ob* задан тип *T*, они получат один и тот же действительный тип в момент создания объекта класса *Gen*.

Тип *T*, объявленный как параметр, можно также использовать для задания типа данных, возвращаемых методом, как в случае метода *getob()*, приведенного в следующих строках.

```
T getob()   {
  return ob;
}
```

Поскольку у объекта *ob* тип *T*, его тип совместим с типом данных, возвращаемых методом *getob()*.

Метод *showType()* отображает на экране название типа *T* с помощью вызова метода *getName()* для объекта типа *class*, возвращенного методом *getClass()*, который вызывается для объекта *ob*. Метод *getClass()* определен в классе *Object* и таким образом является членом всех классов. Он возвращает объект типа C*lass*, который соответствует типу класса того объекта, для которого он был вызван. В типе *Class* определен метод *getName( )*, возвращающий строковое представление имени класса.

Класс *GenDemo* демонстрирует применение класса *Gen*. Сначала он создает вариант этого класса для целочисленных объектов, как показано в приведенной далее строке.

```
Gen<Integer> iOb;
```

Посмотрите внимательно на это объявление. Прежде всего, обратите внимание на то, что тип *Integer* задан в угловых скобках после названия класса *Gen*. В этом случае Integer — это *аргумент, определяющий тип*, или *аргумент типа* (type argument), который передается параметру *T* класса *Gen*. Это эффективный способ создания версии класса *Gen*, в которой все ссылки на тип *T* будут преобразованы в ссылки на тип *Integer*. Таким образом, в приведенном объявлении у объекта *ob* тип *Integer* и метод *getob{)* возвращает объект типа *Integer*.

Прежде чем продолжать, следует отметить, что компилятор языка Java на самом деле не создает разные версии класса *Gen* или других настраиваемых классов. Хотя это удобная абстрактная модель, она не соответствует тому, что происходит на самом деле. Вместо этого компилятор удаляет всю информацию о настраиваемом типе, применяя необходимые преобразования типов, для того чтобы заставить Вашу программу вести себя так, как будто в ней создается конкретная версия класса *Gen*. Таким образом, в Вашей программе действительно существует только одна версия класса *Gen*. Процесс удаления информации о настраиваемом типе называется *стиранием* или подчисткой (*erasure*) и мы вернемся к его обсуждению позже в этой главе.

В приведенной далее строке переменной *ob* присваивается ссылка на экземпляр версии Integer класса *Gen*.

```
iOb = new Gen<Integer>(88) ;
```

Обратите внимание на то, что при вызове конструктора класса *Gen* также задается *Integer* — аргумент типа. Это действие необходимо, так как у объекта

(В данном случае *iOb*), которому присваивается ссылка, тип *Gen<lnteger>*.
Следовательно, ссылка, возвращаемая операцией new, также должна указывать на объект *типа Gen<Integer>*. Если условие не соблюдается, возникнет ошибка во время компиляции, подобная приведенной далее.

```
iOb = new Gen<Double>(88.0); //Ошибка!
```

Поскольку у объекта *ob* тип *Gen<Integer>*, он не может использоваться для ссылки на объект типа *Gen<Double>*. Такая проверка соответствия типов — одно из главных преимуществ настраиваемых типов, обеспечивающее типовую безопасность (*type safety*).

Как сказано в комментариях к программе из листинга 3.1, в приведенном далее присваивании:

```
iOb = new Gen<Integer>(88);
```

применяется автоупаковка для инкапсуляции значения 88 базового типа int в тип *Integer*. Этот действие выполняется потому, что тип *Gen<Integer>* порождает конструктор, принимающий аргумент *Integer*. Поскольку ожидается тип *Integer*, Java автоматически инкапсулирует 88 в объект этого типа. Конечно, можно описать это действие в явном виде, как показано в следующей строке:

```
iOb = new Gen<Integer>(new Integer(88));
```

Однако Вы не получите никакого выигрыша от применения этого варианта.

Далее программа отображает на экране тип *Integer* — это тип переменной-члена *ob* из объекта *iOb*. Далее программа получает значение переменной *ob* с помощью следующей строки кода:

```
int v = iOb.getob();
```

У данных, возвращаемых методом *getob()* — тип *T*, который был заменен типом *Integer* при объявлении объекта *iOb*, следовательно, метод *getob( )* также возвращает объект типа *Integer*, который распаковывается в значение типа *int* при присваивании переменной *v* (базового типа *int*). Таким образом, нет необходимости приводить в соответствие тип, возвращаемый методом *getob()*, к типу *Integer*. Конечно, не нужно явно использовать распаковку. Приведенную строку можно записать в следующем виде:

```
int v = iOb.getob( ). intValue( );
```

но автораспаковка делает код короче.

Далее в классе GenDemo объявляется объект типа Gen<String>:

```
Gen<String> strOb = new Gen<String>("Generics Test");
```

Поскольку задан аргумент типа *String*, тип *String* замещает параметр типа *T* внутри класса *Gen*. Эта замена создает (логически) версию *String* класса *Gen*, что демонстрируют оставшиеся строки листинга 3.1.

Средства настройки типов работают только с объектами

При объявлении экземпляра настраиваемого типа аргумент типа, передаваемый типу, объявленному как параметр, должен быть каким-либо классом. Вы не можете использовать для этой цели базовый тип, такой как *int* или *char*. Например, в класс *Gen* можно передать любой класс как тип в параметре *T*, но нельзя заменить *T* ни одним базовым типом. Следовательно, приведенная далее строка кода недопустима:

```
Gen<int> strOb = new Gen<int>(53);  // Ошибка, нельзя использовать
                                    // базовый тип
```

Конечно, запрет на задание базового типа нельзя считать серьезным ограничением, т. к. Вы можете применять оболочки типов (как было показано в примерах *главы 2)* для инкапсуляции базового типа. Более того, механизм автоупаковки и автораспаковки языка Java делает применение оболочек типов очевидным.

Различия настраиваемых типов, основанных

на разных аргументах типа

Для понимания механизма настройки типов важно уяснить, что две ссылки на разные конкретные версии одного и того же настраиваемого типа не совместимы по типу. Например, если вставить приведенную далее строку кода в листинг 3.1, компиляция завершится с ошибкой:

```
iOb - strOb; // Неправильно!
```

Несмотря на то, что у переменных *iOb* и *strOb* тип *Gen<T>*, они ссылаются на объекты разных типов потому, что их аргументы типа отличаются. Это подход используется для усиления типовой защиты и предупреждения ошибок.

Как настраиваемые типы улучшают типовую безопасность

С этой точки зрения следовало бы спросить себя: "Можно ли реализовывать функциональную возможность, которой обладает класс *Gen*, без применения настраиваемых типов, простым заданием типа данных *Object* и выполнением необходимых операций приведения типов? Каков выигрыш от применения настраиваемого класса *Gen*?" Ответ — применение настраиваемых типов автоматически обеспечивает типовую безопасность всех операций с участием настраиваемого типа *Gen*. При этом они избавляют Вас от необходимости вручную вводить код для контроля и приведения типов.

Для того чтобы понять выгоды применения настраиваемых типов, прежде всего рассмотрим приведенную в листинге 3.2 программу, которая создает ненастраиваемый эквивалент класса *Gen*.

Листинг 3.2. Создание непараметризованного типа NonGen

```
// NonGen функционально эквивалентен классу Gen
// но не использует средства настройки типов.
class NonGen {
    Object ob;

    // Передает конструктору ссылку на
    // объект типа Object
    NonGen(Object o) {
      ob = o;
    }

    // Возвращает тип Object
    Object getob() {
      return ob;
    }
```

```
    // Показывает тип ob.
    void showType() {
      System.out.println("Type of ob is " +
                         ob.getClass().getName());
    }
}

// Демонстрирует ненастраиваемый класс
class NonGenDemo {
  public static void main(String args[]) {
    NonGen iOb;

    // Создает объект типа NonGen и запоминает
    // целое число в нем. Выполняется автоупаковка.
    iOb = new NonGen(88);


    // Показывает тип объекта iOb.
    iOb.showType();

    // Получает значение, хранящееся в iOb.
    // В этот момент требуется приведение типов.
    int v = (Integer) iOb.getob();
    System.out.println("value: " + v);


    System.out.println()

    // Создает другой объект типа NonGen и
    // запоминает в нем строку.
    NonGen strOb = new NonGen("Non-Generics Test");

    // Показывает тип данных объекта strOb.
    strOb.showType();


    // Получает значение, хранящееся в strOb.
    // Снова необходимо приведение типов.
    String str = (String) strOb.getob();
    System.out.println("value: " + str);

    // Эта строка компилируется, но концептуально неправильна!
    iOb = strOb;
    v = (Integer)iOb.getob(); // runtime error!
  }
}
```

В листинге 3.2 есть несколько интересных фрагментов. Прежде всего, в классе *NonGen* все вхождения параметра *T* заменены типом *Object*. Это замещение позволяет классу *NonGen* хранить объекты любого типа, как и в случае настраиваемого типа. Но при этом у компилятора нет никаких реальных сведений о том, какой конкретный тип данных содержится в *NonGen*. Подобная ситуация плоха по двум причинам. Во-первых, для извлечения хранящихся данных должны выполняться явные приведения типов. Во-вторых, разнообразные ошибки несовместимости типов невозможно обнаружить до запуска программы. Обсудим каждую проблему подробно.

Обратите внимание на приведенную далее строку: *int v = (Integer)iOb.getob();*
Поскольку метод *getob( )* возвращает тип *Object*, понадобится преобразование в тип *Integer* для автораспаковки значения и сохранения его в переменной *v*. Если Вы удалите операцию приведения типа, программа не будет компилироваться. При использовании настраиваемого класса это действие скрыто от Вас. В случае применения не настраиваемого типа, приведение типа следует выполнять явно. Это не только неудобство, но и является потенциальным источником ошибок.

Теперь рассмотрим следующий фрагмент из заключительной части листинга 3.2.

```
// Эта строка компилируется, но концептуально неправильна!
Ob = StrOb;
v = (Integer)iOb.getob() ; // ошибка во время выполнения!
```

Здесь переменной *iOb* присваивается переменная *strOb*. Однако *strOb* ссылается на объект, содержащий строку, а не целое число. Синтаксически такое присваивание корректно, поскольку ссылки типа *NonGen* одинаковы, и любая ссылка типа *NonGen* может быть перенаправлена на другой объект этого типа. Тем не менее описываемый оператор семантически неверен, что подтверждается следующей строкой. В ней тип данных, возвращаемых методом *getob()* приводится к типу Integer и затем делается попытка присвоить это значение переменной *v*. Беда в том, что объект *iOb* уже ссылается на объект, содержащий тип *String*, не *Integer*. К сожалению, без использования настраиваемых типов невозможно известить об этом компилятор. Вместо этого возникает ошибка времени выполнения (*runtime error*) при попытке приведения к типу *Integer*. Как Вам известно, наличие в Вашей программе исключений, генерируемых во время выполнения, крайне нежелательно.

Описанная ситуация никогда не возникнет при применении настраиваемых типов. Если такой фрагмент кода встретится в версии программы, использующей настраиваемые типы, компилятор обнаружит ошибку и сообщит о ней, предупреждая тем самым серьезный сбой, возникающий при генерации исключения во время выполнения. Возможность разработки кода, обеспечивающего типовую безопасность и позволяющего обнаруживать ошибки несоответствия типов на этапе компиляции, — важнейшее преимущество использования настраиваемых типов. Несмотря на то, что с помощью ссылок типа *Object* всегда можно создать "настраиваемый" код, в нем не будет должной защиты типов и их неправильное использование может привести к возникновению исключительных ситуаций во время выполнения программы. Применение настраиваемых типов препятствует их возникновению. По существу, благодаря применению настраиваемых типов ошибки времени выполнения можно превратить в ошибки этапа компиляции. А это важное преимущество.

Настраиваемый класс

с двумя параметрами типа

Вы можете объявить несколько параметров типа в настраиваемом типе данных. Для определения двух или нескольких параметров типа используйте список, разделенный запятыми. Например, приведенный в листинге 3.3 класс *TwoGen* представляет собой вариант класса *Gen* с двумя параметрами типа.

Листинг 3.3. Пример простого настраиваемого класса с двумя параметрами

```
class TwoGen<T, V> {
  T ob1;
  V ob2;

  // Передает конструктору ссылку на
  // объекты типов T и V.
  TwoGen(T o1, V o2) {
    ob1 = o1;
    ob2 = o2;
  }
```

```
  // Показывает типы T и V.
  void showTypes() {
    System.out.println("Type of T is " +
                       ob1.getClass().getName());

    System.out.println("Type of V is " +
                       ob2.getClass().getName());
  }

  T getob1() {
    return ob1;
  }

  V getob2() {
    return ob2;
  }
}



// Демонстрирует использование класса TwoGen.
class SimpGen {
  public static void main(String args[]) {

    TwoGen<Integer, String> tgObj =
      new TwoGen<Integer, String>(88, "Generics");

    // Show the types.
    tgObj.showTypes();

    // Obtain and show values.
    int v = tgObj.getob1();
    System.out.println("value: " + v);

    String str = tgObj.getob2();
    System.out.println("value: " + str);
  }
}
```

Далее приведен вывод результата работы программы из листинга 3.3.

```
Type of  T is Java.lang.Integer
Type of  V is  Java.lang.String
value: 88
value: Generics
```

Обратите внимание на объявление класса *TwoGen*:

```
class TwoGen<T, v> {
```

В нем задаются два параметра: *T* и *v*, разделенные запятыми. Поскольку у класса два параметра типа, при создании объекта следует передать в типе *TwoGen* два аргумента типа, как показано в следующих строках: *TwoGen<Integer, String> tgObj = new TwoGen<:Integer, String>(88, "Generics");*

В данном случае тип *T* заменяется типом *Integer*, а тип *v* — типом *String*. Несмотря на то, что в приведенном примере аргументы типа разные, можно применять и одинаковые аргументы. Приведенный в следующей строке код полностью корректен:

```
TwoGen< String, String> x = new TwoGen< String, String>("A", "B");
```

В этом случае и *T*, и *V* заменяются типом *String*. Правда, если аргументы типа всегда одинаковы, нет необходимости использовать два параметра типа.

Общий вид объявления настраиваемого класса

Можно обобщить синтаксические конструкции, использованные в приведенных примерах. Ниже приведена синтаксическая запись для объявления настраиваемого класса.

```
class  имя-класса<список-парамегров-типа>    {//...
```

Далее приведена синтаксическая запись для объявления ссылки на настраиваемый класс.

```
class  имя-класса<список-аргумеитов-типа> имя-переменной =
     new имя-класса<список-аргументов-типа>(список-констант-аргументов);
```

Ограниченные типы

В предыдущих примерах параметры типа могли заменяться классами любого типа. В большинстве случаев это замечательно, но иногда полезно ограничить количество типов, которые можно передавать параметру типа. Например, Вы хотите создать настраиваемый класс, который содержит метод, возвращающий среднее арифметическое элементов числового массива. Более того, хотите использовать этот класс для получения среднего арифметического элементов массивов разных числовых типов, включая целые, числа с плавающей запятой и двойной точности. Следовательно, следует задать тип элементов массива в общем виде с помощью параметра типа. Для создания такого класса можно попытаться написать код, похожий на приведенный в листинге 3.4.

Листинг 3.4. Неудачная попытка создать настраиваемый класс дли вычисления среднего арифметического элементов массива любого заданного числового типа

```
// Класс содержит ошибку!
class Stats<T> {
  T[] nums;

  // передает конструктору ссылку на
  // массив типа T.
  Stats(T[] o) {
    nums = o;
  }

  //  Возвращает тип double во всех случаях,
  double average() {
    double sum = 0.0;

    for(int i=0; i < nums.length; i++)
      sum += nums[i].doubleValue(); // Error!!!

    return sum / nums.length;
  }
}
```

В классе *Stats* метод *average()* пытается получить все элементы массива *num*, приведенные к типу *double* с помощью метода *doubleValue ()*. Поскольку все числовые классы, такие как *Integer* и *Double*, являются подклассами класса *Number*, а в классе *Number*

определен метод *doubleValue()*, этот метод доступен для всех числовых классов-оболочек. Но проблема состоит в том, что компилятор ничего не знает о Вашем намерении создавать объекты типа *Stats*, используя только числовые типы для замены параметра типа *m*. Более того, Вам нужно каким-либо способом обеспечить действительную передачу только числовых типов. Для обработки подобных ситуаций язык Java предлагает *ограниченные типы* (*bounded types*). При объявлении параметра типа Вы можете задать верхнюю границу, определяющую суперкласс, от которого должны быть унаследованы все аргументы типа. Такое ограничение устанавливается с помощью ключевого слова extends при описании параметра типа, как показано в следующей строке:

```
<T extends superclass>
```

Приведенное объявление указывает на то, что параметр *T* можно заменить только типом *superclass* или его подклассами (производными от него классами). Таким образом, *superclass* задает верхнюю границу включительно.

Вы можете использовать суперкласс *Number* как верхнюю границу для настройки класса *Stats*, описанного ранее (листинг 3.5).

Листинг 3.5. Использование ограниченного типа при объявлении класса Stats

```
// В этой версии класса Stats, аргумент типа для
// T должен быть Number, или производный
// от Number класс.
class Stats<T extends Number> {
    T[] nums; // массив типа Number или его подкласса

  // Передает конструктору ссылку на
  // массив типа Number или его подкласса.
  Stats(T[] o) {
    nums = o;
  }


  // Возвращает тип double в любом случае.
  double average() {
    double sum = 0.0;

    for(int i=0; i < nums.length; i++)
      sum += nums[i].doubleValue();

    return sum / nums.length;
  }
}



// Демонстрирует применение класса Stats.
class BoundsDemo {
  public static void main(String args[]) {

    Integer inums[] = { 1, 2, 3, 4, 5 };
    Stats<Integer> iob = new Stats<Integer>(inums);
    double v = iob.average();
    System.out.println("iob average is " + 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("dob average is " + w);

    // Эти строки не будут компилироваться,так как String
    // не является подклассом суперкласса Number.
    // String strs[]   =   { "1", "2", "3", "4", "5"  };
    // Stats<String> strob = new Stats<String>(strs);
    // double x = strob.average() ;
    // System.out.printlnf"strob average is  "  + v);
  }
}
```

Далее приведен вывод результатов работы программы из листинга 3.5.

```
Average is 3.0
Average is 3.3
```

Обратите внимание на новое объявление класса Stats, приведенное в следующей строке:

```
class Stats<T extends Number>  {
```

Поскольку теперь тип *m* ограничен суперклассом *Number*, компилятор языка Java знает, что все объекты типа *m* могут вызывать метод *doubleValue()*, определенный в суперклассе *Number*. Это само по себе значительное преимущество. Но кроме этого, ограничение типа *m* препятствует созданию нечисловых объектов типа *Stats*. Если удалить символы комментария из заключительных строк листинга 3.5, а затем выполнить компиляцию, Вы получите ошибки на этапе компиляции, так как тип *String* не является подклассом суперкласса *Number*.

Применение метасимвольных аргументов

Как ни полезна безопасность типов, иногда она может мешать формированию вполне приемлемых конструкций. Предположим, что в имеющийся класс *Stats*, описанный в предыдущем разделе, Вы хотите добавить метод *sameAvg()* который определяет, содержатся ли в двух объектах *Stats* массивы с одинаковым значением среднего арифметического, независимо от типа числовых данных массивов. Например, если один объект содержит значения 1.0, 2.0 и 3.0 типа *double*, а второй целые числа 1, 2 и 3, средние арифметические массивов будут одинаковы. Один из способов реализации метода *sameAvg( )* — передача в *класс Stats* аргумента, последующее сравнение среднего арифметического этого аргумента со средним арифметическим объекта, вызвавшего метод, и возврат значения *true*, если средние арифметические одинаковы. Например, можно попытаться вызвать метод *sameAvg()*, как показано в следующем фрагменте кода:

```
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("Averages are the same.");
else
   System.out.println("Averages differ.");
```

Поскольку *Stats* — настраиваемый класс, его метод *sameAvg()* может обрабатывать любой объект типа *Stats* и кажется, что создать этот метод просто. К сожалению, возникнут проблемы, как только Вы попытаетесь объявить параметр типа для класса *Stats*. Класс *Stats* — это параметризованный тип и неясно, какой же тип объявлять для параметра типа класса *Stats* в списке параметров метода.

Вам может показаться, что решение выглядит так, как показано в следующих строках кода, использующих *T* как параметр типа.

```
// Этот пример не будет работать!
// Определяет, равны ли средние арифметические.
boolean sameAvg(Stats<T> ob) {
  if ((average) == ob.average())
    return true;
  return false;
}
```

К сожалению, приведенный пример будет обрабатывать только те объекты класса *Stats*, у которых тип такой же, как у объекта, вызвавшего метод. Например, если метод вызывает объект типа *Stats<Integer>*, параметр *ob* должен тоже быть типа *Stats<Integer>*. Такой метод нельзя использовать для сравнения среднего арифметического объекта *Stats<Double>* со средним арифметическим объекта типа *Stats<Short>*. Следовательно, предложенный подход не будет работать, за исключением нескольких ситуаций, и не даст общего (т. е. универсального) решения.

Для создания универсального метода *sameAvg()* Вы должны использовать другую функциональную возможность средств настройки типов — *мета символьный аргумент*, или символьную маску (*wildcard argument*). Метасимвольный аргумент задается знаком ? и представляет неизвестный тип. Используя такую маску, можно описать метод *sameAvg( )* так, как показано в следующем фрагменте кода:

```
// Определяет, равны ли средние арифметические.
// Обратите внимание на применение метасимвола.
boolean sameAvg(Stats<?> ob) {
  if ((average) == ob.average())
    return true;
  return false;
}
```

В приведенном примере тип *Stats<?>* соответствует любому объекту типа *Stats* и позволяет сравнивать средние арифметические двух объектов типа *Stats*, как показано в листинге 3.6.

Листинг 3.6. Применение метасимвола, или символьной маски

```
class Stats<T extends Number> {
  T[] nums;
  // массив типа Number или его подкласса
  // Передает конструктору ссылку на
  // массив типа Number или его подкласса.
  Stats(T[] o) {
    nums = o;
  }
  // Всегда возвращает тип double.
  double average() {
    double sum = 0.0;

    for(int i=0; i < nums.length; i++)
      sum += nums[i].doubleValue();

    return sum / nums.length;

  }
  // Определяет, равны ли два средних арифметических.
  // Обратите внимание на использование метасимвола (или маски).
  boolean sameAvg(Stats<?> ob) {
    if(average() == ob.average())
      return true;
```

```
    return false;
  }
}

// Демонстрирует применение метасимвола.
class WildcardDemo {
  public static void main(String args[]) {
    Integer inums[] = { 1, 2, 3, 4, 5 };
    Stats<Integer> iob = new Stats<Integer>(inums);
    double v = iob.average();
    System.out.println("iob average is " + 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("dob average is " + 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("fob average is " + x);

    // Проверяет, у каких массивов одинаковые средние арифметические.
    System.out.print("Averages of iob and dob ");
    if(iob.sameAvg(dob))
      System.out.println("are the same.");
    else
      System.out.println("differ.");

    System.out.print("Averages of iob and dob ");
    if(iob.sameAvg(fob))
      System.out.println("are the same.");
    else
      System.out.println("differ.");
  }
}
```

Далее приведен вывод программы из листинга 3.6:

```
iob average is 3.0
dob average is 3.3
fob average is 3.0
Averages of iob and dob differ.
Averages of iob and fob are the same.
```

Последнее замечание: важно понять, что метасимвол не влияет на тип создаваемого объекта класса *Stats*. Тип определяется ключевым словом extends в объявлении класса *Stats*. Метасимвол, или маска, обеспечивает совместимость любых допустимых объектов типа *Stats*.

Ограниченные метасимвольные аргументы

Метасимвольные аргументы могут ограничиваться почти так же, как параметры типа. Ограничение метасимвольных аргументов особенно важно, когда Вы создаете настраиваемый

тип, который будет оперировать иерархией классов. Для того чтобы лучше понять это, обсудим пример, приведенный в листинге 3.7. Рассмотрим следующую иерархию классов, инкапсулирующих координаты.

```
// Двухмерные координаты.
class TwoD {
  int x, y;

  TwoD(int a, int b) {
    x = a;
    y = b;
  }
}



// Трехмерные координаты.
class ThreeD extends TwoD {
  int Z;

  ThreeDtint 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 класс *TwoD*, в котором хранятся двухмерные координаты *XY*. Класс *TwoD* наследуется классом *ThreeD*, добавляющим третье измерение и описывающим координаты *XYZ*. Класс *ThreeD* наследуется классом *FourD*, включающим четвертое измерение (время) и создающим четырехмерные координаты.

Далее приведено описание класса *Coords*, содержащего массив координат.

```
// Этот класс поддерживает массив объектов-координат.
class Coords<T extends TwoD> {
  T[] coords;

  Coords(T[] o) { coords = o; }
}
```

Обратите внимание, в классе *Coords* задастся параметр типа ограниченный классом *TWOD*. Это означает, что любой массив, хранящийся в объекте типа *Coords*, будет содержать объекты класса *TWOD* ИЛИ ОДНОГО ИЗ его производных классов.

Теперь предположим, что Вы хотите написать метод, отображающий на экране координаты *X* и *Y*, хранящиеся в каждом элементе массива *Coords* из объекта типа *Coords*. Поскольку все типы объектов класса *Coords* содержат 2 координаты (*X* и *Y*), это легко сделать с помощью метасимвола, как показано в следующем фрагменте кода:

```
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() ;
}
```

Класс *Coords* — ограниченный настраиваемый тип, задающий класс *TwoD* как верхнюю границу настраиваемого типа, следовательно, все объекты, которые могут быть использованы для создания объекта типа *Coords*, будут массивами типа *TwoD* или классами, производными от *TwoD*. Таким образом, метод *showXY( )* может отобразить на экране содержимое любого объекта типа *Coords*.

А если Вы захотите создать метод, который выводит на экран координаты *X*, *Y* и *Z* объекта класса *ThreeD* или объекта типа *FourD*? Сложность состоит в том, что не у всех объектов класса *Coords* есть три координаты, у объекта класса *TwoD* их всего две (*X* и *Y*). Как же написать метод, который отображает на экране координаты *X*,*Y* и *Z* для объектов типов *Coords*<*ThreeD*> и *Coords*<*FourD*> и одновременно препятствовать его использованию с объектами класса *TwoD*? Выход — применить *ограниченный метасимвольный аргумент*.

Ограниченный метасимвол, или маска, задает как верхнюю, так и нижнюю границы аргумента типа. Это позволяет ограничить набор типов объектов, которые будут обрабатываться методом. Чаще всего ограниченный метасимвольный аргумент используется для указания верхней границы с помощью ключевого слова *extends* практически так же, как при создании ограниченного типа.

Применяя ограниченный метасимвольный аргумент, легко разработать метод, который отображает на экране координаты *X*, *Y* и *Z* из объекта типа *Coords*, если они действительно содержатся в этом объекте. Например, в приведенном далее фрагменте метод *showXYZ( )* отображает координаты *X*, *Y* и *Z* элементов массива, хранящихся в объекте типа *Coords*, если эти элементы имеют тип *ThreeD* (или производный от типа *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 на то, что ключевое слово *extends* добавлено к метасимволу в объявлении параметра *c*. Оно констатирует, что маска ? соответствует типу *ThreeD* и любому производному от него классу. Таким образом, ключевое слово *extends* устанавливает верхнюю границу, которой может соответствовать метасимвол ?. Благодаря этому ограничению метод *showXYZ()* может вызываться со ссылками на объекты типа *Coords*<*ThreeD*> или *Coords*<*FourD*>, но никак не на объекты типа *Coords* <*TwoD*>. Попытка вызвать метод *showXYZ ()* со ссылкой на объект *Coords* <*TwoD*> приведет к ошибке во время компиляции, обеспечивая таким образом типовую безопасность.

В листинге 3.7 полностью приведена программа, демонстрирующая применение ограниченного метасимвольного аргумента.

Листинг 3.7. Применение ограниченных метасимвольных аргументов

```
// Двухмерные координаты.
class TwoD {
  int x, y;
```

```
  TwoD(int a, int b) {
    x = a;
    y = b;
  }
}

// Трехмерные координаты.
class ThreeD extends TwoD {
  int z;

  ThreeD(int a, int b, int c) {
    super(a, b);
    z = c;
  }
}


// Четырехмерные координаты.
class FourD extends ThreeD {
  int t;

  FourD(int a, int b, int c, int d) {
    super(a, b, c);
    t = d;
  }
}


// Этот класс поддерживает массив объектов, содержащих координаты.
class Coords<T extends TwoD> {
  T[] coords;

  Coords(T[] o) { coords = o; }
}


// Демонстрирует применение ограниченной символьной маски.
class BoundedWildcard {
  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();
  }

  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();
  }

  static void showAll(Coords<? extends FourD> c) {
    System.out.println("X Y Z T 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 + " " +
                         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("Contents of tdlocs.");
    showXY(tdlocs); // OK, is a TwoD.coords.length;
 // showXYZ(tdlocs); // Ошибка, так как тип не ThreeD
 // showAll(tdlocs); // Ошибка, так как тип не FourD

    // Теперь создает объекты типа FourD.
    FourD fd[] = {
      new FourD(1, 2, 3, 4),
      new FourD(6, 8, 14, 8),
      new FourD(22, 9, 4, 9),
      new FourD(3, -2, -23, 17)
    };

    Coords<FourD> fdlocs = new Coords<FourD>(fd);

    System.out.println("Contents of fdlocs.");
    // These are all OK.
    showXY(fdlocs);
    showXYZ(fdlocs);
    showAll(fdlocs);
  }
}
```

Далее приведен вывод результатов работы программы из листинга 3.7:

```
Contents of tdlocs.
X Y Coordinates:
0 0
7 9
18 4
```

```
-1 -23

Contens of fdlock.
X Y Coordinates:
1 2
6 8
22 9
3 -2

X Y Z Coordinates:
1 2 3
6 8 14
22 9 4
3 -2 -23

X Y Z T Coordinates:
1 2 3 4
6 8 14 8
22 9 4 9
3 -2 -23 17
```

Обратите внимание на приведенные далее строки, помеченные как комментарии:

```
// showXYZ(tdlocs); // Ошибка, так как тип не ThreeD
// showAll(tdlocs); // Ошибка, так как тип не FourD
```

Поскольку *tdlocs* — объект типа *TwoD*, он не может использоваться при вызове методов *showXYZ ( )* или *showAll( )*, потому что этому препятствуют ограниченные метасимвольные аргументы, заданные при объявлении методов. Для того чтобы убедиться в справедливости этого замечания, удалите знаки комментария из этих строк и откомпилируйте программу. Вы получите ошибки компиляции из-за несоответствия типов.

Как правило, для задания верхней границы метасимвольного аргумента применяют следующую синтаксическую запись:

<? extends superclass>

в которой *superclass* обозначает имя класса, который служит верхней границей. Помните, что указанная верхняя граница (обозначенная как *superclass)* включена в область допустимых типов.

Вы можете также задать нижнюю границу для метасимвола с помощью ключевого слова *super* в объявлении метасимвольного аргумента. Далее приведена соответствующая синтаксическая запись:

<? super subclass>

В этом случае, допустимыми аргументами считаются только суперклассы класса, заданного как *subclass.* При этом *subclass* не является допустимым типом аргумента.

Создание настраиваемого метода

Как показано в предыдущих примерах, методы в настраиваемых классах могут использовать параметр типа и таким образом автоматически становятся настраиваемыми по отношению к этому параметру типа. Однако можно и объявить *настраиваемый метод* (*generic method*) с одним или несколькими собственными параметрами типа. Более того, есть возможность создать настраиваемый метол внутри ненастраиваемого класса.

Начнем с примера. В программе из листинга 3.8 объявляется ненастраивае-мый класс, названный *GenMethDemo,* и внутри класса статический настраиваемый метод с именем *isIn ().* Метод *isIn ( )* определяет, является ли объект элементом массива. Он может использоваться с

объектом любого типа и любым массивом, при условии, что массив содержит объекты, тип которых сопоставим с типом проверяемого объекта.

Листинг 3.8. Демонстрация простого настраиваемого метода

```
class GenMethDemo {

  // Определяет, является ли объект элементом массива.
  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[]) {

    // Использует метод isin() для объектов типа Integer.
    Integer nums[] = { 1, 2, 3, 4, 5 };

    if(isIn(2, nums))
      System.out.println("2 is in nums");

    if(!isIn(7, nums))
      System.out.println("7 is not in nums");

    System.out.println();
    // Использует метод isIn() для объектов типа String.
    String strs[] = { "one", "two", "three",
                      "four", "five" };

    if(isIn("two", strs))
      System.out.println("two is in strs");

    if(!isIn("seven", strs))
      System.out.println("seven is not in strs");

    // Opps! Не откомпилирует, поскольку типы не совместимы.
    //   if(isln("two", nums})
    //   System.out.println("two is in strs”);
  }

}
```

Далее приведен вывод результатов работы программы из листинга 3.8:

```
2 is in nums
7 is not in nums

two is in strs
seven is not in strs
```

Рассмотрим подробно метод *isIn()*. Во-первых, обратите внимание на его объявление, приведенное в следующей строке:

```
static <T, V extends T> boolean isIn(T x, V[] y) {
```

Параметры типа указаны *перед* типом значения, возвращаемого методом. Во-вторых, обратите внимание на то, что параметр типа *v* ограничен сверху параметром типа *m*. Следовательно, тип параметра *v* должен быть таким же, как у параметра *T* или классом, производным от *T*. Описанная взаимосвязь параметров обеспечивает вызов метода *isIn( )* с аргументами, сопоставимыми друг с другом. Отметьте также то, что метод *isIn( )* статический, т. е. может вызываться независимо от какого-либо объекта. Однако настраиваемые методы могут быть как статическими, так и нестатическими, на этот счет нет никаких ограничений.

Теперь рассмотрим как метод *isIn ()* вызывается в методе *main ( )* с использованием обычного синтаксиса вызова без необходимости указания аргументов типа. Это возможно благодаря тому, что типы аргументов распознаются автоматически и типы *T* и *V* настраиваются соответственно. Например, в первом вызове:

```
if(isIn(2, nums))
```

тип первого аргумента — *Integer* (в результате автоупаковки), что вызывает замену типа *T* классом *Integer*, Базовый тип второго аргумента тоже *Integer*, что в свою очередь ведет к замене параметра *V* классом *Integer*.

Во втором вызове используется тип *String*, и параметры *T* и *V* заменяются классом *String*.

Теперь рассмотрим строки комментария приведенные далее:

```
//   if(isIn("two", nums))
//     System.out.println("two is in strs");
```

Если Вы удалите знаки комментария и попробуете откомпилировать программу, то получите сообщение об ошибке. Причина заключается в том, что параметр типа *v* ограничен типом *T* с помощью ключевого слова *extends* при объявлении параметра *V*. Это означает, что тип параметра *V* должен быть таким же, как тип параметра *T* или классом, производным от типа *T*. В нашем случае у первого аргумента тип — *String*, заменяющий *T* классом *String*, а у второго аргумента тип *Integer* который не является подклассом типа *String*. Подобная ситуация приводит к появлению ошибки несовместимости типов на этапе компиляции. Описанная способность обеспечения типовой безопасности — одно из важнейших преимуществ настраиваемых методов.

Далее приведена синтаксическая запись для настраиваемого метода:

```
<type-param-list> ret-type meth-name(param-list) {  //...
```

type-param-list всегда представляет собой разделенный запятыми список параметров типа. Обратите внимание на то, что у настраиваемых методов этот список предшествует типу значения, возвращаемого методом.

Настраиваемые конструкторы

Конструкторы также могут быть настраиваемыми, даже если их класс не является настраиваемым типом. Рассмотрим короткую программу, приведенную в листинге 3.9.

Листинг 3.9. Применение настраиваемого 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();
  }
}
```

Далее приведен вывод программы из листинга 3.9:

```
val:  100.0
val:  123.5
```

Поскольку конструктор *GenCons( )* задает параметр настраиваемого типа, который должен быть производным классом от класса *Number*, его можно вызвать с любым числовым типом, включая *Integer*, *Float* или *Double.*
Следовательно, хотя класс *GenCons* не является настраиваемым типом, его конструктор настраиваемый.

Настраиваемые интерфейсы

Кроме настраиваемых классов и методов Вы можете создавать *настраиваемые интерфейсы* (*generic interface*). Они задаются так же, как настраиваемые классы. В листинге 3.10 приведен пример настраиваемого интерфейса. В нем создается интерфейс, названный *MinMax*, объявляющий методы *min()* и *max()*, которые должны возвращать минимальное и максимальное значения некоторого множества объектов.

Листинг 3.10. Пример настраиваемого интерфейса

```
// Интерфейс Min/Max.
interface MinMax<T extends Comparable<T>> {
  T min();
  T max();
}


// Теперь реализуем 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("Max value in inums: " + iob.max());
    System.out.println("Min value in inums: " + iob.min());

    System.out.println("Max value in chs: " + cob.max());
    System.out.println("Min value in chs: " + cob.min());
  }
}
```

Далее приведен вывод результатов работы программы из листинга 3.10:

```
Max value in inums: 8
Min value in inums: 2
Max value in chs: w
Min value in chs: b
```

Несмотря на то, что большая часть кода листинга 3.10 понятна, следует сделать несколько замечаний. Во-первых, обратите внимание на объявление интерфейса *MinMax*, приведенное в следующей строке:

```
interface MinMax<T extends Comparable<T>> {
```

Вообще настраиваемый интерфейс объявляется так же, как настраиваемый класс. В нашем случае параметр типа — *T*, и он должен расширять тип *comparable*. Обратите внимание на то, что тип *comparable* — тоже настраиваемый тип. Он принимает параметр типа, который задает тип сравниваемых объектов.

Далее класс *MyClass* реализует интерфейс *MinMax*. Рассмотрим объявление класса *MyClass*, приведенное в следующей строке:

```
class MyClass<T extends Comparable<T>> implements MinMax<T> {
```

Уделите особое внимание способу, которым параметр типа т объявляется в классе *MyClass* и затем передается в интерфейс *MinMax*. Поскольку интерфейсу *MinMax* требуется тип, расширяющий тип *comparable*, в классе, реализующем интерфейс (в нашем случае *MyClass*), должна быть задана та же самая граница. Более того, как только эта граница установлена, нет необходимости задавать ее снова в той части объявления класса, которая начинается с

ключевого слова *implements*, На самом деле подобное действие было бы ошибкой. Например, приведенная далее часть кода некорректна и не будет компилироваться.

```
// Это неправильно!
class MyClass<T extends Comparable<T>>
      implements MinMax<T extends Comparable<T>> {
```

Уже заданный параметр типа просто передается в интерфейс без дальнейшей модификации.

Вообще, если класс реализует настраиваемый интерфейс, этот класс также должен быть настраиваемым, по крайней мере, до той степени, которая обеспечивает получение параметра типа и передачу его в интерфейс. Например, приведенная далее строка кода, в которой делается попытка объявить класс *MyClass*, ошибочна:

```
class MyClass implements MinMax<T> { //Ошибка!
```

Поскольку в классе *MyClass* не объявлен параметр типа, не существует способа передачи его в интерфейс *MinMax*. В данном случае идентификатор *T* просто неизвестен, и компилятор сообщит об этой ошибке. Конечно, если класс реализует *конкретную версию* настраиваемого интерфейса, такую как приведенная в следующей строке:

```
class MyClass  implements MinMax<Integer>   {   // OK
```

реализующему классу нет необходимости быть настраиваемым.

Применение настраиваемого интерфейса обладает двумя преимуществами. Во-первых, интерфейс можно реализовать для данных разных типов. Во-вторых, у вас появляется возможность наложить ограничения (т. е. установить границы) на типы данных, для которых может быть реализован интерфейс. В случае интерфейса *MinMax*, например, только типы, реализующие интерфейс *comparable*, могут передаваться для замены параметра *T*.

Далее приведена обобщенная синтаксическая конструкция для описания настраиваемого интерфейса:

```
interface interface-name<type-param-list>  {//...
```

В данной записи *type-param-list* — это разделенный запятыми список параметров типа. Когда настраиваемый интерфейс реализуется, Вы должны заменить его списком аргументов типа, как показано в следующей строке:

```
class class-name<type-param-list>
     implements interface-name<type-param-list> {//...
```

Типы raw и разработанный ранее код

Поскольку настраиваемые типы — это новое средство программирования, языку Java необходимо обеспечить совместимость с уже действующим кодом, разработанным до появления настраиваемых типов. Помните, что существуют миллионы и миллионы строк возникшего до версии 5.0 кода, который должен продолжать выполняться и быть совместимым с настраиваемыми типами. Созданный ранее код должен уметь работать с настраиваемыми типами, а настраиваемый код должен выполняться совместно с уже действующим кодом.

Для обеспечения перехода к настраиваемым типам язык Java разрешает использовать настраиваемый класс без аргументов типа. В этом случае создается *несформированныn тип данных* (*raw type*) для класса. Этот тип данных совместим с разработанным ранее кодом, ничего не знающим о настраиваемых типах. Основной недостаток применения несформированного типа — потеря типовой безопасности, обеспечиваемой настраиваемыми типами.

В листинге 3.11 приведен пример использования несформированного (*raw*) типа.

Листинг 3.11. Демонстрация применения несформированного типа

```
class Gen<T> {

  T ob;
```

```
  // передает конструктору ссылку на
  // объект типа T.
  Gen(T o) {
    ob = o;
  }

  //Возвращает ob.
  T getob() {
    return ob;
  }
}

// Демонстрирует несформированный (raw) тип,
class RawDemo {
  public static void main(String args[]) {

    // Create a Gen Object for Integers.
    Gen<Integer> iOb = new Gen<Integer>(88);

    // Create a Gen Object for Strings.
    Gen<String> strOb = new Gen<String>("Generics Test");

    // Create a raw-type Gen Object and give it
    // a Double value.
    Gen raw = new Gen(new Double(98.6));

    // Cast here is necessary because type is unknown.
    double d = (Double) raw.getob();
    System.out.println("value: " + d);

    // The use of a raw type can lead to runtime.
    // exceptions.  Here are some examples.

    // The following cast causes a runtime error!
    // int i = (Integer) raw.getob(); // runtime error

    // This assigment overrides type safety.
    strOb = raw; // OK, but potentially wrong
    // String str = strOb.getob(); // runtime error

    // This assignment also overrides type safety.
    raw = iOb; // OK, but potentially wrong
    // d = (Double) raw.getob(); // runtime error
  }
}
```

В листинге 3.11 есть несколько интересных моментов. Во-первых, обратите внимание на создание объекта несформированиого (*raw*) типа настраиваемого класса *Gen*, приведенное в следующей строке:

```
Gen raw = new Gen(new Double(98.6));
```

При этом не задано никаких аргументов типа. По существу, формируется объект класса *Gen*, у которого параметр типа *T* заменен типом *Object*.

Несформированный тип лишен типовой безопасности. Следовательно, переменной этого типа можно присвоить ссылку на любой объект класса *Gen*. Обратное также справедливо: переменной конкретного типа класса *Gen* может быть присвоена ссылка на объект несформированного (*raw*) типа класса *Gen*. Однако обе операции небезопасны, так как контроль типов механизма настройки типов обойден.

Отсутствие типовой безопасности иллюстрируется помеченными как комментарий заключительными строками листинга 3.11 Давайте рассмотрим каждую ситуацию. Начнем с приведенной в следующей строке:

```
int  i = (Integer) raw.getob();   //  ошибка времени выполнения
```

В этом операторе извлекается значение объекта *ob*, содержащегося в объекте *raw*, и преобразуется в тип *Integer*. Проблема заключается в том, что объект *raw* содержит величину типа *Double*, а не целое число. Но это несоответствие не определяется во время компиляции, поскольку неизвестен тип объекта *raw*. Таким образом, этот оператор приведет к ошибке во время выполнения.

В следующем фрагменте объекту *strOb* (ссылка на тип *Gen<String>*) присваивается ссылка на объект не сформированного типа класса *Gen*:

```
strOb = raw; // OK, но потенциально неверно
//  String str = strOb.getob{); // ошибка времени выполнения
```

Приведенное присваивание синтаксически корректно, но сомнительно. Поскольку у объекта *strOb* тип *Gen<String>*, предполагается, что он содержит данные типа *String*. Но после присваивания объект, на который ссылается *strOb*, содержит данные типа *Double*. Следовательно, во время выполнения, когда делается попытка присвоить содержимое объекта *strOb* переменной *str*, возникает ошибка, так как в этот момент в объекте *strOb* хранится значение несформированного типа. Таким образом, присваивание ссылки на объект несформированного типа ссылке на объект настраиваемого типа обходит средства типовой безопасности.

В следующем фрагменте представлена обратная ситуация:

```
raw = iOb; // OK, но потенциально неверно
//   d = (Double) raw.getob(); // ошибка времени выполнения
```

В этом случае ссылка на объект настраиваемого типа присваивается ссылке на переменную несформированного (*raw*) типа. Несмотря на то, что это присваивание синтаксически корректно, оно приведет к проблеме, которая показана во второй строке фрагмента. В ней объект raw ссылается на объект, содержащий объект класса *Integer*, а механизм приведения типов предполагает, что в нем хранится объект класса *Double*. Эта ошибка не может быть обнаружена и исправлена на этапе компиляции, следовательно, она проявится во время выполнения программы.

Поскольку существует потенциальная опасность, кроящаяся в данных не-сформированного (*raw*) типа, компилятор java выводит на экран *предупреждения об отсутствии контроля типов* (*unchecked warning*), когда применение несформированного типа может нарушить типовую безопасность. Приведенные далее строки из листинга 3.11 порождают такие предупреждения:

```
Gen raw = new Gen(new Double(98.6)) ;

strOb *= raw; // OK, но потенциально неверно
```

В первой строке вызов конструктора класса *Gen* без указания аргумента типа приводит к появлению подобного предупреждения. Во второй строке присваивание переменной настраиваемого типа ссылки на объект несфор-мированного типа вызывает генерацию такого предупреждения.

С первого взгляда Вам может показаться, что следующая строка также приведет к генерации предупреждения компилятора, но это не так.

```
raw = iOb;    // OK, но потенциально неверно
```

Компилятор не выводит никакого предупреждения, потому что приведенное присваивание не ведет к дальнейшему снижению типовой безопасности, которая была потеряна в момент создания объекта *raw*.

Заключительное замечание: данные несформированного (*raw*) типа следует использовать только в тех случаях, когда необходимо совместное применение уже действующего и вновь создаваемого кода, содержащего настраиваемые типы. Несформированные типы — это средство, обеспечивающее совместимость, а не функциональная возможность, предназначенная для использования во вновь разрабатываемом коде.

Иерархии настраиваемых классов

Настраиваемые классы могут входить в состав иерархии классов точно так же, как и ненастраиваемые классы. Следовательно, настраиваемый класс может функционировать как суперкласс или быть производным классом (подклассом). Ключевое различие между иерархиями настраиваемых и не-настраиваемых классов состоит в том, что в иерархии настраиваемых классов любые аргументы типа, необходимые суперклассу, должны передаваться на верхние уровни иерархии всем производным классам. Этот механизм аналогичен передаче аргументов конструктора на все уровни иерархии.

Использование настраиваемого суперкласса

В листинге 3.12 приведен простой пример иерархии, в которой применяется настраиваемый суперкласс.

Листинг 3.12. Пример простой иерархии с применением настраиваемого класса

```
class Gen<T> {
  T ob;

  Gen(T o) {
    ob = o;
  }

  // Return ob.
  T getob() {
    return ob;
  }
}

// A subclass of Gen.
class Gen2<T> extends Gen<T> {
  Gen2(T o) {
    super(o);
  }
}
```

В иерархии из листинга 3.12 класс *Gen2* является расширением класса *Gen*. Рассмотрим приведенное в следующей строке объявление этого класса:

```
class Gen2<T> extends Gen<T> {
```

Параметр типа т задан в классе *Gen2* и передается в класс *Gen* с помощью ключевого слова *extends*. Это означает, что при любой передаче типа классу *Gen2* этот же параметр будет передан классу *Gen*. Например, следующее объявление:

```
Gen2<Integer> num = new Gen2<Integer>(100);
```

передает класс *Integer* как параметр типа классу *Gen.* Следовательно, объект *ob* в составляющей класса *Gen2*, унаследованной от класса *Gen*, будет
иметь тип *Integer*.

Обратите внимание также на то, что параметр типа *T* в классе *Gen2* используется только для передачи его в суперкласс *Gen*. Таким образом, даже если нет необходимости формировать настраиваемый

подкласс от настраиваемого суперкласса, в нем все равно нужно определить параметр (или параметры) типа, требующиеся для его настраиваемого суперкласса.
Конечно, в производный класс при необходимости можно добавить его собственные параметры типа. В листинге 3.13 приведен вариант иерархии из листинга 3.12, в которой в класс *Gen2* включен его собственный параметр типа.

Листинг 3.13. Пример подкласса с собственным параметром типа

```
class Gen<T> {
  T ob; // declare an Object of type T

  // Pass the constructor a reference to
  // an Object of type T.
  Gen(T o) {
    ob = o;
  }

  // Return ob.
  T getob() {
    return ob;
  }
}

// A subclass of Gen that defines a second
// type parameter, called V.
class Gen2<T, V> extends Gen<T> {
  V ob2;

  Gen2(T o, V o2) {
    super(o);
    ob2 = o2;
  }

  V getob2() {
    return ob2;
  }
}

// Create an Object of type Gen2.
class HierDemo {
  public static void main(String args[]) {

    // Create a Gen2 Object for String and Integer.
    Gen2<String, Integer> x =
      new Gen2<String, Integer>("Value is: ", 99);

    System.out.print(x.getob());
    System.out.println(x.getob2());
  }
}
```

Рассмотрим объявление версии класса Gen2 в приведенной далее строке:

```
class Gen2<T, V> extends Gen<T> {
```

Здесь *T* — параметр типа, передаваемый в класс *Gen*, а *V* — параметр типа, специфичный для класса *Gen2*. Параметр v используется для объявления объекта *ob2* и определения типа данных, возвращаемых методом *getob2()*. В методе *main* () создается объект класса *Gen2*, в котором тип *T* заменен классом *String*, а параметр *V* — классом *Integer*. Результат работы программы приведен в следующей строке:

```
Value is:  99
```

Настраиваемый подкласс

Не настраиваемый класс вполне может быть суперклассом для настраиваемого подкласса. Рассмотрим программу из листинга 3.14.

Листинг 3.14. Не настраиваемый класс в качестве суперкласса для настраиваемого подкласса

```
class NonGen {
  int num;

  NonGen(int i) {
    num = i;
  }

  int getnum() {
    return num;
  }
}

// A generic subclass.
class Gen<T> extends NonGen {
  T ob; // declare an Object of type T

  // Pass the constructor a reference to
  // an Object of type T.
  Gen(T o, int i) {
    super(i);
    ob = o;
  }

  // Return ob.
  T getob() {
    return ob;
  }
}

// Create a Gen Object.
class HierDemo2 {
  public static void main(String args[]) {

    // Create a Gen Object for String.
    Gen<String> w = new Gen<String>("Hello", 47);

    System.out.print(w.getob() + " ");
    System.out.println(w.getnum());
  }
}
```

Вывод результатов работы программы из листинга 3.14 приведен в следующей строке:

```
Hello 47
```

Обратите внимание на то, как класс *Gen* наследует класс *NonGen* в объявлении, приведенном в следующей строке:

```
class Gen<T> extends NonGen {
```

Поскольку класс *NonGen* не настраиваемый, в нем не задан аргумент типа. Следовательно, несмотря на то, что в классе *Gen* объявлен параметр *T*, он не нужен классу *NonGen* (и не может им использоваться). Таким образом, класс *Gen* унаследовал класс *NonGen* обычным способом без каких-либо специальных условий.

Сравнения типов настраиваемой иерархии во время выполнения программы

Несколько слов об операции времени выполнения *instanceof* для получения информации о типе. Она определяет, является ли объект экземпляром класса. Результат операции равен *true*, если объект заданного типа, или может быть преобразован в объект заданного типа. Операцию *instanceof* можно применять к объектам настраиваемых классов. В листинге 3.15 приведено описание класса, демонстрирующего некоторые проверки вида "если — то" типовой совместимости иерархии настраиваемых классов.

Листинг 3.15. Применение операции inetanceof к иерархии настраиваемых классов

```
class Gen<T> {
  T ob;

  Gen(T o) {
    ob = o;
  }

  // Return ob.
  T getob() {
    return ob;
  }
}

// A subclass of Gen.
class Gen2<T> extends Gen<T> {
  Gen2(T o) {
    super(o);
  }
}

 // Demonstrate runtime type ID implications of generic class hierarchy.
class HierDemo3 {
  public static void main(String args[]) {

    // Create a Gen Object for Integers.
    Gen<Integer> iOb = new Gen<Integer>(88);

    // Create a Gen2 Object for Integers.
    Gen2<Integer> iOb2 = new Gen2<Integer>(99);

    // Create a Gen2 Object for Strings.
    Gen2<String> strOb2 = new Gen2<String>("Generics Test");

    // See if iOb2 is some form of Gen2.
    if(iOb2 instanceof Gen2<?>)
      System.out.println("iOb2 is instance of Gen2");

    // See if iOb2 is some form of Gen.
    if(iOb2 instanceof Gen<?>)
      System.out.println("iOb2 is instance of Gen");

    System.out.println();
```

```
    // See if strOb2 is a Gen2.
    if(strOb2 instanceof Gen2<?>)
      System.out.println("strOb is instance of Gen2");

    // See if strOb2 is a Gen.
    if(strOb2 instanceof Gen<?>)
      System.out.println("strOb is instance of Gen");

    System.out.println();

    // See if iOb is an instance of Gen2, which its not.
    if(iOb instanceof Gen2<?>)
      System.out.println("iOb is instance of Gen2");

    // See if iOb is an instance of Gen, which it is.
    if(iOb instanceof Gen<?>)
      System.out.println("iOb is instance of Gen");

    // The following can't be compiled because
    // generic type info does not exist at runtime.
//    if(iOb2 instanceof Gen2<Integer>)
//      System.out.println("iOb2 is instance of Gen2<Integer>");
  }
}
```

Далее приведен вывод результатов работы программы:

```
iOb2 is instance of Gen2
iOb2 is instance of Gen

strOb is instance of Gen2
strOb is instance of Gen

iOb is instance of Gen
```

Переопределенные методы в настраиваемом классе

Метод в настраиваемом классе можно переопределить, как любой другой метод. Рассмотрим пример (листинг 3.16), в котором переопределен метод *getob*().

Листинг 3.16. Переопределение настраиваемого метода в настраиваемом классе

```
class Gen<T> {

  T ob; // declare an Object of type T


  // Pass the constructor a reference to

  // an Object of type T.

  Gen(T o) {

    ob = o;

  }
```

```
    // Return ob.
  T getob() {
    System.out.print("Gen's getob(): " );
    return ob;
  }
}
 // A subclass of Gen that overrides getob().
class Gen2<T> extends Gen<T> {

  Gen2(T o) {
    super(o);
  }

  // Override getob().
  T getob() {
    System.out.print("Gen2's getob(): ");
    return ob;
  }
}

// Demonstrate generic method override.
class OverrideDemo {
  public static void main(String args[ ]) {

    // Create a Gen Object for Integers.
    Gen<Integer> iOb = new Gen<Integer>(88);

    // Create a Gen2 Object for Integers.
    Gen2<Integer> iOb2 = new Gen2<Integer>(99);

    // Create a Gen2 Object for Strings.
    Gen2<String> strOb2 = new Gen2<String>("Generics Test");

    System.out.println(iOb.getob());
```

```
    System.out.println(iOb2.getob());

    System.out.println(strOb2.getob());

  }

}
```

Далее приведен вывод результатов работы программы:

```
Gen's getob(): 88
Gen2's getob(): 99
Gen2's getob(): Generics Test
```

Как показывает вывод программы, переопределенная версия метода *getob* () вызывается для объектов типа *Gen2,* а версия из суперкласса — для объектов типа *Gen.*

Настраиваемые типы и коллекции

Как уже упоминалось в начале главы, наиболее важной сферой применения настраиваемых типов можно считать подсистему *Collections Framework.* В ее состав вошли классы, реализующие разные структуры данных, такие как списки, стеки и очереди. В вышедшей версии Java 2 5.0 подсистема *Collections Framework* была полностью модифицирована для применения настраиваемых типов. Изменения затронули все классы коллекций, такие как *ArrayList, LinkedList* и *Treeset.* Это означает, что все связанные с ними классы и интерфейсы, такие как *Iterator,* теперь стали настраиваемыми. В общем, настраиваемый параметр типа задает тип объекта, хранящегося в коллекции и получаемого с помощью итератора. Применение настраиваемых типов существенно улучшает типовую безопасность *Collections Framework.* Для того чтобы лучше понять это, давайте рассмотрим пример, в котором не используются настраиваемые типы. В листинге 3.17 массив строк запоминается в объекте класса *ArrayList,* а затем его содержимое выводится на экран.

Листинг 3.17. Пример использования коллекции без применения настраиваемых типов

```
import java.util.*;

class OldStyle {
  public static void main(String args[]) {
    ArrayList list = new ArrayList();

    // These lines store Strings, but any type of Object
    // can be stored.  In old-style code, there is no
    // convenient way restrict the type of Objects stored
    // in a collection
    list.add("one");
    list.add("two");
    list.add("three");
    list.add("four");

    Iterator itr = list.iterator();
    while(itr.hasNext()) {

      // To retrieve an element, an explicit type cast is needed
      // because the collection stores only Object.
      String str = (String) itr.next(); // explicit cast needed here.

      System.out.println(str + " is " + str.length() + " chars long.");
    }
```

```
  }
}
```

До появления настраиваемых типов в коллекции запоминались ссылки на тип *Object*. Это позволяло сохранять в коллекции ссылки любого типа. В приведенной программе используется этот прием для запоминания в коллекции *list* ссылок на объекты типа *String*, но вместо них можно было сохранить ссылки на объекты любого типа.

К сожалению, хранение в коллекции ссылок на тип *Object* могло легко привести к возникновению ошибок. Во-первых, скорее Вы, а не компилятор, должны были гарантировать, что в конкретной коллекции сохраняются только объекты соответствующего типа. В листинге 3.17 коллекция *list* явно предназначена для хранения объектов типа *String*, но нет средств, способных помешать включению в коллекцию ссылки на объект другого типа. Например, компилятор не найдет никакой ошибки в следующей строке:

```
list.add(new Integer(100));
```

Поскольку коллекция *list* может запоминать ссылки на тип *Object*, ничто не помешает ей сохранить ссылку на тип *Integer* точно так же, как ссылку на тип *String*. Однако если коллекция *list* предназначалась для хранения только строк, приведенная строка кода испортит ее. А у компилятора нет информации о том, что эта строка ошибочна.

Вторая проблема использования коллекций, существовавшая до появления настраиваемых типов, заключалась в необходимости приведения типа вручную при извлечении ссылки из коллекции. Именно поэтому в листинге 3.17 выполняется преобразование в тип *String* ссылки, возвращаемой методом `next()`. До применения настраиваемых типов коллекции просто запоминали ссылки типа *Object* и не содержали больше никакой информации о типе. Следовательно, при извлечении объектов из коллекции требовалось приведение типов.

Помимо неудобств от постоянной необходимости преобразовывать извлекаемую ссылку в нужный тип, этот недостаток информации о типе приводил к довольно серьезной, но удивительно легко возникающей ошибке. Поскольку тип *Object* может быть приведен к любому типу объекта, есть возможность преобразовать ссылку, полученную из коллекции, в неверный тип. Например, если следующий оператор вставить в листинг 3.17, он откомпилируется без ошибки, но во время выполнения вызовет генерацию исключения:

```
Integer i = (Integer) itr.next();
```

Напомню, что в листинге 3.17 в коллекции *list* запоминаются только ссылки на тип String. Следовательно, когда этот оператор попытается преобразовать тип *String* в тип *Integer*, возникнет исключение недопустимого приведения типа! Поскольку эта ошибка появится во время выполнения, она может привести к серьезным последствиям.

Введение настраиваемых типов коренным образом улучшает практичность и безопасность коллекций, потому что при этом:

- Гарантируется сохранение в коллекции ссылок на объекты только определенного типа. Следовательно, в ней всегда будут храниться ссылки только известного типа.
- Исключается необходимость явного приведения типа ссылки, извлекаемой из коллекции. Вместо этого извлекаемая ссылка автоматически преобразуется в соответствующий тип. Подобное действие препятствует появлению во время выполнения программы ошибок недопустимого приведения типов и позволяет избавиться от целой категории ошибок.

Эти улучшения стали возможны, потому что каждый класс коллекции снабжается параметром типа, задающим тип коллекции. Например, класс *ArrayList* теперь объявляется следующим образом:

```
class ArrayList<E>
```

В приведенном примере *E* — тип элементов, содержащихся в коллекции. В следующей строке объявляется коллекция типа *ArrayList* для хранения объектов типа *String*:

```
ArrayList<String> list = new ArrayList<String>();
```

Теперь только ссылки типа *String* могут быть включены в коллекцию *list*. Интерфейс *Iterator* тоже настраиваемый. Теперь он объявляется следующим образом:

```
interface Iterator<E>
```

Параметр *E* — это тип элемента, на который указывает итератор. Тип должен быть согласован с типом коллекции, для которой применяется итератор. Более того, подобное соответствие типов обязательно при компиляции.
В листинге 3.18 приведен современный, с применением настраиваемых типов, вариант программы из листинга 3.17.

Листинг 3.18. Современная версия предыдущего примера с применением настраиваемых типов

```
import java.util.*;

class NewStyle {
  public static void main(String args[]) {

    // Now, list holds references of type String.
    ArrayList<String> list = new ArrayList<String>();

    list.add("one");
    list.add("two");
    list.add("three");
    list.add("four");

    // Notice that Iterator is also generic.
    Iterator<String> itr = list.iterator();

    // The following statement will now cause a compile-time eror.
//    Iterator<Integer> itr = list.iterator(); // Error!

    while(itr.hasNext()) {
      String str = itr.next(); // no cast needed

      // Now, the following line is a compile-time,
      // rather than runtime, error.
//    Integer i = itr.next(); // this won't compile

      System.out.println(str + " is " + str.length() + " chars long.");
    }
  }
}
```

Теперь в коллекции *list* содержатся ссылки только на объекты типа *String*. Более того, как показано в следующей строке, не требуется привидения типа объекта, возвращаемого методом *next*(), к типу *String*:

```
String str = itr.next(); // не требуется приведения типов
```

Приведение типа выполняется автоматически.
Благодаря поддержке несформированного (*raw*) типа нет необходимости немедленно переписывать Ваш код, предназначенный для обработки коллекций. Тем не менее в новых программах следует применять настраиваемые типы и по мере возможности обновлять разработанный ранее код. Включение средств настройки типов в подсистему *Collections Framework* — это принципиальное улучшение, которое нужно использовать везде, где это возможно.

Стирание

Обычно программисту не нужно знать, как компилятор языка Java превращает исходный текст программы в объектный код. Однако в случае настраиваемых типов некоторое общее представление об этом процессе важно, так как помогает понять, почему настраиваемые типы ведут себя так, а не иначе и почему иногда их поведение вызывает удивление. Именно поэтому уместно краткое обсуждение способа реализации настраиваемых типов в языке Java.

Необходимость сохранения совместимости с более ранними версиями Java существенно повлияла на способ введения в язык настраиваемых типов. Настраиваемый код должен быть совместим с уже существующим кодом, не использующим настраиваемые типы. Таким образом, любые изменения синтаксиса языка Java или виртуальной машины Java (JVM, Java Virtual Machine) не должны нарушать работоспособность уже действующего кода. Для удовлетворения этого ограничения в языке Java была выбрана реализация настраиваемых типов с помощью *стирания* (erasure).

В общем, стирание выполняется следующим образом. Когда компилируется исходный текст программы на языке Java, вся информация о настраиваемых типах удаляется (стирается). Это означает замену параметров типа ограничивающими их типами или верхними границами (*bound type*), например, типом *Object* при отсутствии явно заданной границы, и далее необходимое приведение типов (в соответствии с аргументами типа) для обеспечения совместимости с заданными аргументами типа. Компилятор также следит за совместимостью типов. Такой подход ведет к отсутствию какой-либо информации о настраиваемых типах во время выполнения приложения. Настраиваемые типы — только средство для разработки исходного текста программы. Для того чтобы лучше понять механизм стирания рассмотрим два класса, приведенные в листинге 3.19.

Листинг 3.19. Описание классов для демонстрации стирания

```
class Gen<T> {
  T ob; // here, T will be replaced by Object

  Gen(T o) {
    ob = o;
  }

  // Return ob.
  T getob() {
    return ob;
  }
}

// Here, T is bound by String.
class GenStr<T extends String> {
  T str; // here, T will be replaced by String

  GenStr(T o) {
    str = o;
  }

  T getstr() { return str; }
}
```

Благодаря наличию стирания некоторые действия выполняются не так, как Вы могли бы себе представить. Например, рассмотрим короткую программу из листинга 3.20, в которой создаются два объекта настраиваемого класса *Gen*.

Листинг 3.20. Пример создания объектов настраиваемого класса

```
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());
  }
}
```

Далее приведен вывод программы:

```
Gen
Gen
```

Как видите, и у объекта *iOb* и у объекта *fOb* — тип *Gen*, а не *Gen<Integer>* и *Gen<Float>*, как Вы могли бы предположить. Запомните, все параметры типа стираются в процессе компиляции. Во время выполнения действительно существуют только несформированные (*raw*) типы.

Методы-подставки

В тех случаях, когда стирание информации о настраиваемом типе в переопределенном методе производного класса не выполняет такого же стирания, как метод в его суперклассе, компилятору приходится вставлять в класс *метод-подставку* (bridge method). В подобной ситуации генерируется метод, выполняющий стирание информации о настраиваемом типе в суперклассе и уже этот метод вызывает метод производного класса, стирающий информацию о типах заданную в подклассе. Конечно, методы-подставки появляются только на уровне байт-кода, они скрыты от Вас и Вы не можете ими воспользоваться.
Несмотря на то, что методы-подставки, как правило, не должны Вас интересовать, рассмотрим очень поучительную ситуацию (листинг 3.21), в которой генерируется один из них.

Листинг 3.21. Ситуация, порождающая метод-подставку

```
class Gen<T> {
  T ob; // declare an Object of type T

  // Pass the constructor a reference to
  // an Object of type T.
  Gen(T o) {
    ob = o;
  }

  // Return ob.
  T getob() {
    return ob;
  }
}

// A subclass of Gen.
class Gen2 extends Gen<String> {

  Gen2(String o) {
    super(o);
  }

  // A String-specific override of getob().
  String getob() {
```

```
    System.out.print("You called String getob(): ");
    return ob;
  }
}

// Demonstrate a situation that requires a bridge method.
class BridgeDemo {
  public static void main(String args[]) {

    // Create a Gen2 Object for Strings.
    Gen2 strOb2 = new Gen2("Generics Test");

    System.out.println(strOb2.getob());
  }
}
```

В программе класс *Gen2* расширяет класс *Gen*, но делает это, используя версию класса *Gen* для типа *String*, как показано в его объявлении, приведенном далее:

```
class Gen2 extends Gen<String> {
```

Более того, в классе Gen2 переопределен приведенный далее метод getob(), в нем задан тип String как тип возвращаемого методом объекта:

```
// Версия для типа String, переопределяющая метод getob()
String getob() {
  System.out.print ("You called String getob(): ");
  return ob;
}
```

Все перечисленные описания вполне корректны. Но из-за стирания информации о настраиваемом типе тип метода *getob*() будет следующим:

```
Object getob()  {//...
```

Для обработки этой ситуации компилятор генерирует метод-подставку с только что приведенной сигнатурой, который вызывает версию этого же метода для типа *String*. Таким образом, если Вы просмотрите файл класса *Gen2*, полученный с помощью утилиты *javap*, то найдете в нем следующие методы:

```
class Gen2 extends Gen{
  Gen2(Java.lang.String);
  Java.lang.String getob();
  java.lang.Object getob(); // метод-подставка
}
```

как видите, в перечне есть и метод-подставка (комментарий добавил автор, а не утилита *javap*).И последнее замечание о методах-подставках. Обратите внимание на то, что единственное различие между двумя методами *getob*() — тип возвращаемого ими объекта. Обычно такая ситуация вызывает ошибку, но поскольку она возникла не в Вашем исходном тексте, виртуальная машина Java (JVM) обрабатывает ее корректно, без каких-либо проблем.

Ошибки неоднозначности

Введение настраиваемых типов породило новую категорию ошибок, которых Вы должны беречься, *ошибки неоднозначности* (*ambiguity errors*). Такая ошибка возникает, когда механизм стирания

вызывает два на вид отличающихся объявления настраиваемых типов для удаления информации об одном типе, порождая тем самым конфликт. В листинге 3.22 приведен пример, содержащий переопределение метода.

Листинг 3.22. Неоднозначность, вызванная стиранием в переопределенных методах

```
class MyGenClass<T, V> {
  T ob1;
  V ob2;

  // ...

  // These two overloaded methods are ambiguous.
  // and will not compile.
  void set(T o) {
    ob1 = o;
  }

  void set(V o) {
    ob2 = o;
  }
}
```

Обратите внимание на то, что класс *MyGenClass* содержит объявления двух настраиваемых типов: *T* и *V*. В классе *MyGenClass* делается попытка переопределения метода *set*(), основанного на параметрах типа *T* и *V*. Это выглядит разумным, так как кажется, что *T* и *V* — разные типы. Но при этом возникает два вида неоднозначности.

Во-первых (судя по описанию класса *MyGenClass*), не требуется, чтобы типы *T* и *V* всегда были разными. Например, приведенное далее создание объекта класса *MyGenClass* — совершенно правильно (в принципе):

```
MyGenClass<String, String> obj = new MyGenClass<String, String>()
```

В этом случае и *T*, и *V* замещаются типом *String*. Это делает обе версии метода `set()` одинаковыми, что, конечно же, является ошибкой.

Во-вторых, и это более существенно, стирание информации о типе превратит обе версии метода *set* () в следующую:

```
void set(Object o)
```

Таким образом, переопределение метода *set* (), которое делается в классе *MyGenClass*, — в основе своей неоднозначно.

Ошибки неоднозначности бывает трудно обнаружить. Например, если Вы знаете, что параметр типа *V* всегда будет некоторым типом *String*, можно попробовать переписать объявление класса *MyGenClass* следующим образом:

```
MyGenClass<T, V extends String> { //почти хорошо!
```

Это изменение позволит откомпилировать класс *MyGenClass* и Вы даже сможете создавать объекты класса, такие как приведенный в следующей строке:

```
MyGenClass<Integer, String> x = new MyGenClass<Integer, String>();
```

Это работающий вариант, потому что Java безошибочно определяет, какой метод следует вызывать. Но неоднозначность вернется, как только Вы попробуете ввести следующую строку:

```
MyGenClass< String, String> x = new MyGenClass< String, String>();
```

В данном случае, поскольку и `y` *T*, и `y` *V* — тип *String*, какую версию метода `set()` вызывать?

Откровенно говоря, в листинге 3.22 гораздо лучше использовать два метода с разными именами, чем пытаться переопределять метод `set()`. Часто разрешение неоднозначности приводит к переработке кода, поскольку неоднозначность или неопределенность зачастую свидетельствует о концептуальной ошибке в вашем проекте.

Некоторые ограничения применения настраиваемых типов

Существует несколько ограничений, которые Вам следует помнить при использовании настраиваемых типов. Они касаются создания объектов настраиваемого типа, статических членов класса, исключений и массивов. Рассмотрим каждое из них в отдельности.

Нельзя создавать объекты, используя параметры типа

Невозможно создать экземпляр класса, задавая его тип с помощью параметра типа. Рассмотрим пример, приведенный в листинге 3.23.

Листинг 3.23. Нельзя с помощью параметра типа T создать объект

```
class Gen<T> {
  T ob;
  Gen() {
    ob = new T(); // Illegal!!!
  }
}
```

В листинге 3.23 сделана недопустимая попытка создания экземпляра типа *T*. Причину легко понять: поскольку параметра типа *T* во время выполнения не существует, как компилятор узнает объект какого типа нужно создать? Напоминаю о том, что в процессе компиляции происходит стирание всех параметров типа.

Ограничения для статических членов класса

Ни один статический член класса не может использовать параметр типа, объявленный этим классом. Все статические члены класса, приведенного в листинге 3.24, недопустимы.

Листинг 3.24. Пример недопустимых членов класса

```
class Wrong<T> {
  // Wrong, no static variables of type T.
  static T ob;

  // Wrong, no static method can use T.
  static T getob() {
    return ob;
  }

  // Wrong, no static method can access Object
  // of type T.
  static void showob() {
    System.out.println(ob);
 }
}
```

Несмотря на то, что нельзя объявить статические члены, использующие параметры типа, объявленные в охватывающем классе, Вы можете объявлять статические настраиваемые методы, которые определяют собственные параметры типа, как было показано ранее в этой главе.

Ограничения для настраиваемого массива

Есть два важных ограничения применения настраиваемых типов, касающиеся массивов. Во-первых, нельзя создать экземпляр массива, у которого базовый тип задан с помощью параметра типа. Во-вторых, Вы не можете создать массив из ссылок на объекты конкретной версии настраиваемого типа. В листинге 3.25 показаны обе ситуации.

Листинг 3.25. Настраиваемые типы и массивы

```
class Gen<T extends Number> {
  T ob;

  T vals[]; // OK

  Gen(T o, T[] nums) {
    ob = o;

    // This statement is illegal.
    // vals = new T[10]; // can't create an array of T

    // But, this statement is OK.
    vals = nums; // OK to assign reference to existent array
  }
}

class GenArrays {
  public static void main(String args[]) {
    Integer n[] = { 1, 2, 3, 4, 5 };

    Gen<Integer> iOb = new Gen<Integer>(50, n);

    // Can't create an array of type-specific generic references.
    // Gen<Integer> gens[] = new Gen<Integer>[10]; // Wrong!

    // This is OK.
    Gen<?> gens[] = new Gen<?>[10]; // OK
  }
}
```

Как показано в листинге 3.25, можно объявить ссылку на массив типа *T*, такую как в следующей строке:

```
T valsU; // OK
```

Но нельзя создать массив из элементов типа *T*, подобно попытке, приведенной в следующей помеченной как комментарий строке:

```
// vals = new T[10]; // не может создать массив из объектов типа T
```

Вы не можете создать массив из элементов типа *T*, потому что параметр *T* не существует во время выполнения, и у компилятора нет способа узнать, массив из элементов какого типа формировать в действительности.

Тем не менее, можно передать ссылку на совместимый по типу массив в конструктор *Gen*. о при создании объекта и присвоить эту ссылку переменной *vai*, как показано в следующей строке:

```
vals = nums // можно присвоить ссылку существующему массиву
```

Приведенная строка выполнится, потому что у массива, переданного в класс *Gen*, известен тип, который в момент создания объекта будет таким же, как параметр типа *T*.

Внутри метода *main*() Вы не можете объявить массив ссылок на конкретную версию настраиваемого типа. Следующая строка:

```
// Gen<Integer> gens[] = new Gen<Integer>[10]; // Неверно!
```

не будет компилироваться. Массивы из элементов конкретной версии настраиваемого типа просто не разрешены, поскольку могут привести к потере типовой безопасности.

Однако Вы можете создать массив из ссылок на настраиваемый тип, если используете метасимвол, как показано в следующей строке:

```
Gen<?> gens[] = new Gen<?>[10]; // OK
```

Такой подход предпочтительней, чем использование массива из элементов несформированного (*raw*) типа, так как, по крайней мере, какой-то контроль типов будет выполнен.

Ограничение настраиваемых исключений

Настраиваемый класс не может расширять класс *Throwable*. Это означает, что у Вас нет возможности создавать настраиваемые классы исключений.

Заключительные замечания

Настраиваемые типы — мощное расширение языка Java, потому что они упрощают создание повторно используемого кода, обладающего типовой безопасностью. Несмотря на то, что, на первый взгляд, синтаксис настраиваемых типов может показаться несколько устрашающим, он станет Вашей второй натурой, после того, как Вы поработаете с ним какое-то время. Откровенно говоря, код с применением настраиваемых типов — это неотъемлемая часть будущего всех программистов, пишущих на языке Java.

Глава 4

Вариант *For-Each* цикла *for*

Современная теория программирования использует цикл *for-each*. Он разработан для циклической обработки объектов коллекции, такой как массив, в строгой последовательности, от начала к концу. Благодаря удобному способу обработки, предложенному циклом *for-each*, этот вид цикла быстро превратился в средство, необходимое программистам. Более ранние версии языка Java не поддерживали цикл *for-each*, но в версию Java 2 5.0 он включен. Это расширение языка наверняка порадует всех, пишущих на языке Java.

В отличие от некоторых языков программирования, таких как C#, в котором цикл *for-each* реализован с помощью ключевого слова *for-each*. Java включает функциональные возможности цикла *for-each*, усовершенствуя оператор цикла *for*. Преимущество такого подхода заключается в том, что не требуется нового ключевого слова и не нарушается работоспособность разработанного ранее кода. *Цикл for в стиле for-each* (т. е. наделенный функциональными возможностями цикла *for-each*) иногда называют *улучшенным циклом for* (enhanced for loop) и в книге будут использоваться оба эти термина.

Описание цикла *for-each*

Общая форма записи цикла for в стиле for-each приведена в следующей строке:

```
foritype itr-var : iterableObj) statement-block
```

Здесь *type*— тип, а *itr-var*— имя *переменной цикла* (iteration variable), которая будет получать элементы, содержащиеся в *iterabieObj*, последовательно один за другим от начала к концу. Объект, на который

ссылается переменная *iterabieObj*, должен быть массивом или объектом, реализующим новый интерфейс *Iterable*. В любом случае тип *type* должен совпадать (или быть совместимым) с типом элементов, извлекаемых из объекта *iterableObj*. Таким образом, в случае циклической обработки массива *type* должен быть совместим с базовым типом массива. В каждом проходе тела цикла извлекается очередной элемент объекта *iterabieObj* и запоминается в переменной *itr-var*. Цикл выполняется до тех пор, пока не будут извлечены все элементы.

Для того чтобы понять принцип действия цикла *for-each*, рассмотрим, как работает цикл *for*, для замены которого и был разработан цикл *for-each*.

В приведенном далее фрагменте используется традиционный цикл *for* для подсчета суммы значений, хранящихся в элементах массива:

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

Для вычисления суммы считывается каждый элемент в массиве *nums*, начиная с первого и заканчивая последним. Следовательно, массив целиком читается строго последовательно. Эта операция выполняется вручную с помощью индексирования *nums* переменной *i*, управляющей переменной цикла. Более того, приходится явно задавать начальное и конечное значения управляющей переменной цикла, а также ее приращение.

Цикл *for* стиля *for-each* автоматизирует приведенный в предыдущем примере цикл. А именно, он устраняет необходимость устанавливать счетчик цикла, задавая начальное и конечное значения, и вручную индексировать массив. Вместо этого он автоматически проходит весь массив, получая поочередно каждый элемент, начиная с первого и заканчивая последним. Далее приведен переписанный предыдущий пример с использованием цикла for в стиле *for-each*:

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

В каждом проходе тела цикла переменная *x* получает значение из следующего элемента массива *nums*. Таким образом, во время первого прохода переменная цикла *x* содержит значение 1, во время второго — значение 2 и т. д. При этом не только упрощается синтаксис, но и устраняются ошибки выхода за границы диапазона (*boundary errors*).

В листинге 4.1 приведена целиком программа, демонстрирующая подсчет суммы элементов массива с помощью цикла *for* в стиле *for-each*.

Листинг 4.1. Применение цикла for в стиле for-each

```
class ForEach {
 public static void main(String args[]) {
   int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   int sum = 0;
    // use for-each style for to display and sum the values
   for(int x : nums) {
     System.out.println("Value is: " + x);
     sum += x;
   }
    System.out.println("Summation: " + sum);
```

```
  }
}
```

Далее приведен вывод результатов работы программы из листинга 4.1:

```
Value is: 1
Value is: 2
Value is: 3
Value is: 4
Value is: 5
Value is: 6
Value is: 7
Value is: 8
Value is: 9
Value is: 10
Summation: 55
```

Как показывает вывод программы, цикл *for* в стиле *for-each* автоматически обрабатывает массив в цикле от наименьшего значения индекса до наибольшего его значения.

Несмотря на то, что цикл *for* в стиле *for-each* повторяется до тех пор, пока не обработаны все элементы массива, можно прервать его с помощью оператора *break*. В листинге 4.2 приведена программа, суммирующая только первые пять элементов массива nums.

Листинг 4.2. Применение оператора break в цикле for в стиле for-each

```
class ForEach2 {
  public static void main(String args[]) {
    int sum = 0;
    int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    // Use for to display and sum the values.
    for(int x : nums) {
      System.out.println("Value is: " + x);
      sum += x;
      if(x == 5) break; // stop the loop when 5 is obtained
    }
    System.out.println("Summation of first 5 elements: " + sum);
  }
}
```

Далее приведен вывод результатов работы программы из листинга 4.2:

```
Value is: 1
Value is:  2
Value is: 3
Value is: 4
Value is: 5
Summation of first 5 elements: 15
```

Ясно, что цикл for прерывается после получения пятого элемента массива.

У цикла *for-each* есть важная особенность. Его переменная цикла доступна только для чтения (*read-only*), так как она связана с обрабатываемым массивом. Другими словами, Вы не можете изменить содержимое массива, присваивая переменной цикла новое значение. Рассмотрим пример, приведенный в листинге 4.3.

Листинг 4.3. Цикл for-each по существу доступен 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; // no effect on nums
    }

    System.out.println();

    for(int x : nums)
      System.out.print(x + " ");

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

В первом цикле *for* значение переменной цикла увеличивается с помощью множителя 10. Однако это присваивание никак не влияет на содержимое обрабатываемого в цикле массива nums, что иллюстрирует второй цикл *for* в листинге 4.3, а также приведенный далее вывод результатов работы программы:

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

Обработка многомерных массивов в цикле

Улучшенная версия цикла *for* может применяться для обработки многомерных массивов (*multidimensional array*).

Напоминаю, что в языке Java многомерные массивы представляют собой массивы массивов (*arrays of arrays*) (например двухмерный массив — это массив из одномерных массивов). Это важно при циклической обработке многомерного массива, так как в каждом проходе цикла извлекается следующий массив, а не отдельный элемент. Более того, тип переменной цикла в цикле *for* должен быть совместим с типом получаемого массива. Например, в случае двухмерного массива переменная цикла должна быть ссылкой на одномерный массив. Вообще при использовании цикла *for* в стиле *for-each* для обработки массива с *N* измерениями получаемые объекты должны быть массивами с *N-1* измерениями. Для того чтобы лучше понять смысл этого рассмотрим программу, приведенную в листинге 4.4. В ней применены вложенные циклы *for* для получения всех элементов двухмерного массива в порядке их следования в строках

Листинг 4.4. Использование цикла for в стиле for-each для обработки двухмерного массива

```
class ForEach3 {
  public static void main(String args[]) {
    int sum = 0;
    int nums[][] = new int[3][5];

    // give nums some values
    for(int i = 0; i < 3; i++)
      for(int j=0; j < 5; j++)
        nums[i][j] = (i+1)*(j+1);
```

```
    // use for-each for to display and sum the values
    for(int x[] : nums) {
      for(int y : x) {
        System.out.println("Value is: " + y);
        sum += y;
      }
    }
    System.out.println("Summation: " + sum);
  }
}
```

Далее приведен вывод результатов работы программы из листинга 4.4:

```
Value is:   1
Value is:   2
Value is:   3
Value is:   4
Value is:   5
Value is:   2
Value is:   4
Value is:   6
Value is:   8
Value is:   10
Value is:   3
Value is:   6
Value is:   9
Value is:   12
Value is:   15
Summation: 90
```

В листинге 4.4 обратите особое внимание на следующую строку:

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

Посмотрите, как определена переменная цикла, *x*. Это ссылка на одномерный массив целых чисел. Подобное объявление необходимо, так как в каждом проходе цикла *for* извлекается следующий *массив* из двухмерного массива *nums*, начиная с массива заданного как *nums[0]*. Внутренний цикл *for* затем просматривает каждый из этих массивов, отображая значения каждого элемента.

Область применения цикла *for* в стиле *for-each*

Поскольку цикл *for* в стиле *for-each* может обрабатывать массив только последовательно от начала к концу, Вы можете подумать, что область его применения ограничена, но это неверно. Множество алгоритмов нуждается именно в такой обработке. Один из наиболее общих примеров — поиск. В программе, приведенной в листинге 4.5, цикл *for* используется для поиска значения в не отсортированном массиве. Цикл прерывается, когда искомое значение найдено.

Листинг 4.5. Поиск в массиве с помощью 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;
     // use for-each style for to search nums for val
    for(int x : nums) {
```

```
      if(x == val) {
        found = true;
        break;
      }
    }
      if(found)
      System.out.println("Value found!");
  }
}
```

Цикл *for* в стиле *for-each* — отличный выбор для приложения из листинга 4.5, поскольку поиск в не отсортированном массиве подразумевает последовательную проверку каждого элемента (конечно, если массив отсортирован, можно применить двоичный поиск, который потребует применения цикла другого стиля). К другим типам приложений, выигрывающих от использования цикла *for* в стиле *for-each*, относятся вычисление среднего арифметического, поиск минимального или максимального значения в наборе, поиск дублирующих значений и т. д.

Использование цикла *for в* стиле *for-each для* обработки коллекций

Несмотря на то, что в предыдущих примерах использовались массивы, цикл *for* в стиле *for-each* не ограничивается этой областью применения. Его можно применять для обработки в цикле элементов объекта любого типа, реализующего интерфейс *Iterable*. К таким типам относятся все коллекции, определенные в подсистеме *Collections Framework*, модернизация которой для Java 2 v5.0 включает и реализацию интерфейса *Iterable*. В прошлом обработка коллекции в цикле требовала от Вас применения итератора и явного вызова его методов *hasNext()* и *next()*. Улучшенный цикл for теперь автоматизирует этот процесс.

В программе, приведенной в листинге 4.6, демонстрируется использование
улучшенного цикла *for* для обработки в цикле коллекции типа *ArrayList*, содержащей числовые значения. Программа вычисляет среднее арифметическое значений списка.

Листинг 4.6. Применение улучшенного цикла for для коллекций

```
class AvgCollection {
  static double getAvg(ArrayList<Double> nums) {
    double sum = 0.0;

    for(double itr : nums)
      sum = sum + itr;

    return sum / nums.size();
  }

  public static void main(String args[]) {
    ArrayList<Double> list = new ArrayList<Double>();

    list.add(10.14);
    list.add(20.22);
    list.add(30.78);
    list.add(40.46);
```

```
    double avg = getAvg(list);

    System.out.println("List average is " + avg);

  }
}
```

Программа из листинга 4.6 выводит на экран следующую строку:

```
List average is 25.4
```

В методе *getAvg*() следующий цикл *for* в стиле *for-each*:

```
for(double itr : nums)
  sum = sum + itr;
```

заменяет приведенный далее фрагмент кода, в котором применяется явная итерация:

```
Iterator<Double> itr = nums.iterator();
  while(itr.hasNext()) {
  Double d = itr.next();
  sum = sum + d;
}
```

Как видите, улучшенный цикл *for* существенно короче и более уместен.

Еще одно замечание: если Вы хотите использовать улучшенный цикл *for* для обработки в цикле содержимого коллекции с заданным несформированным (*raw*) типом, переменная цикла должна иметь тип *Object*. Это означает, что переменную цикла следует явно преобразовать в заданный тип. Откровенно говоря, лучше всего избегать применения несформированных типов, как в цикле *for*, так и в других ситуациях. Настраиваемые типы предлагают лучшую, более безопасную и удобную альтернативу.

Создание объектов, реализующих интерфейс *Iterable*

Несмотря на то, что цикл *for* в стиле *for-each* разрабатывался в расчете на массивы и коллекции, его можно применять для обработки в цикле содержимого любого объекта, реализующего интерфейс *iterable*. Это позволяет Вам создавать классы, объекты которых можно использовать в цикле *for* в стиле *for-each*. Такая мощная функциональная возможность существенно расширяет область применения улучшенного цикла *for*.

terable — это настраиваемый интерфейс, который включен в версию Java 2 5.0. Он определен в пакете Java.lang и объявляется следующим образом:

```
interface Iterable<T>
```

Здесь *T* — тип элемента, который будет храниться в объекте, а также тип объекта, получаемого в каждом проходе тела цикла *for*.

Интерфейс *Iterable* содержит только один метод *iterator ()*, приведенный в следующей строке:

```
Iterator<T> iterator()
```

Этот метод возвращает тип *Iterator* для элементов, содержащихся в вызвавшем его объекте. Обратите внимание на то, что *iterator* — настраиваемый класс. В предыдущих версиях он не был настраиваемым типом. Он стал таким в версии Java 2 5.0, когда вся подсистема *Collections Framework* была модернизирована с включением в нее средств настройки типов. Параметр *T* задает тип элемента, который будет обрабатываться в цикле.

Поскольку метод *iterator*() интерфейса *Iterable* возвращает объект типа *Iterator*, часто класс, реализующий интерфейс *Iterable*, также реализует интерфейс *Iterator*. Определение интерфейса *iterator* приведено в следующей строке:

```
Interface Iterator<E>
```

Параметр *E* задает тип элемента, который обрабатывается в теле цикла. Методы, определенные в интерфейсе *Iterator*, перечислены в табл. 4.1.

Таблица 4.1. *Методы интерфейса Iterator*

Название метода	Описание метода
boolean hasNext()	Возвращает true, если есть еще элементы. В противном случае— false
E next()	Возвращает следующий элемент. Генерирует исключение типа NoSuchElementException, если нет следующего элемента
void remove()	Удаляет текущий элемент. Этот метод необязателен. Генерирует исключение типа UligalstateException, если делается попытка вызвать метод remove() без предшествующего вызова метода next(). Генерирует исключение типа UnsupportedoperationException, если это действие не реализовано

Как видно из описаний методов, объект, обрабатываемый в цикле, вернет значение *true* при вызове метода *hasNext*(), если у него есть элементы для дальнейшей обработки. Он вернет следующий элемент при вызове метода *next* (). Метод remove о нет необходимости реализовывать.

Когда объект с реализованным интерфейсом *Iterable* используется в цикле *for* в стиле *for-each*, выполняются скрытые вызовы методов, определенных в интерфейсах *Iterable* и *Iterator*. Таким образом, вместо явных вызовов методов *hasNext*() и *next* (), выполненных Вами вручную, цикл *for* делает это для Вас скрытно.

В листинге 4.7 приведен пример, в котором создается объект типа *StrIterable*, реализующий интерфейс *Iterable*, в нем также реализован интерфейс *Iterator*, позволяющий обрабатывать в цикле символы, составляющие строку. Внутри метода *main*() создается объект типа *StrIterable* и его элементы извлекаются поочередно с помощью цикла *for* в стиле *for-each*.

Листинг 4.7. Использование цикла for в стиле for-each для обработки объекта, реализующего интерфейс Iterable

```
import java.util.*;
// This class supports iteration of the
// characters that comprise a String.
class StrIterable implements Iterable<Character>,
                             Iterator<Character> {
  private String str;
  private int count = 0;

  StrIterable(String s) {
    str = s;
  }
```

```
   // The next three methods impement Iterator.
  public boolean hasNext() {
    if(count < str.length()) return true;
    return false;
  }
   public Character next() {
    if(count == str.length())
      throw new NoSuchElementException();

    count++;
    return str.charAt(count-1);
  }

  public void remove () {
    throw new UnsupportedOperationException();
  }
  // This method implements Iterable.
  public Iterator<Character> iterator() {
    return this;
  }
}

class ForEachIterable {
  public static void main(String args[]) {
    StrIterable x = new StrIterable("This is a test.");

    // Show each character.
    for(char ch : x)
      System.out.print(ch);

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

Далее приведен вывод результатов работы программы из листинга 4.7:

```
This is a test.
```

В методе *main* () конструируется объект *x* типа *StrIterable* и ему передается строка *"This is a test."* Эта строка сохраняется в *str* — переменной-члене класса *StrIterable*. Далее выполняется цикл *for* в стиле *for-each*, обрабатывающий содержимое объекта *x*. Каждый проход цикла извлекает следующий символ в строке. Это действие выполняется скрыто, с помощью неявных вызовов методов интерфейсов *Iterable* и *Iterator*. Вы можете использовать код листинга 4.7 как модель для обработки объекта любого типа с помощью улучшенного цикла *for*.

Глава 5

Аргументы переменной длины

Версия Java 2 5.0 содержит новое средство, упрощающее создание методов, которым приходится принимать разное количество аргументов. Это средство названо *varags*, сокращение от английского *"variable-length arguments" (аргументы переменной длины).* Метод, принимающий разное число аргументов, называется *методом с переменным количеством аргументов* (*variable-arity*).

Ситуации, требующие передачи методу переменного числа аргументов, не так уж редки. Например, метод, открывающий интернет-соединение, может принимать имя пользователя, пароль, имя файла, протокол и так далее, но при отсутствии некоторых из этих аргументов способен заменить их значениями по умолчанию. В этом случае было бы удобно передавать только те аргументы, значения которых отличаются от значений по умолчанию. Другим примером может служить новый метод *printf*(), входящий в библиотеку ввода/вывода языка Java. Как Вы увидите в *главе 9*, он принимает переменное число аргументов, которые форматируются, а затем выводятся.

Средство формирования списка с переменным числом аргументов

До выхода версии Java 2 5.0 переменное количество аргументов обрабатывалось двумя способами, ни один из которых нельзя назвать удобным. Во-первых, если максимальное число аргументов не велико и известно, можно было создать перегружаемые версии метода, по одной на каждый вариант списка передаваемых в метод аргументов. Хотя этот способ работает и подходит для некоторых случаев, он применим лишь к узкому классу задач.

Если максимальное количество аргументов было велико или неизвестно, применялся второй способ: аргументы помещались в массив, и этот массив передавался методу. Этот подход иллюстрирует программа, приведенная в листинге 5.1.

Листинг 5.1. Применение массива для передачи разного количества аргументов методу

```
// Use an array to pass a variable number of
// arguments to a method. This is the old-style
// approach to variable-length arguments.
class PassArray {
  static void vaTest(int v[]) {
    System.out.print("Number of args: " + v.length +
                     " Contents: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  public static void main(String args[])
  {
    // Notice how an array must be created to
```

```
    // hold the arguments.
    int n1[] = { 10 };
    int n2[] = { 1, 2, 3 };
    int n3[] = { };

    vaTest(n1); // 1 arg
    vaTest(n2); // 3 args
    vaTest(n3); // no args
  }
}
```

Далее приведен вывод результатов работы программы из листинга 5.1:

```
Number of args: 1 Contents: 10
Number of args: 3Contents: 1 2 3
Number of args: 0 Contents:
```

В листинге 5.1 методу *vaTest*() аргументы передаются в массиве *v*. Этот устаревший подход, тем не менее, позволяет методу *vaTest*() принимать произвольное число аргументов. Но при этом необходимо вручную упаковать аргументы в массив перед вызовом метода *vaTest*(). Кроме того, что формировать массив для каждого вызова *vaTest*() утомительно, это действие может быть источником ошибок. Новое средство создания списков с произвольным количеством аргументов предлагает более простой и удобный способ.

Аргумент переменной длины (содержащий переменное число аргументов) задается тремя точками (...). В следующей строке приведен пример описания метода *vaTest*(), использующего аргумент переменной длины:

```
static void vaTest (int...v) {
```

Эта синтаксическая запись сообщает компилятору, что *vaTest*() может вызываться без параметров, с одним или несколькими параметрами. В результате переменная v, аргумент переменной длины, неявно объявляется массивом типа *int*[ ]. Таким образом, в теле метода *vaTest*() для доступа к v используется нормальный синтаксис работы с массивом. В листинге 5.2 приведен вариант программы из листинга 5.1, но с применением средства формирования списка с переменным числом аргументов.

Листинг 5.2. Демонстрация средства формирования списка с переменным числом элементов

```
// vaTest() now uses a vararg.
  static void vaTest(int ... v) {
    System.out.print("Number of args: " + v.length +
                     " Contents: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  public static void main(String args[])
  {

    // Notice how vaTest() can be called with a
    // variable number of arguments.
    vaTest(10);      // 1 arg
    vaTest(1, 2, 3); // 3 args
    vaTest();        // no args
  }
}
```

Вывод у программы из листинга 5.2 такой же, как у примера из листинга 5.1.

В приведенной программе следует обратить внимание на два момента. Во-первых, как уже упоминалось, внутри метода *vaTest*() аргумент переменной длины, *v*, обрабатывается как массив, потому что *v* и есть массив. Символьная комбинация ... сообщает компилятору о том, что будет использоваться переменное число аргументов, и что эти аргументы будут храниться в массиве, ссылка на который содержится в переменной *v*. Во-вторых, в методе *main*() метод *vaTest*() вызывается с разным числом аргументов, включая полное их отсутствие. Аргументы автоматически помещаются в массив и передаются переменной *v*. В случае отсутствия аргументов длина массива равна *0*.

В список параметров метода могут быть включены "обычные" (обязательно указываемые при вызове метода) параметры наряду параметром переменной длины. При этом параметр, содержащий переменное число аргументов, должен быть последним в списке параметров метода. В следующей строке приведен пример корректного объявления такого метода:

```
int doIt (int a, int b, double c, int...vals) {
```

В приведенном примере первые три аргумента, используемые при вызове метода *doIt* (), соответствуют первым трем параметрам. Любые оставшиеся аргументы будут передаваться в переменной *vals*.

Запомните: параметр переменной длины (т. е. содержащий произвольное число аргументов) должен быть последним в списке параметров метода. В следующей строке приведен пример некорректного объявления:

```
Int doIt (int a, int b, double c,int-vals,Boolean stopFlag){//Ошибка!
```

В приведенной строке делается попытка объявить обычный параметр после параметра переменной длины, что недопустимо.

Следует знать еще об одном ограничении: в списке параметров метода может быть только один параметр переменной длины. В следующей строке тоже приведено неправильное объявление:

```
int doIt {int a, int b, double c, int…vals, double…morevals) { // Ошибка!
```

Попытка указать второй параметр переменной длины также недопустима.

В листинге 5.3 приведена переработанная версия метода *vaTest*(), которая принимает обычный аргумент и аргумент переменной длины.

Листинг 5.3. Использование аргумента переменной длины и обычного аргумента

```
class VarArgs2 {

  // Here, msg is a normal parameter and v is a
  // varargs parameter.
  static void vaTest(String msg, int ... v) {
    System.out.print(msg + v.length +
                     " Contents: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  public static void main(String args[])
  {
    vaTest("One vararg: ", 10);
    vaTest("Three varargs: ", 1, 2, 3);
    vaTest("No varargs: ");
  }
}
```

Далее приведен вывод результатов работы программы из листинга 5.3:

```
One vararg: I Contents: 10
Three varargs: 3 Contents: 1 2 3
No varargs:  0 contents:
```

Перегрузка методов с аргументом переменной длины

Вы можете перегружать метод с аргументом переменной длины. В листинге 5.4 приведена программа, в которой метод *vaTest* () перегружается трижды.

Листинг 5.4. Аргументы переменной длины и перегрузка метода

```
class VarArgs3 {


  static void vaTest(int ... v) {
    System.out.print("vaTest(int ...): " +
                     "Number of args: " + v.length +
                     " Contents: ");

    for(int x : v)
      System.out.print(x + " ");

    System.out.println();
  }

  static void vaTest(boolean ... v) {
    System.out.print("vaTest(boolean ...) " +
                     "Number of args: " + v.length +
                     " Contents: ");

    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 +
                     " Contents: ");

    for(int x : v)
```

```
      System.out.print(x + " ");

    System.out.println();
  }

  public static void main(String args[])
  {
    vaTest(1, 2, 3);
    vaTest("Testing: ", 10, 20);
    vaTest(true, false, false);
    vaTest(); // Ошибка: неоднозначность!
  }
}
```

Далее приведен вывод, формируемый программой из листинга 5.4:

```
vaTest (int...)  : Number of args: 3 Contents: 1 2 3
vaTest (String, int...)  : Testing: 2 Contents: 10 20
vaTest (boolean...)  : Number of args: 3 Contents: true false false
```

В листинге 5.4 показаны два варианта перегрузки метода, содержащего аргумент переменной длины. Первый вариант — *vaTest* (*boolean*...) — отличается от *vaTest* (*int*...) типом параметра переменной длины. Напоминаю, что символьная комбинация ... заставляет интерпретировать параметр как массив заданного типа. Таким образом, так же, как Вы можете перегружать методы при использовании разных типов элементов массива-параметра, Вы можете перегружать методы при использовании разных типов параметров переменной длины. В этом случае язык Java по различию типов определяет, какой из перегруженных методов вызвать.

Во втором варианте перегрузки метода — *vaTest* (*String*, *int*...) — в метод с параметром переменной длины добавлен обычный параметр. В этой ситуации язык Java анализирует число аргументов и их тип для того, чтобы определить какой вариант метода вызвать.

Аргументы переменной длины и неоднозначность

Несколько неожиданные ошибки могут возникнуть при перегрузке методов, принимающих аргументы переменной длины. Эти ошибки связаны с неопределенностью, поскольку можно создать неоднозначный вызов перегруженного метода, принимающего аргумент переменной длины. Рассмотрим программу, приведенную в листинге 5.5.

Листинг 5.5. Аргументы переменной длины, перегрузка и неоднозначность

```
// Эта программа содержит ошибку и
// не  будет компилироваться
class VarArgs4 {
   static void vaTest(int ... v) {
    System.out.print("vaTest(int ...): " +
                     "Number of args: " + v.length +
                     " Contents: ");
    for(int x : v)
```

```
      System.out.print(x + " ");
    System.out.println();
  }

  static void vaTest(boolean ... v) {
    System.out.print("vaTest(boolean ...) " +
                     "Number of args: " + v.length +
                     " Contents: ");

    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(); // Error: Ambiguous!
  }
}
```

В приведенном примере перегрузка метода *vaTest*() вполне корректна. Но программа не будет компилироваться из-за, приведенного в следующей строке.

```
vaTest(); // Ошибка: неоднозначность!
```

Поскольку аргумент переменной длины может быть пустым, приведенный вызов можно интерпретировать как вызов варианта *vaTest*(*boolean* ...) или варианта *vaTest*(*int* ...). Оба одинаково правомерны. Следовательно, вызов по сути неоднозначен.

Далее приведен еще один пример неоднозначности. Следующие варианты метода *vaTest*() изначально неоднозначны, хотя один из них принимает обычный параметр:

```
static void vaTest(int .. .v) { //...
static void vaTest(int n, int .. .v)  { //...
```

Несмотря на то, что списки параметров у вариантов метода *vaTest* () отличаются, компилятор не может выбрать вариант для следующего вызова.

```
vaTest(1)
```

Перевести этот вызов в вариант *vaTest* (*int* ...) с одним значением в аргументе переменной длины или в вариант *vaTest*(*int, int* ...) без аргумента переменной длины? Компилятор не может ответить на этот вопрос. Следовательно, возникшая ситуация неоднозначна.

Из-за ошибок неоднозначности, подобных приведенным ранее, иногда Вам следует отказаться от перегрузки и просто использовать два метода с разными именами. В некоторых случаях ошибки

неоднозначности выявляют концептуальный изъян в Вашем коде, который можно исправить, тщательно подбирая решение.

Глава 6

Перечислимые типы

Начиная с выхода первой версии языка Java, в нем было пропущено одно средство, в котором нуждалась большая часть программистов: перечислимые типы или перечисления (*enumirations*). В простейшей форме *перечислимый тип* — это список именованных констант. Хотя язык Java предлагал другие способы для реализации подобной функциональности, например переменные с модификатором поля *final*, многие программисты были лишены концептуальной чистоты перечислимого типа, который поддерживается в других популярных языках программирования. С выходом Java 2 версии 5.0 перечислимый тип стал доступен и программистам, пишущим на языке Java.

На первый взгляд перечислимый тип языка Java кажется подобным одноименному типу в других языках программирования. Но это подобие обманчиво. В языках, таких как C++, перечислимые типы — просто списки именованных целочисленных констант. В языке Java перечислимый тип определен как класс. Превращение перечислимого типа в класс существенно расширило его возможности. Например, в языке Java у перечислимого типа могут быть конструкторы, методы и поля. Таким образом, хотя перечислимые типы готовились несколько лет, реализация их в языке Java оправдала ожидания.

Описание перечислимого типа

Перечислимый тип создается с помощью нового ключевого слова *enum*. Например, в следующем фрагменте приведен пример простого перечислимого типа, в который включены разные сорта яблок.

```
// Перечислимый тип для сортов яблок.
enum Apple {
  Jonathan, GoldenDel, RedDel, Winesap, Cortland
}
```

Идентификаторы *Jonathan*, *GoldenDel* и так далее называются *константами перечислимого типа* (*enumeration constants*). Каждая объявляется неявно общедоступным, статическим членом класса *Apple*, Более того, их тип — это перечислимый тип (класс), в котором они объявлены, в нашем случае — *Apple*. В языке Java эти константы называются само типизированными (*self-typed*), причем "само" ссылается на содержащий их перечислимый тип.

После того как Вы описали перечислимый тип, можете создать переменную этого типа. Но, несмотря на то, что перечислимые типы определены как классы, Вы не можете создать экземпляр типа *enum* с помощью операции *new*. Вместо этого Вы объявляете и используете переменную перечислимого типа почти так же, как Вы поступаете с переменными одного из базовых типов, таких как *int* или *char*. В следующей строке приведен пример объявления ap как переменной класса перечислимого типа *Apple*:

```
Apple ap;
```

Поскольку у переменной ap тип *Apple*, ей можно присвоить (или она может содержать) только те значения, которые определены в классе *Apple*. В следующей строке приведено присваивание переменной *ap* значения *RedDel*:

```
ap = Apple.RedDel;
```

Обратите внимание на то, что наименованию *RedDel* предшествует имя

```
Apple.
```

Можно проверить равенство двух констант перечислимого типа с помощью операции отношения = =. В следующей строке приведен пример сравнения переменной *ap* с константой *GoldenDel*.

```
if(ap = = Apple.GoldenDel)    //
```

Значение перечислимого типа можно использовать для управления оператором *switch*. Конечно, во всех операторах *case* должны быть константы того же перечислимого типа, который используется в условном выражении оператора *switch*. Оператор *switch*, приведенный в следующих строках, вполне корректен.

```
// Использует тип enum для управления оператором switch.
switch(ap) {
  case Jonathan:     //...

  case GoldenDel:    //...

  case RedDel:       //...

  case Winsap:       //...
```

Обратите внимание на то, что в операторах *case* используются имена констант перечислимого типа без уточнения с помощью имени их класса. Просто *Winsap*, а не *Apple.Winsap*. Такой способ задания имени возможен, по тому, что перечислимый тип в выражении оператора *switch* уже неявным образом задал тип констант в операторах *case*, поэтому нет необходимости задавать в них константы с указанием их имени типа.

Когда константа перечислимого типа отображается, например, с помощью метода *println*(), на экран выводится ее имя. Далее приведен пример:

```
System.out.println(Apple.Winsap)
```

который выводит на экран имя Winsap.

В листинге 6.1 собраны все фрагменты, демонстрирующие применение перечислимого типа *Apple*.

Листинг 6.1. Перечислимый тип для сортов яблок

```
enum Apple {
  Jonathan, GoldenDel, RedDel, Winsap, Cortland
}
```

```
class EnumDemo {
  public static void main(String args[])
  {
    Apple ap;

    ap = Apple.RedDel;

    // Output an enum value.
    System.out.println("Value of ap: " + ap);
    System.out.println();

    ap = Apple.GoldenDel;

    // Compare two enum values.
    if(ap == Apple.GoldenDel)
      System.out.println("ap conatins GoldenDel.\n");
    // Use an enum to control a switch statement.
    switch(ap) {
      case Jonathan:
        System.out.println("Jonathan is red.");
        break;
      case GoldenDel:
        System.out.println("Golden Delicious is yellow.");
        break;
      case RedDel:
        System.out.println("Red Delicious is red.");
        break;
      case Winsap:
        System.out.println("Winsap is red.");
        break;
      case Cortland:
        System.out.println("Cortland is red.");
        break;
    }
  }
}
```

Далее приведен вывод результатов работы программы из листинга 6.1:

```
Value of ap: RedDel

ap contains GoldenDel.

Golden Delicious is yellow.
```

Методы values() и valueOf()

Во все перечислимые типы автоматически включены два предопределенных метода: *values* () и *vaiueOf*(). Их синтаксическая запись приведена в следующих строках.

```
public static enum-type[] values()
public static enum-type valueOf(String str)
```

Метод *values*() возвращает массив, который содержит список констант перечислимого типа. Метод *vaiueOf*() возвращает перечислимую константу, значение которой соответствует содержимому строки,

передаваемой в параметре *str*. В обоих случаях *enum-type* — перечислимый тип. В программе листинга 6.2 демонстрируется применение методов *values*() и `valueOf()`.

Листинг 6.2. Применение встроенных методов перечислимого типа

```
enum Apple {
  Jonathan, GoldenDel, RedDel, Winsap, Cortland
}
 class EnumDemo2 {
  public static void main(String args[])
  {
    Apple ap;
     System.out.println("Here are all Apple constants");
     // use values()
    Apple allapples[] = Apple.values();
    for(Apple a : allapples)
      System.out.println(a);

    System.out.println();

    // use valueOf()
    ap = Apple.valueOf("Winsap");
    System.out.println("ap contains " + ap);
  }
}
```

Далее приведен вывод результатов программы из листинга 6.2:

```
Here are all Apple constants
Jonathan
GoldenDel
RedDel
Winesap
Cortland

ap contains Winesap
```

Обратите внимание, в программе из листинга 6.2 используется цикл *for* в стиле *for-each* для обработки массива констант, полученного из метода *values*(). Для большей иллюстративности создана переменная *allapples* и ей присвоена ссылка на массив констант перечислимого типа. Но этот шаг необязателен, так как цикл *for* можно записать так, как показано в следующем фрагменте, и исключить необходимость применения переменной *allapples*:

```
for(Apple a ; Apple, values())
  System.out.println(a);
```

Теперь посмотрим, как значение, соответствующее имени *Winesap*, получается с помощью вызова метода *valueOf* ().

```
ap = Apple.valueOf("Winsap");
```

Как уже объяснялось, метод *valueOf*() возвращает значение, связанное с именем константы, представленной в виде строки.

Замечание.

Как могут отметить программисты, пишущие на C/C++, в языке Java гораздо легче, чем в других языках, выполнять переходы между понятной человеку формой представления константы перечислимого типа и ее бинарной формой представления. Это существенное преимущество реализации перечислимого типа, принятой в языке Java.

Перечислимый тип в Java — это класс

Как уже упоминалось, перечислимый тип в языке Java — это класс. Несмотря на то, что Вы не можете инициализировать переменную типа enum с помощью операции new, у перечислимого типа много функциональных возможностей таких же, как у других классов. То, что тип *enum* определен как класс, наделяет перечислимый тип в языке Java такими возможностями, каких нет у перечислимых типов в других языках программирования. К примеру, Вы можете создать для него конструкторы, добавить поля и методы и даже реализовать интерфейсы.

Важно понять, что каждая константа перечислимого типа — это объект перечислимого типа. Таким образом, если создать конструктор для типа *enum*, он будет вызываться при создании каждой константы перечислимого типа. Кроме того, у каждой константы перечислимого типа есть собственная копия переменных, определенных перечислимым типом. Рассмотрим версию описания класса *Apple*, приведенную в листинге 6.3.

Листинг 6.3. Использование конструктора, поля и метода в перечислимом типе

```
enum Apple {
  Jonathan(10), GoldenDel(9), RedDel(12), Winsap(15), Cortland(8);

  private int price; // price of each apple

  // Constructor
  Apple(int p) { price = p; }

  int getPrice() { return price; }
}

class EnumDemo3 {
  public static void main(String args[])
  {
```

```
    Apple ap;

    // Display price of Winsap.
    System.out.println("Winsap costs " +
                       Apple.Winsap.getPrice() +
                       " cents.\n");

    // Display all apples and prices.
    System.out.println("All apple prices:");
    for(Apple a : Apple.values())
      System.out.println(a + " costs " + a.getPrice() +
                         " cents.");
  }
}
```

Далее приведен вывод результатов программы из листинга 6.3:

```
Winesap costs 15 cents.

All apple prices:
Jonathan costs 10 cents.
GoldenDel costs 9 cents.
RedDel costs 12 cents.
Winesap costs 15 cents.
Cortland costs 8 cents.
```

Приведенная в листинге 6.3 версия класса *Apple* содержит следующие добавления.

Во-первых, введена переменная price для хранения цены каждого сорта яблок. Во-вторых, в описание класса *Apple* включен конструктор, которому передается цена сорта яблок. И, в-третьих, добавлен метод *getPrice*(), возвращающий значение переменной price.

Когда в методе *main* () объявляется переменная *ap*, вызывается конструктор для класса *Apple* по одному на каждую заданную константу. Обратите внимание, как конструктору передаются аргументы — они заключаются в круглые скобки и указываются после каждого имени константы, как показано в следующей строке:

```
Jonathan (10), GoldenDel (9), RedDel(12), Winesap(15), Cortland(8);
```

Эти значения передаются в конструктор *Apple* () в параметре *p*, который затем присваивает это значение переменной price. Повторю: конструктор вызывается отдельно для каждой константы.

Поскольку у каждой константы есть своя копия переменной *price*, Вы можете получить цену конкретного сорта яблок, вызвав метод *getPrice*(). В следующей строке приведен пример получения в методе *main*() цены сорта *Winesap*:

```
Apple.Winsap.getPrice()
```

Цены всех сортов яблок получены благодаря циклической обработке перечислимого типа с помощью цикла *for*. Поскольку у каждой константы перечислимого типа есть копия переменной *price*, значение, связанное с одной константой, отделено и отличается от значения, связанного с другой константой. Эта важная концепция может быть воплощена, только когда перечислимые типы реализованы как классы, что и сделано в языке Java.

Несмотря на то, что в листинге 6.3 приведен один конструктор, перечислимый тип может предложить две или несколько перегруженных форм, как любой другой класс. В листинге 6.4

приведена версия класса *Apple*, предоставляющая конструктор по умолчанию (без аргументов) (*default constructor*), присваивающий цене значение, равное *—1* для обозначения отсутствия сведений о цене.

Листинг 6.4. Использование конструктора типа enum

```
enum Apple {
  Jonathan(10), GoldenDel(9), RedDel, Winsap(15), Cortland(8);
  private int price; // price of each apple
  // Constructor
  Apple(int p) { price = p; }
  // Overloaded constructor
  Apple() { price = -1; }
  int getPrice() { return price; }
}
```

Обратите внимание на то, что в листинге 6.4 конструктору для константы *RedDel* не передается аргумент. Это означает, что вызывается конструктор по умолчанию и переменной price константы перечислимого типа *RedDel* присваивается значение, равное *-1*.

Существуют два ограничения применения перечислимых типов. Во-первых, перечислимый тип не может наследовать другой класс. Во-вторых, тип *enum* не может быть суперклассом. Это означает, что он не может расширяться. Во всех остальных ситуациях перечислимый тип действует как любой другой класс. Важно помнить, что каждая константа перечислимого типа — это объект класса, в котором она определена.

Перечислимые типы, наследующие тип enum

Несмотря на то, что Вы не можете наследовать суперкласс при объявлении типа enum, все перечислимые типы автоматически наследуют один класс *java.lang.Enum*. *Enum*— это настраиваемый класс, его объявление приведено в следующей строке:

```
abstract class Enum <E extends Enum> E>>
```

Параметр *E* обозначает перечислимый тип. У класса *Enum* нет общедоступных (*public*) конструкторов. В этом классе определено несколько методов, перечисленных в табл. 6.1 и доступных для всех перечислимых типов. Большая часть методов класса *Enum* понятна, но есть среди них четыре, заслуживающие более пристального внимания.

Таблица 6.1. Методы, определенные в классе Enum

Метод	Описание
`rotected final Object clone() throws CloneNotSupportedException`	Вызов этого метода приводит к генерации исключения типа CloneNotSupportedException. Это препятствует клонированию перечислимых типов

`final int compareTo (E e)`	Сравнивает порядковые номера двух констант одного перечислимого типа. Возвращает отрицательное значение, если у вызывающей константы порядковый номер меньше, чем у константы e, 0, если номера одинаковые, и положительное значение, если номер вызывающей константы больше номера константы e
`final boolean equals(Object` *`obj)`*	возвращает true, если вызывающий объект и переменная *obj* ссылаются на одну и ту же константу
`final Class<E> getDeclaringClass()`	Возвращает перечислимый тип, членом которого является вызывающая константа
`final int hashCode{)`	Возвращает хэш-код для вызывающего объекта
`final String name()`	Возвращает полностью определенное (unaltered) имя вызывающей константы
`final int ordinal()`	Возвращает порядковый номер константы в списке констант
`String toString()`	Возвращает имя вызывающей константы, Оно может отличаться от имени, использованного при объявлении перечислимого типа
`Static<T extends Enum<T>> T valueOf(Class<T>` *`e-type.`* `String` *`name)`*	Возвращает константу, связанную с именем, заданным в параметре *name,* перечислимого типа, указанного в параметре *e-type*

Вы можете получить порядковый номер константы перечислимого типа в списке констант. Его называют *порядковым значением* (*ordinal value*) и извлекают с помощью вызова метода *ordinal*(). Нумерация констант в списке начинается с 0. Таким образом, в перечислимом типе *Apple* у константы *Jonathan* нулевое порядковое значение, у константы *GoldenDel* оно равно *1*, а у константы *RedDel — 2* и т.д.

Существует возможность сравнения двух констант одного и того же перечислимого типа с помощью метода *compareTo*(), его сигнатура повторена в следующей строке:

```
final int compareTo (E e)
```

Важно понять, что и константа, вызывающая метод, и сравниваемая с ней константа должны принадлежать одному перечислимому типу. Если это условие нарушено, возникает ошибка на этапе компиляции. Если порядковое значение константы, вызывающей метод, меньше, чем у константы *e*, метод *compareTo*() вернет отрицательное значение. Если обе константы равны, метод возвращает 0. Если же порядковое значение у вызывающей метод константы больше, чем у константы e, возвращается положительное значение.

С помощью метода *equals* (), перегружающего метод *equals*() класса *Object*, Вы можете проверить равенство константы перечислимого типа любому другому объекту. Хотя метод способен сравнивать константу с объектом любого типа, равными будут считаться два объекта, ссылающиеся на одну и ту же константу в одном и том же перечислимом типе. В общем, если у двух констант одинаковые порядковые значения, но константы принадлежат разным перечислимым типам, метод *equals()* не вернет значение *true*.

Напоминаю о том, что Вы также можете проверять равенство двух констант с помощью операции = =. Приведенная в листинге 6.5 программа демонстрирует применение методов *ordinal*(), *compareTo*() и *equals* () .

Листинг 6.5. Демонстрация методов ordinal(), compareTo() и equales()

```
enum Apple {
  Jonathan, GoldenDel, RedDel, Winsap, Cortland
}
class EnumDemo4 {
  public static void main(String args[])
  {
    Apple ap, ap2, ap3;

    // Obtain all ordinal values using ordinal().
    System.out.println("Here are all apple constants" +
                       " and their ordinal values: ");
    for(Apple a : Apple.values())
      System.out.println(a + " " + a.ordinal());

    ap =  Apple.RedDel;
    ap2 = Apple.GoldenDel;
    ap3 = Apple.RedDel;

    System.out.println();

    // Demonstrate compareTo() and equals()
    if(ap.compareTo(ap2) < 0)
      System.out.println(ap + " comes before " + ap2);

    if(ap.compareTo(ap2) > 0)
      System.out.println(ap2 + " comes before " + ap);

    if(ap.compareTo(ap3) == 0)
      System.out.println(ap + " equals " + 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);
  }
}
```

Далее приведен вывод результатов работы программы из листинга 6.5:

```
Here are all apple constants and their ordinal values:
Jonathan 0
GoldenDel 1
RedDel 2
Winsap 3
Cortland 4

GoldenDel comes before RedDel

RedDel equals RedDel

RedDel equals RedDel

RedDel == RedDel
```

Глава 7

Метаданные

В версию Java 2 5.0 включена новая мощная функциональная возможность, названная *метаданными* (*metadata*), с ее помощью можно внедрять дополнительную информацию в исходный код. Эта

информация, именуемая *аннотациями* или *примечаниями* (*annotations*), не изменяет алгоритм работы программы. Таким образом, аннотации оставляют семантику программы неизменной. Но эта информация может использоваться различными программными средствами, как во время разработки, так и во время установки программы. Например, аннотации могут обрабатываться генератором исходного кода. Хотя компания Sun называет это новое средство метаданными, более информативный термин "возможность аннотирования программ" так же применяется.

Описание средства "метаданные"

Метаданные реализованы с помощью средств, основанных на интерфейсах. Начнем с примера. В следующих строках приведено объявление аннотации с именем *MyAnno*:

```
@interface MyAnno {
  String str();
  int val();

}
```

Во-первых, обратите внимание на то, что ключевому слову *interface* предшествует символ @. Такая запись сообщает компилятору об объявлении аннотации. В объявлении также есть два метода-члена: *String str*() и *int val* (). Все аннотации содержат только объявления методов, но Вы не должны добавлять тела этим методам. Их реализует язык Java. Кроме того, как Вы увидите, эти методы действуют скорее как поля.
Все типы аннотаций автоматически расширяют интерфейс *Annotation*. Следовательно, тип *Annotation* служит суперинтерфейсом для всех аннотаций. Он объявлен в пакете *java.lang.annotation*, В этом интерфейсе перегружены методы: *hashCode*(), *equals* () и *toString*(), определенные в типе *Object*. В нем также определен метод *annotationType*(), который возвращает объект типа *class*, представляющий вызывающую аннотацию.

После того как Вы объявили аннотацию, Вы можете использовать ее для включения примечаний в объявления. Объявление любого типа может иметь аннотацию, связанную с ним. Например, можно снабжать примечаниями классы, методы, поля, параметры и константы типа *enum*. Даже к аннотации можно добавить аннотацию. Во всех случаях аннотация предшествует объявлению.

Когда Вы применяете аннотацию, Вы задаете значения для ее методов-членов. Далее приведен фрагмент, в котором аннотация *MyAnno* сопровождает объявление метода:

```
// Аннотация для метода,
@MyAnno(str = "Annotation Example", val = 100)
public static void myMeth{) {
```

Приведенная аннотация связана с методом *myMeth*(). Рассмотрим внимательно синтаксис аннотации. За именем аннотации, начинающимся с символа @, следует заключенный в круглые скобки список инициализирующих значений для методов-членов. Для того чтобы передать значение методу-члену, имени этого метода присваивается значение. Таким образом, в приведенном фрагменте строка "*Annotation Example*" (Пример аннотации) присваивается методу *str*, члену аннотации типа *MyAnno*. При этом в присваивании после имени *str* нет круглых скобок. Когда методу-члену передается инициализирующее значение, используется только имя метода. Следовательно, в данном контексте методы-члены выглядят как поля.

Задание правил сохранения

Прежде чем продолжить изучение метаданных, следует обсудить *правше сохранения аннотаций* (*annotation retention policies*). Правила сохранения определяют, в какой момент аннотации уничтожаются. В языке Java описаны три таких правила — SOURCE, CLASS И RUNTIME, включенные в перечислимый тип `Java.lang.annotation.RetentionPolicy`.

Аннотация, заданная с правилом сохранения SOURCE, существует только в исходном тексте программы и отбрасывается во время компиляции.

Аннотация, заданная с правилом сохранения CLASS, помещается в файл *.class* в процессе компиляции. Но она не доступна в JVM (виртуальная машина Java) во время выполнения.
Аннотация, заданная с правилом сохранения RUNTIME, помещается в факт .class в процессе компиляции и доступна в JVM во время выполнения. Следовательно, правило RUNTIME предлагает максимальную продолжительность существования аннотации.

Правило сохранения для аннотации задается с помощью одной из встроенных аннотаций @Retention. Ее синтаксическая запись приведена в следующей строке:
@Retention *{retention-policy)*
В этой записи *retention-policy* задается одной из описанных ранее констант перечислимого типа. Если для аннотации не задано правило сохранения, по умолчанию задается правило CLASS.

В приведенном далее фрагменте аннотация типа *MyAnno* использует аннотацию @Retention для задания правила сохранения RUNTIME. Следовательно, аннотация типа MyAnno будет доступна в *JVM* во время выполнения программы:

```
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str() ;
  int val();
}
```

Получение аннотаций во время выполнения программы с помощью рефлексии

Несмотря на то, что аннотации созданы для других программных средств разработки и распространения или установки, их может запросить любая Java-программа во время выполнения с помощью средств рефлексии. Многие читатели, вероятно, знают, *рефлексия* — это функциональная возможность, позволяющая получать информацию о классе во время выполнения. Интерфейс рефлексии (Reflection API) содержится в пакете *java.lang.reflect*. Существует ряд способов применения рефлексии, но мы не будем обсуждать их все в этой книге. Рассмотрим несколько примеров применения этого программного средства для получения информации об аннотации.

Первый вариант использования рефлексии — получение объекта типа *class*, представляющего класс, аннотации которого Вы хотите получить. Существуют разные способы получения объекта типа *class*. Простейший — вызов метода *getClass*(), который определен в классе *Object*. В следующей строке приведен общий вид такого вызова:

```
final Class<? Extends Object>getClass()
```

Метод возвращает объект типа *Class*, который представляет вызывающий объект. Учтите, что тип *class* теперь, настраиваемый.
После того, как Вы получили объект типа *Class*, можно использовать его методы для извлечения информации об отдельных элементах, объявленных в классе, включая аннотации. Если Вы хотите получить аннотации, связанные с конкретным элементом, определенным в пределах класса, то сначала необходимо извлечь объект, представляющий этот элемент. Например, тип класс предоставляет (среди прочих) методы: *getMethod*(), *getField*() и *getConstructor*(), получающие сведения о методе, поле и конструкторе соответственно. Эти методы возвращают объекты типа *Method*, *Field* и *Constructor*.

Для того чтобы лучше понять процесс, рассмотрим пример, в котором извлекаются аннотации, связанные с методом. Для этого Вы сначала получаете объект типа *Class*, представляющий класс, а затем вызываете метод get-*Method* о для этого объекта типа *Class*, задавая имя интересующего Вас метода. Синтаксическая запись вызова метода *getMethod*() приведена в следующей строке: *Method getMethod*(*String methName, Class ... paramTypes)*
Имя метода передается в параметре *methName.* Если у метода есть аргументы, то в параметре *paramTypes* следует указать объекты типа class, представляющие их типы. Обратите внимание на то, что *paramTypes* — параметр переменной длины (*varargs parameter*). Это означает, что можно задать столько параметров, сколько нужно, включая их отсутствие. Метод `getMethod`() возвращает объект типа

Method, который представляет метод. Если метод не найден, генерируется исключение *NoSuchMethodException*.

Из объекта типа *Class*, *Method*, *Field*, *Constructor* или *Package* Вы можете получить конкретную аннотацию, связанную с объектом, с помощью вызова метода *getAnnotation*(), синтаксическая запись которого приведена в следующей строке:

```
<T extends Annotation> T getAnnotation(class<T>annoType)
```

Параметр *annoType* — это объект типа *Class*, который представляет интересующую вас аннотацию. Он возвращает ссылку на аннотацию. Используя ее, Вы можете получить значения, связанные с методами-членами аннотации.

В листинге 7.1 собраны все приведенные ранее фрагменты кода и использована рефлексия для отображения аннотации, связанной с методом.

Листинг 7.1. Применение рефлексии для отображения аннотации, связанной с методом

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// An annotation type declaration.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str();
  int val();
}

class Meta {
  // Annotate a method.
  @MyAnno(str = "Annotation Example", val = 100)
  public static void myMeth() {
    Meta ob = new Meta();
    // Obtain the annotation for this method
    // and display the values of the members.
    try {
      // First, get a Class Object that represents
      // this class.
      Class c = ob.getClass();
      // Now, get a Method Object that represents
      // this method.
      Method m = c.getMethod("myMeth");
   // Next, get the annotation for this class.
      MyAnno anno = m.getAnnotation(MyAnno.class);

      // Finally, display the values.
      System.out.println(anno.str() + " " + anno.val());
    } catch (NoSuchMethodException exc) {
       System.out.println("Method Not Found.");
    }
  }
  public static void main(String args[]) {
    myMeth();
  }
}
```

Далее приведен вывод результатов работы программы из листинга 7.1:

```
Annotation Example 100
```

В программе используется рефлексия для получения и отображения значений методов *str* и *val* аннотации типа *MyAnno*, связанной с методом *myMeth*() в классе *Meta*.

Листинг 7.2 Применение рефлексии для отображения аннотации, связанной с методом, имеющим аргументы

```
import java.lang.annotation.*;
import java.lang.reflect.*;
 @Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str();
  int val();
}
 class Meta {
  // myMeth now has two arguments.
  @MyAnno(str = "Two Parameters", val = 19)
  public static void myMeth(String str, int i)
  {
    Meta ob = new Meta();
   try {
      Class c = ob.getClass();
    // Here, the parameter types are specified.
      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("Method Not Found.");
    }
  }
   public static void main(String args[]) {
    myMeth("test", 10);
  }
}
```

Листинг 7.3. Получение всех аннотаций 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 = "An annotation test class")
@MyAnno(str = "Meta2", val = 99)
class Meta2 {
   @What(description = "An annotation test method")
  @MyAnno(str = "Testing", val = 100)
  public static void myMeth() {
    Meta2 ob = new Meta2();
 try {
      Annotation annos[] = ob.getClass().getAnnotations();
      // Display all annotations for Meta2.
      System.out.println("All annotations for Meta2:");
      for(Annotation a : annos)
        System.out.println(a);
      System.out.println();
      // Display all annotations for myMeth.
      Method m = ob.getClass( ).getMethod("myMeth");
      annos = m.getAnnotations();
    System.out.println("All annotations for myMeth:");
      for(Annotation a : annos)
        System.out.println(a);
  }
 catch (NoSuchMethodException exc) {
       System.out.println("Method Not Found.");
    }
  }  public static void main(String args[]) {
    myMeth();
  }
}
```

Далее приведен вывод результатов работы программы из листинга 7.3:

```
All annotations for Meta2:
```

```
@What (description=An annotation test class)
SMyAnno (str=Meta2, val=99)

All annotations for myMeth:
@What {description=An annotation test method)
@MyAnno (str=Testing, val=100}
```

В листинге 7.3 используется метод *getAnnotations*() для получения массива всех аннотаций, связанных с классом *Meta2* и методом *myMeth*(). Как уже объяснялось, метод *getAnnotations*() возвращает массив объектов типа Annotation. Напоминаю, что Annotation — это суперинтерфейс всех интерфейсов аннотаций, в котором перегружены методы *equals*() и *toString*() класса *Object*. Следовательно, когда выводится ссылка на интерфейс *Annotation*, вызывается его метод *toString*() для генерации строки, описывающей аннотацию, как показано в выводе результатов программы из листинга 7.3.

Интерфейс AnnotatedElement

Методы *getAnnotation*() и *getAnnotations*(), применявшиеся в предыдущих примерах, определены в новом интерфейсе *AnnotatedElement*, который включен в пакет *Java.lang.reflect*. Этот интерфейс поддерживает рефлексию для аннотаций и реализован в классах *Method*, *Field*, *Constructor*, *Class* и *Package*. В нем определены методы, перечисленные в табл. 7.1. Следовательно, эти методы доступны для любого объекта, который сопровождается аннотацией.

Таблица7.1. *Методы, определенные в интерфейсе AnnotatedElement*

Методы	Описание
`<T extends Annotation> T getAnno-tatioiUClass<T> annoType)`	Возвращает аннотацию типа *annoType*, связанную с вызывающим объектом
`Annotation[] getAnnotations()`	Возвращает массив, содержащий все аннотации, связанные с вызывающим объектом
`Annotation[] getDeclaredAnnota-tions()`	Возвращает массив, содержащий все неунаследованные аннотации, связанные с вызывающим объектом
`Boolean isAnnotationPre-sent(Class<? extends Annotation> annoType)`	Возвращает значение true, если аннотация заданного типа *annoType* связана с вызывающим объектом. В противном случае возвращает false

Использование значений по умолчанию

Вы можете передавать значения по умолчанию методам-членам аннотаций, которые будут использоваться, если не задано значение при вставке аннотации. Значение по умолчанию указывается с помощью ключевого слова *default* в объявлении метода-члена. Синтаксическая запись такого объявления приведена в следующей строке:

```
type member() default value;
```

В приведенной записи значение *value* должно иметь тип, совместимый с типом *type*.

Далее приведен фрагмент, в котором в объявление интерфейса *@MyAnno* включены значения по умолчанию:

```
// Объявление типа аннотации и включение в него значений по умолчанию.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str() default "Testing";
  int val() default 9000;
}
```

В приведенном объявлении методу-члену *str* передается значение по умолчанию "*Testing*", а методу *val* — 9000. Это означает, что при использовании аннотации типа *@MyAnno* не нужно задавать никаких значений. Однако при желании каждому методу-члену или обоим можно передать другие значения. Далее приведены четыре варианта применения аннотации типа *@MyAnno*:

```
@MyAnno() // для str и val используются значения по умолчанию
@MyAnno(str = "some String") // для val использовано значение по
                             // умолчанию
@MyAnno(val = 100) // для str использовано значение по умолчанию
@MyAnno(str = "Testing"/ val = 100) // не используются значения по
                                    // умолчанию
```

В листинге 7.4. приведен пример программы, в которой демонстрируется применение значений по умолчанию в аннотации.

Листинге 7.4 Использование значений по умолчанию в аннотации

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// An annotation type declaration and include defaults.
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnno {
  String str() default "Testing";
  int val() default 9000;
}

class Meta3 {

  // Annotate a method using the default values.
  @MyAnno()
  public static void myMeth() {
    Meta3 ob = new Meta3();

    // Obtain the annotation for this method
    // and display the values of the members.
    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("Method Not Found.");
    }
  }

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

Далее приведен вывод результатов работы программы из листинга 7.4:

```
Testing 9000
```

Аннотации-маркеры

Аннотация-маркер (marker annotation) — это специальный тип аннотации, не содержащий методов-членов. Единственная цель такой аннотации — пометить объявление. В этом случае достаточно присутствия аннотации. Лучше всего для проверки наличия аннотации-маркера использовать метод *isAnnotationPresent*(), который определен в интерфейсе *AnnotatedElement* и, следовательно, доступен для объектов типа *Class, Field, Method, Constructor* и *Package*.

В листинге 7.5 приведен пример применения аннотации-маркера. Для определения Присутствия маркера используется метод *isAnnotationPresent* ().
Поскольку у интерфейса аннотации-маркера нет методов-членов, достаточно определить его наличие.

Листинг 7.5. Применение аннотации-маркера

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// A marker annotation.
@Retention(RetentionPolicy.RUNTIME)
@interface MyMarker { }

class Marker {

  // Annotate a method using a marker.
  // Notice that no ( ) is needed.
  @MyMarker
  public static void myMeth() {
    Marker ob = new Marker();

    try {
      Method m = ob.getClass().getMethod("myMeth");

      // Determine if the annotation is present.
      if(m.isAnnotationPresent(MyMarker.class))
        System.out.println("MyMarker is present.");

    } catch (NoSuchMethodException exc) {
       System.out.println("Method Not Found.");
    }
  }
  public static void main(String args[]) {
    myMeth();
```

```
  }
}
```

В приведенном далее выводе результатов работы программы из листинга 7.5 подтверждается наличие аннотации типа *@MyMarker*:

```
MyMarker is present.
```

Обратите внимание на то, что в программе не нужно вставлять круглые скобки после имени интерфейса *@ MyMarker* при создании аннотации. Таким образом, когда вставляется аннотация *MyMarker*, просто указывается ее имя, как показано в следующей строке:

```
@MyMarker
```

Не будет ошибкой применение пустых скобок, но они не нужны.

Одночленные аннотации

Одночленная аннотация (single-member annotation) — это еще один специальный тип аннотации, содержащий единственный метод-член. Она применяется так же, как и обычная аннотация, за исключением того, что для этого типа аннотации допускается краткая условная форма задания значения для метода-члена. Если есть только один метод-член, Вы можете просто указать значение для этого метода-члена, когда создается аннотация; при этом не нужно указывать имя метода-члена. Но для того чтобы воспользоваться краткой формой, следует для метода-члена использовать имя *value*.

Листинг 7.6. Создание и применение одночленной аннотации

```
import java.lang.annotation.*;
import java.lang.reflect.*;

// A single-member annotation.
@Retention(RetentionPolicy.RUNTIME)
@interface MySingle {
  int value(); // this variable name must be value
}

class Single {

  // Annotate a method using a marker.
  @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()); // displays 100

    } catch (NoSuchMethodException exc) {
       System.out.println("Method Not Found.");
    }
  }

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

Как и ожидалось, программа выводит на экран значение *100*. В листинге 7.6 аннотация типа *@MySingle* используется для создания примечания к методу *myMeth* (), как показано в следующей строке:

```
@MySingle(100)
```

Обратите внимание на то, что не указано выражение `value =`.

Вы можете применять синтаксическую запись одночленной аннотации, когда создаете аннотацию, у которой есть и другие методы-члены, но для них должны быть заданы значения по умолчанию. В приведенном далее фрагменте добавлен метод-член *xyz* со значением по умолчанию, равным *0*.

```
@interface SomeAnno {
  int valued ;
  int xyz() default 0;
}
```

Если Вы хотите использовать значение по умолчанию для метода *xyz*, можно создать аннотацию типа *@SomeAnno*, как показано в следующей строке, применяя синтаксическую запись для одночленной аннотации с указанием значения для метода *value*:

```
@SomeAnno  (88)
```

В этом случае в метод *xyz* передается нулевое значение по умолчанию, а метод *value* получает значение *88*. Конечно, передача в метод *xyz* значения, отличного от значения по умолчанию, потребует явного задания имен обоих методов-членов, как показано в следующей строке:

```
@SomeAnno (value = 88, xyz = 99)
```

Помните, у одночленной аннотации имя метода-члена должно быть *value*.

Встроенные аннотации

В языке Java определено семь типов встроенных аннотаций, четыре типа — *@Retention*, *@Documented*, *@Target* и *@Inherited* — импортируются из пакета *java.lang.annotation*. Оставшиеся три — *@Ovveride*, *@Deprecated* и *@Suppresswarnings* — из пакета *java.lang*. Далее описаны все встроенные типы.

@Retention

Тип *@Retention* разработан для применения в качестве аннотации к другой аннотации. Он задает правило хранения, как описывалось ранее в этой главе.

@Documented

Аннотация типа *@Documented* — это маркер, сообщающий средству обработки о том, что аннотация должна быть документирована. Этот тип разработан для применения в качестве аннотации к объявлению аннотации.

@Target

Аннотация типа *@Target* задает типы объявлений, которые могут снабжаться аннотациями. Она разработана для применения в качестве аннотации к другой аннотации. Тип *@Target* принимает один аргумент, который должен быть константой перечислимого типа *ElementType*. Этот аргумент задает типы объявлений, которые могут снабжаться аннотациями. В табл. 7.2 перечислены константы и соответствующие им типы объявлений.

Таблица 7.2. *Перечень констант ElementType и типы объектов, сопровождаемых аннотациями*

Константа для типа @Target	**Объект, снабженный аннотацией**
ANNOTATION_TYPE	Другая аннотация
CONSTRUCTOR	Конструктор
FIELD	Поле
LOCAL_VARIABLE	Локальная переменная

METHOD	Метод
PACKAGE	Пакет
PARAMETER	Параметр
TYPE	Класс, интерфейс или перечислимый тип

Вы можете задать одну или несколько констант из табл. 7.2 в аннотации типа *@Target*. При указании нескольких значений их следует заключить фигурные скобки. Например, если необходимо создать аннотации к полям и локальным переменным, можно использовать аннотацию типа *@Target*, описанную в следующей строке:

```
@Target( { ElementType.FIELD, ElementType.LOCAL_VARIABLE } )
```

@Inherited

Тип *@inherited* предназначен для аннотации-маркера, которая может применяться только для объявления другой аннотации. Более того, аннотация этого типа влияет только на аннотации к объявлениям классов. Тип *@Inherited* вызывает наследование производным классом аннотации суперкласса. Следовательно, когда определенная аннотация запрашивается в производном классе и не обнаруживается в нем, ее поиск продолжается в суперклассе. Если в суперклассе есть эта аннотация, и она снабжена аннотацией-маркером типа *@Inherited*, аннотация извлекается.

@Ovveride

Аннотация типа *@Ovveride* — это аннотация-маркер, которая может применяться только для методов. Метод, снабженный аннотацией типа *@Ovveride*, должен переопределять метод суперкласса. Если этого не происходит, возникает ошибка на этапе компиляции. Аннотация этого типа используется для подтверждения того, что метод суперкласса действительно переопределен (*overridden*), а не просто перегружен (*overloaded*).

@Deprecated

Аннотация-маркер типа *@Deprecated* указывает, что объявление является устаревшим или вышедшим из употребления и было заменено новым вариантом.

@Suppress Warnings

Тип *@SuppressWarnings* указывает, что одно или несколько предупреждений компилятора должны быть подавлены. Эти предупреждения задаются с помощью имен в строковой форме. Данным типом аннотации можно снабжать объявления любого типа.

Несколько ограничений

Существует ряд ограничений, действующих при объявлении аннотаций. Во-первых, ни одна аннотация не может наследовать другую аннотацию.
Во-вторых, все методы, объявляемые в аннотации, не должны иметь параметров. Кроме того, они должны возвращать один из перечисленных далее типов:

- простой тип, такой как *int* или *double*;
- объект типа *String* или *class*;
- тип *enum* ;
- другой тип аннотации;
- массив элементов, одного из перечисленных в предыдущих пунктах типов.

Аннотации не могут быть настраиваемыми. Другими словами, они не могут принимать параметры типа. И последнее, в них нельзя задавать ключевое слово *throws*.

Глава 8

Статический импорт

Новое средство, которое многие программисты сочтут весьма полезным, называется *статическим импортом* (static import) и расширяет возможности ключевого слова *import*. Оператор языка *import*, снабженный ключевым словом *static*, следующим за ключевым словом *import*, может применяться для импорта статических членов класса или интерфейса. Благодаря наличию статического импорта стало возможным ссылаться на статические члены непосредственно по их именам без уточнения имени их класса. Это упрощает и сокращает синтаксис обращения к статическому члену.

Описание статического импорта

Для того чтобы понять пользу статического импорта, начнем с примера, в котором это средство не применяется. В листинге 8.1 вычисляется гипотенуза прямоугольного треугольника. В программе используются два статических метода из встроенного в язык Java класса Math, являющегося частью пакета *Java.lang*. Первый метод, *Math.pow*(), возвращает значение, возведенное в определенную степень. Второй — *Math.sqrt*() — возвращает квадратный корень своего аргумента.

Листинг 8.1. Вычисление гипотенузы прямоугольного треугольника

```
class Hypot {
  public static void main(String args[]) {
    double side1, side2;
    double hypot;

    side1 = 3.0;
    side2 = 4.0;

    // Notice how sqrt() and pow() must be qualified by
    // their class name, which is Math.
    hypot = Math.sqrt(Math.pow(side1, 2) +
                      Math.pow(side2, 2));
```

```
    System.out.println("Given sides of lengths " +
                       side1 + " and " + side2 +
                       " the hypotenuse is " +
                       hypot);
  }
}
```

Поскольку *pow*() и *sqrt*() — статические методы, они должны вызываться с использованием имени их класса, *Math*. Это приводит к довольно громоздкому вычислению гипотенузы, приведенному в следующем фрагменте:

```
hypot = Math.sqrt(Math.pow(side1, 2) + Math.pow(side2, 2));
```

Как показывает этот простой пример, необходимость указания имени класса при каждом обращении к методам *pow*() и *sqrt*() (или к любому другому математическому методу, такому как *sin* (), *cos* () и *tan*()) может превратиться в утомительное занятие.
Благодаря использованию статического импорта, Вы можете избавиться от обязательного указания имени класса в подобных ситуациях, как показано в новой версии программы (листинг 8.2) из листинга 8.1.

Листинг 8.2. Применение статического импорта для имен методов sqrt () и pow()

```
import static java.lang.Math.sqrt;
import static java.lang.Math.pow;

// Compute the hypotenuse of a right triangle.
class Hypot {
  public static void main(String args[]) {
    double side1, side2;
    double hypot;

    side1 = 3.0;
    side2 = 4.0;

    // Here, sqrt() and pow() can be called by themselves,
    // without their class name.
    hypot = sqrt(pow(side1, 2) + pow(side2, 2));

    System.out.println("Given sides of lengths " +
                       side1 + " and " + side2 +
                       " the hypotenuse is " +
                       hypot);
  }
}
```

В листинге 8.2 имена *sqrt* и *pow* импортированы в область видимости благодаря приведенным далее операторам статического импорта:

```
import static Java.lang.Math.sqrt;
import static Java.lang.Math.pow;
```

После включения в листинг 8.2. этих операторов нет необходимости уточнять имена методов

sqrt () и *pow* () с помощью имени их класса. Следовательно, вычисление гипотенузы можно описать более удобным способом, как показано в следующей строке:

```
hypot = sqrt(powfsidel, 2) + pow(side2, 2));
```

Как видите, этот вариант читается гораздо легче.

Общий вид оператора статического импорта

Существуют два варианта синтаксической записи оператора *import static*. Первый, приведенный в листинге 8.2, вариант импортирует единичное имя. Его общий вид приведен в следующей строке:

```
import static pkg.type-name.static-member-name;
```

В этой записи *type - name* — это имя класса или интерфейса, содержащее нужный статический член. Полное имя пакета, в который входит заданный класс или интерфейс, определено в *pkg*. А имя члена содержится в *static-member-name*.

Во втором варианте статического импорта, общий вид которого приведен в следующей строке, импортируются все статические члены:

```
import static pkg. type-name.*;
```

Если Вы будете использовать много статических методов или полей, определенных в классе, второй вариант записи позволит импортировать их без явного перечисления. Таким образом, программа из листинга 8.2 могла бы с помощью единственного оператора *import* импортировать оба метода: *sqrt* () и *pow* () (и все остальные статические члены класса *Math*) в область видимости следующим образом:

```
import static Java.lang.Math.*;
```

Конечно, статический импорт не ограничен только классом *Math* или только методами. В следующей строке приведен пример импортирования статического поля *System.out*:

```
import static Java.lang.System.out;
```

После вставки этого оператора, можно вывести на консоль без уточнения имени стандартного вывода *out* именем *system*, как показано в следующей строке:

```
out.println("After importing System.out, you can use out directly.")
```

Хорошо ли импортировать поле *system.out* таким образом— это вопрос для обсуждения. Хотя такой импорт укорачивает оператор, после его применения у человека, читающего программу, пропадает уверенность в том, что имя out ссылается на поле *System.out*.

Импорт статических членов классов, созданных Вами

Помимо импорта статических членов классов и интерфейсов, определенных в прикладном программном интерфейсе (API) языка Java, Вы можете использовать это средство для импорта статических членов классов и интерфейсов, созданных Вами. Рассмотрим класс *Msg*, приведенный в листинге 8.3. Обратите внимание на то, что он содержится в пакете, названном *MyMsg*.

Листинг 8.3. Статический импорт членов созданного Вами класса

```
package MyMsg;

public class Msg {
  public static final int UPPER = 1;
  public static final int LOWER = 2;
  public static final int MIXED = 3;

  private String msg;

  // Display a message in the specified case.
  public void showMsg(int how) {
    String str;

    switch(how) {
```

```
      case UPPER:
        str = msg.toUpperCase();
        break;
      case LOWER:
        str = msg.toLowerCase();
        break;
      case MIXED:
        str = msg;
        break;
      default:
        System.out.println("Invalid command.");
        return;
    }

    System.out.println(str);
  }

  public Msg(String s) { msg = s; }
}
```

В классе *Msg* инкапсулирована строка, которая может выводиться на экран в первоначальном виде (содержит как заглавные, так и строчные буквы), в верхнем регистре (только заглавные буквы) или в нижнем регистре (только строчные буквы) в зависимости от значения, переданного в метод *showMsg*(). Значения, определяющие, какой регистр используется,— это целочисленные поля, описанные с модификаторами *static final* и названные UPPER, LOWER и MIXED. Обычно эти члены должны уточняться с помощью имени класса, например Msg.UPPER. Предположим, что объект класса Msg назван *m*, для вывода строки в нижнем регистре пришлось бы вызывать метод *showMsg*(), как показано в следующей строке:

```
m.showMsg(Msg.LOWER);
```

Но если Вы статически импортируете эти значения, то сможете непосредственно использовать их имена следующим образом:

```
m.showMsg(LOWER);
```

В листинге 8.4 показан процесс импорта статических членов класса *MyMsg.Msg* и в дальнейшем использование констант UPPER, LOWER И MIXED без уточнения их имен.

Листинг 8.4. Статический импорт статических полей, определенных пользователем

```
import MyMsg.*;

import static MyMsg.Msg.*;

class Test {
  public static void main(String args[]) {
    Msg m = new Msg("Testing static import.");

    m.showMsg(MIXED);
    m.showMsg(LOWER);
    m.showMsg(UPPER);
  }
}
```

Неоднозначность

Используя средство статического импорта, нужно быть очень внимательным, чтобы не создать неоднозначных ситуаций. Если у двух классов или интерфейсов используется одно и то же имя для статического члена, и оба эти класса или интерфейса импортируются в один и тот же блок компиляции, компилятор не знает какое из этих имен выбрать, если они применяются без уточнения с

помощью имени класса. Например, предположим, что в пакет *MyMsg* из листинга 8.4 включен приведенный в листинге 8.5 класс, в котором также объявлено статическое поле, названное UPPER.

Листинг 8.5. Неоднозначность, возникающая при статическом импорте

```
package MyMsg;

public class X {
  public static final int UPPER = 11;
  // ...
}
```

Если статические члены класса из листинга 8.5 импортируются в программу, которая также импортирует статические члены класса *Msg*, возникнет неоднозначность, как только обнаружится идентификатор UPPER. Например, в приведенных далее операторах статического импорта:

```
import static MyMsg.Msg.*;
import static MyMsg.X.*;
```

неуточненное имя UPPER ссылается на поле Msg.UPPER или на поле X.UPPER?

Предупреждение

Несмотря на удобство, предоставляемое статическим импортом, важно не злоупотреблять этим программным средством. Помните, что в языке Java библиотеки собраны в пакеты для того, чтобы избежать конфликтов пространств имен. Когда Вы импортируете статические члены, Вы переносите их в глобальное пространство имен (*global namespace*). Следовательно, увеличиваете вероятность возникновения конфликтов в пространстве имен, неоднозначности и непреднамеренного сокрытия других имен. Если Вы используете статический член в программе один или два раза, нет смысла импортировать его. Кроме того, некоторые статические имена, такие как *system.out*, настолько легко узнаваемы, что у Вас не появится желания импортировать их. Статический импорт желателен в тех случаях, когда статический член используется многократно, например, выполняя последовательности математических вычислений. Итак, Вам следует применять это средство программирования, но не злоупотреблять им.

Глава 9

Форматированный ввод/вывод

В версию Java 2 5.0 добавлена функциональная возможность форматирования вывода, которую долго ждали программисты. Несмотря на то, что язык Java всегда предлагал богатый и разнообразный *API*, в нем до сих пор не было легкого способа создания форматированного текстового вывода, особенно для числовых значений. В более ранних версиях Java, снабженных такими классами, как *NumberFormat*, *DateFormat* и *MessageFormat*, обеспечивались полезные функциональные возможности для форматирования вывода, но они были не слишком удобны. Более того, в отличие от языков C и C++, которые поддерживают легко осваиваемое и широко используемое семейство функций *printf*(), которое

предлагает простой способ форматирования вывода, язык Java прежде не содержал таких методов. Причина заключалась в том, что форматирование в *printf*-стиле требует применения аргументов переменной длины (*varargs*), которые не поддерживались в языке Java до выхода **Java** 2 версии 5.0. Теперь, когда аргументы переменной длины включены в язык, добавить средства форматирования общего назначения стало просто.

В версию Java 2 5.0 включена поддержка чтения форматированных входных данных. Несмотря на то, что форматирование ввода всегда было возможно, оно требовало усилий больших, чем хотелось бы подавляющей части программистов. Теперь можно легко читать все типы числовых данных, строки и другие типы данных, независимо от того приходят ли они с диска, клавиатуры или из другого источника.

Форматирование вывода с помощью класса *Formatter*

Базовой частью поддержки создания форматированного вывода в языке Java служит класс *Formatter*, включенный в пакет *java.util*. Он обеспечивает *преобразования формата* (format conversions) позволяющие выводить числа, строки и время и даты практически в любом понравившемся вам формате. Класс функционирует подобно функции *printf* () языков C/C++, а значит, если Вы знакомы с этими языками, научиться использовать класс *Formatter* будет очень легко. Кроме того, это позволит в будущем упростить преобразование кода на языках C/C++ в исходный код на языке Java. Если Вы не знаете языков C/C++, все равно очень легко освоить форматирование данных.
Кроме класса Formatter в Java 2 версии 5.0 добавлен метод printf () для классов printstream и *printwriter*. Метод *printf* () автоматически использует класс Formatter и предлагает функциональные средства почти один к одному совпадающие с возможностями функции *printf* () языков C/C++, таким образом, облегчая в дальнейшем преобразование кода на C/C++ в код на языке Java.

Замечание

Несмотря на то, что класс `Formatter` и метод `printf()` языка `Java` действуют аналогично функции `printf()` языков `C/C++`, у них есть некоторые отличия и новые свойства. Поэтому, даже если Вы знакомы с языками `C/C++`, советую все же внимательно прочесть эту главу.

Конструкторы класса *Formatter*

Прежде чем Вы сможете использовать класс *Formatter* для форматирования
вывода, Вам придется создать объект типа Formatter. В общем, класс *Formatter* преобразует двоичную форму представления данных, используемых программой, в форматированный текст. Он сохраняет форматированный текст в буфере, содержимое которого Ваша программа может получить в любой нужный момент. Можно предоставить классу *Formatter* автоматическую поддержку этого буфера, либо явно задать его, когда создается объект класса *Formatter*. Существует возможность сохранения буфера класса *Formatter* в файле.
В классе Formatter определено много конструкторов, которые позволяют создавать объекты этого класса различными способами. Далее приведены некоторые их образцы.

```
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(OutStream outStrm)
```

В приведенных образцах *buf* задает буфер для форматированного вывода. Если параметр *buf* равен *null*, класс Formatter автоматически размещает объект типа StringBuilder для хранения форматированного вывода. Параметр *loc* определяет региональные и языковые настройки. Если никаких настроек не задано, используются настройки по умолчанию. Параметр *filename* задаст имя файла, который получит форматированный вывод. Параметр *charset* определяет кодировку. Если она не задана, используется кодировка, установленная по умолчанию. Параметр *outF* передаст ссылку на открытый файл, в котором будет храниться форматированный вывод. В параметре *outStrm* передается ссылка на поток вывода, который будет получать отформатированные данные. Если используется файл, выходные данные также записываются в файл.

Возможно, наиболее широко используемый конструктор, первый в приведенном перечне, — это конструктор без параметров. Он использует региональные настройки, принятые по умолчанию, и автоматически размещает объект класса *StringBuilder* для хранения форматированного вывода.

Методы класса *Formatter*

В классе Formatter определены методы, перечисленные в табл. 9.1

Метод	Описание
void close() void flush()	Закрывает вызывающий объект класса Formatter, Это приводит к освобождению ресурсов, используемых объектом, После закрытия объекта типа Formatter, он не может повторно использоваться. Попытка использовать закрытый объект приводит к генерации исключения типа FormatterClosedException Переносит информацию из буфера форматирования. Это вызывает запись в указанное место выходных данных, находящихся в буфере. Чаще всего используется объектом класса Formatter, связанным с файлом
Formatter format(String *fmtString,* Object...*args)*	Форматирует аргументы, переданные в аргументе переменной длины (vararg) *args,* в соответствии со спецификаторами формата, содержащимися в *fmtString.* Возвращает вызывающий объект
Formatter format(Locale loc, String *fmtString,* Object...*args)*	Форматирует аргументы, переданные в аргументе переменной длины args, в соответствии со спецификаторами формата, содержащимися в fmtString. При форматировании используются региональные установки, заданные в *loc.* Возвращает вызывающий объект
IOException ioException()	Если объект, приемник отформатированного вывода, генерирует исключение типа IOException, возвращает это исключение. 8 противном случае возвращает null
Locale locale()	Возвращает региональные установки вызывающего объекта
Appendable out()	Возвращает ссылку на базовый объект-приемник для выходных данных
String toString()	Возвращает объект типа String, содержащий отформатированный вывод

Основы форматирования

После создания объекта класса *Formatter* Вы можете его применить для формирования форматированной строки. Для этого используйте метод *format* (). В следующей строке приведена самая общеупотребительная его версия

```
Formatter format(String fmtString, Object...args)
```

Параметр *fmtString* состоит из элементов двух типов. Первый тип формируется из символов, которые просто копируются в буфер вывода. Второй тип содержит *спецификаторы формата* (*format specifiers*), определяющие способ отображения последующих аргументов.
Простейший спецификатор начинается со знака процента, за которым следует преобразующий спецификатор формата (*format conversion specifier*). Все преобразующие спецификаторы формата состоят из одиночных символов.

Например, спецификатор формата для данных с плавающей точкой — *%f*. Как правило, спецификаторов формата должно быть столько же, сколько аргументов первого типа, и соответствие спецификаторов формата аргументам первого типа устанавливается в направлении слева направо. Рассмотрим приведенный далее фрагмент:

```
Formatter fmt = new Formatter ();
fmt.format("Formatting %s is easy %d %£",  "with Java", 10, 98.6);
```

в нем создается объект класса Formatter, содержащий следующую строку:

```
Formatting with Java is easy 10 98.600000
```

В данном примере спецификаторы формата: *%s*, *%d* и *%f*, замещаются аргументами, следующими далее в строке форматирования. Таким образом, спецификатор *%s* заменен строкой "*with Java*", спецификатор *%d* — числом *10*, а спецификатор *%f* — числом 98.6. Все остальные символы используются без изменения. Как Вы могли догадаться, спецификатор *%s* задает строку, а спецификатор *%<a* — целое число. Как уже упоминалось, спецификатор *%f* задает значение с плавающей точкой.

Метод *format()* принимает множество разных спецификаторов формата, перечисленных в табл. 9.2. Обратите внимание на то, что у многих из них есть две формы представления: в верхнем и нижнем регистрах. Когда используется спецификатор, набранный в верхнем регистре, отображаются заглавные буквы. В остальном спецификаторы, набранные в верхнем и нижнем регистрах, выполняют одни те же преобразования, Важно знать, что язык Java контролирует соответствие каждого спецификатора формата типу связанного с ним аргумента. Если это соответствие нарушено, генерируется исключение типа *IllegalFormatException*.

После форматирования строки Вы можете получить ее, вызвав метод *toString*(). Если продолжить предыдущий пример, оператор, приведенный в следующей строке, получает отформатированную строку, содержащуюся в объекте *fmt*:

```
String str = fmt.toString();
```

Конечно, если Вы хотите просто вывести на экран отформатированную строку, не нужно ее сначала присваивать объекту типа *String*. Когда объект класса *Formatter* передается, например, в метод *println*(), метод *toString*() вызывается автоматически.

Таблица 9.2. Спецификаторы формата

Спецификатор формата	**Выполняемое преобразование**
%a %A	Шестнадцатеричное значение с плавающей точкой
%B %B	Логическое (булево) значение аргумента
%c %C	Символьное представление аргумента
%d	Десятичное целое значение аргумента
%h %H	Хэш-код аргумента
%e %E	Экспоненциальное представление аргумента
%f	Десятичное значение с плавающей точкой
%g	Выбирает более короткое представление из двух: %e или %f

%G	
%o	Восьмеричное целое значение аргумента
%n	Вставка символа новой строки
%s %S	Строковое представление аргумента
%t %T	Время и дата
%x	Шестнадцатеричное целое значение аргумента
%%	Вставка знака %

В листинге 9.1 приведена короткая программа, в которой собраны все фрагменты, демонстрирующие создание и вывод на экран отформатированной строки.

Листинг 9.1. Простой пример использования класса Formatter

```
import java.util.*;

class FormatDemo {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();

    fmt.format("Formatting %s is easy %d %f", "with Java", 10, 98.6);

    System.out.println(fmt);
  }
}
```

Одно замечание: Вы можете получить ссылку на основной буфер вывода, вызвав метод *out* (). Он возвращает объект типа *Appendabie*. Интерфейс *Appendable* включен в версию Java 2 5.0.

Теперь Вы познакомились с основным механизмом, применяемым для создания форматированной строки, в оставшейся части этого раздела будет подробно обсуждаться каждый вид преобразования, а также другие параметры форматирования, такие как выравнивание, минимальная ширина поля и точность представления данных.

Форматирование строк и символов

Для форматирования отдельного символа используйте спецификатор *%c*. Он обеспечивает вывод соответствующего символьного аргумента без преобразования. Для форматирования строки воспользуйтесь спецификатором %s.

Форматирование чисел

Для представления целого числа в десятичном формате используйте спецификатор *%d*. Для представления десятичного значения с плавающей точкой воспользуйтесь спецификатором *%f*. Для экспоненциального представления значения с плавающей точкой применяйте спецификатор *%e*. Общий вид экспоненциального представления чисел приведен в следующей строке: *x.dddddde+/-yy*

Спецификатор формата *%g* заставляет объект класса *Formatter* использовать один из спецификаторов *%f* или *%e*, предлагающий более короткое представление аргумента. В листинге 9.2 демонстрируется эффект применения спецификатора формата *%g*.

Листинг 9.2. 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);
    }

  }
}
```

Далее приведен вывод результатов работы программы из листинга 9.2:

```
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 выводить целые значения в восьмеричном и шестнадцатеричном представлениях с помощью применения спецификаторов *%o* и *%x* соответственно. Например, следующая строка:

```
fmt.format("Hex: %x, Octal:  %o",  196 196);
```

формирует следующий вывод:

```
Hex: c4, Octal:  304
```

Вы можете выводить значения с плавающей точкой в шестнадцатеричном представлении, используя спецификатор формата *%a*. Формат, создаваемый этим спецификатором, на первый взгляд может показаться странным. Дело в том, что в нем применяется форма представления числа, подобная экспоненциальной и состоящая из шестнадцатеричных мантиссы и порядка. В следующей строке приведен общий вид отформатированного числа:

```
0x1.sigpexp
```

в этой записи параметр *sig* содержит дробную часть мантиссы, а параметр *exp* — порядок числа. Символ *p* отмечает начало порядка. Например, следующий вызов:

```
fmt.format (“%a”,  123.123);
```

формирует приведенный в следующей строке вывод:

```
0x1.ec7df3b645aldp6
```

Форматирование времени и даты

Один из наиболее мощных спецификаторов преобразования формата — спецификатор *%t*. Он позволяет форматировать информацию о времени и дате. Этот спецификатор действует несколько иначе, чем другие, поскольку он требует применения *суффикса* для описания отображаемой части и точного формата выбранных времени и даты. Суффиксы перечислены в табл. 9.3. Например, для отображения минут, Вы могли бы использовать спецификатор *%tM*, в котором суффикс M показывает минуты в двух символьном поле. Аргумент, относящийся к спецификатору *%t*, должен быть типа ***Calendar, Date, Long*** или ***long***.

Таблица 9.3. Суффиксы формата для вывода времени и даты

Суффикс	Вывод, замещающий суффикс
a	Сокращенное название дня недели
A	Полное название дня недели
b	Сокращенное название месяца
B	Полное название месяца
c	Стандартное представление в виде: день, месяц чч:мм:сс часовой пояс, год
C	Первые две цифры года
d	День месяца как десятичное целое (от 01 до 31)
D	месяц/день/год
e	День месяца как десятичное целое (от 1 до 31)
F	год-месяц-день
h	Сокращенное название месяца
H	Час (от 00 до 23)
I	Час (от 1 до 12)
j	Номер дня в году как десятичное целое (от 001 до 366)
k	Час (от 0 до 23)
l	Час (от 1 до 12)
L	Миллисекунды (от 000 до 999)
m	Десятичный номер месяца (от 01 до 13)
M	Минуты как десятичное целое (от 00 до 59)
N	Наносекунды (от 000000000 до 999999999)
P	Региональный эквивалент AM или PM заглавными буквами
p	Региональный эквивалент AM или PM строчными буквами
Q	Миллисекунды от 1/1/1970
r	чч:мм (12-часовой формат)
R	чч:мм (24-часовой формат)
S	Секунды (от 00 до 60)

В листинге 9.3 продемонстрировано применение нескольких форматов.

Листинг 9.3. Форматирование времени и даты

```
import java.util.*;

class TimeDateFormat {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();
    Calendar cal = Calendar.getInstance();

    // Display standard 12-hour time format.
    fmt.format("%tr", cal);
    System.out.println(fmt);
```

```
    // Display complete time and date information.
    fmt = new Formatter();
    fmt.format("%tc", cal);
    System.out.println(fmt);

    // Display just hour and minute.
    fmt = new Formatter();
    fmt.format("%tl:%tM", cal, cal);
    System.out.println(fmt);

    // Display month by name and number.
    fmt = new Formatter();
    fmt.format("%tB %tb %tm", cal, cal, cal);
    System.out.println(fmt);
  }
}
```

Далее приведен образец вывода программы из листинга 9.3:

```
09:17:15 AM
Sun Dec 12 09:17:15 CST 2004
9:17
December Dec 12
```

Спецификаторы *%n* и *%%*

Спецификаторы *%n* и *%%* отличаются от других спецификаторов формата тем, что они не связаны с аргументами. Они представляют собой *ESC*-последовательности, вставляющие символ в поток вывода. Спецификатор *%n* вставляет символ перехода на новую строку, а спецификатор *%%* — знак процента. Ни один из этих символов нельзя вставить непосредственно в строку формата. Конечно, Вы также можете использовать стандартную *ESC*-последовательность *\n* для внедрения символа перехода на новую строку. В листинге 9.4 показано применение спецификаторов *%n* и *%%*.

Листинг 9.4. Демонстрация применения спецификаторов %n и %%.

```
import java.util.*;

class FormatDemo3 {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();

    fmt.format("Copying file%nTransfer is %d%% complete", 88);
    System.out.println(fmt);
  }
}
```

Далее приведен вывод результатов работы программы из листинга 9.4:

```
Copying file
Transfer is 88% complete
```

Задание минимальной ширины поля

Целое число, стоящее между знаком *%* и символом преобразования формата в спецификаторе, действует как *спецификатор минимальной ширины поля* (*minimum field width specifier*). Он заполняет вывод пробелами для того, чтобы аргумент достиг заданной минимальной длины. Если строка или число

длиннее заданного минимума, они все равно будут отображены полностью. По умолчанию заполнение выполняется пробелами. Если Вы хотите заполнить поле нулями, поместите 0 перед спецификатором ширины поля. Например, спецификатор *%05d* дополнит слева число, состоящее меньше чем из пяти цифр, нулями так, чтобы его общая длина равнялась пяти цифрам. Спецификатор ширины поля может применяться со всеми спецификаторами формата за исключением спецификатора *%n*.

В листинге 9.5 показано применение спецификатора минимальной ширины поля, добавленного к спецификатору формата *%f*.

Листинг 9.5. Демонстрация применения спецификатора минимальной 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);

  }
}
```

Далее приведен вывод результатов работы программы из листинга 9.5:

```
|10.123450|
|  10/123450|
|00010.123450|
```

В первой строке показана текущая ширина поля для вывода числа 10.123450. Во второй строке отображается это же число в поле с 12-символьной шириной. В третьей строке выведено то же число в поле шириной 12 символов и дополнено ведущими нулями для заполнения ширины поля.

Спецификатор минимальной длины поля часто используется для выравнивания колонок при формировании таблиц. В листинге 9.6 создается таблица квадратов и кубов для чисел от 1 до 10.

Листинг 9.6. Создание таблицы 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);
    }

  }
```

```
}
```

Далее приведен вывод таблицы, созданной программой, из листинга 9.6:

```
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
```

Задание точности представления

Спецификатор точности (precision) можно применять в спецификаторах формата: *%f*, *%e*, *%g* и *%s*. Он вставляется после спецификатора минимальной ширины поля (если таковой есть) и состоит из точки, за которой следует целое значение. Его точный смысл зависит от типа данных, для которых он определяется.

Если Вы задаете, спецификатор точности для данных с плавающей точкой, используя, спецификаторы формата: *%f*, *%e*, или *%g*, он задает количество выводимых десятичных знаков. Например, спецификатор *%10.4f* выводит число с минимальной шириной поля *10* символов и с четырьмя десятичными знаками. Принятая по умолчанию точность равна шести десятичным знакам.

Примененный к строкам спецификатор точности задает максимальную длину поля вывода. Например, спецификатор *%5.7s* выводит строку длиной не менее пяти и не более семи символов. Если строка длиннее, конечные символы отбрасываются.

В листинге 9.7 иллюстрируется применение спецификатора точности.

Листинг 9.7. Демонстрация применения спецификатора точности

```
import java.util.*;

class PrecisionDemo {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();

    // Format 4 decimal places.
    fmt.format("%.4f", 123.1234567);
    System.out.println(fmt);

    // Format to 2 decimal places in a 16 character field.
    fmt = new Formatter();
    fmt.format("%16.2e", 123.1234567);
    System.out.println(fmt);

    // Display at most 15 characters in a String.
    fmt = new Formatter();
    fmt.format("%.15s", "Formatting with Java is now easy.");
    System.out.println(fmt);
  }
}
```

Далее приведен вывод результатов работы программы из листинга 9.7:

```
123.1235
```

```
1.23e+02
Formatting with
```

Применение флагов форматирования

Класс Formatter распознает набор *флагов* (*flags*) *форматирования*. Которые позволят Вам управлять различными аспектами преобразования. Все флаги форматирования (табл. 9.4) — это одиночные символы, следующие за знаком % в спецификаторе формата.

Таблица 9.4. Флаги форматирования

Флаг	Эффект
-	Выравнивание влево
	Дополнительный преобразующий формат
	Вывод заполняется нулями вместо пробелов
пробел (space)	Положительному значению при **выводе** предшествует пробел
+	Положительному значению при выводе предшествует знак +

Не все флаги применяются со всеми спецификаторами формата. В следующих разделах дано подробное объяснение действия каждого флага.

Выравнивание вывода

По умолчанию весь вывод выравнивается вправо. *Это* означает, что если ширина поля больше отображаемых данных, то данные будут выводиться, прижатыми к правому краю поля. Вы можете вывод выровнять влево. Поместив знак минуса сразу после символа % в спецификаторе формата. Например, спецификатор *%-10.2f* выводит в поле шириной *10* символов число с плавающей точкой, выровненное влево и содержащее *2* десятичных знака. В листинге 9.8 приведена программа, задающая выравнивание влево для вывода.

Листинг 9.8. Демонстрация выравнивания влево

```
import java.util.*;

class LeftJustify {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();

    // Right justify by default
    fmt.format("|%10.2f|", 123.123);
    System.out.println(fmt);

    // Now, left justify.
    fmt = new Formatter();
    fmt.format("|%-10.2f|", 123.123);
    System.out.println(fmt);
  }
}
```

Далее приведен вывод результатов работы программы из листинга 9.8:

```
|    123.12|
```

```
|123-12        |
```

Как видите, вторая строка выровнена влево в поле шириной 10 символов.

Флаги *Space*, +, 0 и *(*

Если нужно вывести знак + перед положительными числовыми значениями, добавьте флаг +. Например, строка:

```
fmt.format C%+d",  100);
```

создаст следующую строку:

```
+100
```

При формировании колонок чисел иногда полезно вставить пробел перед положительными значениями для того, чтобы выровнять положительные и отрицательные числа. Для этого добавьте флаг *пробел*. В листинге 9.9 приведен пример применения этого флага.

Листинг 9.9. Демонстрация применения

```
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);
  }
}
```

Далее приведен вывод программы из листинга 9.9:

```
-100
100
-200
200
```

Обратите внимание, у положительных значений есть ведущий пробел, позволяющий выровнять колонку соответствующим образом.

Для того чтобы показать отрицательные значения в скобках, а не со знаком — используйте флаг (. Например, вызов метода:

```
fmt. format ("% (d\  -100)
```

создаст следующую строку:

(100)

Флаг) вызывает заполнение вывода нулями вместо пробелов. Этот флаг может использоваться во всех спецификаторах формата за исключением спецификатора *%n*.

Флаг запятая

При выводе больших чисел часто могут оказаться полезными разделители групп, в английском языке для этого используются запятые. Например, значение *1234567* гораздо легче читать, если оно отформатировано следующим образом: *1,234,567*. Для того чтобы добавить подобные разделители используйте флаг , (запятая). Код, приведенный в следующей строке:

```
fmt.format("%,.2f",  4356783497.34)
```

создаст строку, приведенную далее:

```
4,356,783,497.34
```

Флаг

Флаг # может применяться в спецификаторах: *%o*, *%x*, *%e* и *%f*. В спецификаторах *%e* и *%f* флаг # гарантирует наличие десятичной точки в выводимом числе, даже если у него нет десятичных знаков. Если поставить флаг # в спецификатор формата *%x*, шестнадцатеричное представление числа будет выводиться с префиксом *0x*. Вставка флага # в спецификатор *%o* вызывает вывод числа с ведущими нулями.

Применение верхнего регистра

Как упоминалось ранее, у некоторых спецификаторов формата есть версии, набранные в верхнем регистре (т. е. заглавными буквами), которые вызывают использование заглавных букв для вывода аргументов, когда это возможно. В табл. 9.5 перечислены эти спецификаторы формата и вызываемый ими эффект.

Таблица 9.5. *Спецификаторы формата, набранные в верхнем регистре*

Спецификатор	Преобразование формата
%A	Вывод шестнадцатеричных цифр от a до f заглавными буквами A-F. Префикс 0x выводится как OX и символ p — как P
%B	вывод заглавными буквами значений true и false
%C	Вывод заглавной буквой соответствующего символьного аргумента
%E	Вывод символа e, обозначающего порядок числа, заглавной буквой
%G	Вывод символа e, обозначающего порядок числа, заглавной буквой
%H	Вывод шестнадцатеричных цифр от a до f заглавными буквами A— F
%S	Вывод заглавными буквами соответствующей строки
%T	Вывод всех букв в выводной строке заглавными
%X	Вывод шестнадцатеричных цифр от a до f заглавными буквами A-F. Необязательный префикс Ox выводится как Ox, если он есть

Например, вызов:

```
fmt.format("%X",  250);
```

создаст приведенную далее строку:

```
FA
```

А следующий вызов:

```
fmt. format* "%E\   123.1234);
```

создаст приведенную далее строку:

```
1.231234E+2
```

Использование порядкового номера аргумента

У класса *Formatter* есть очень полезное свойство, которое позволяет задавать аргумент, к которому следует применить конкретный спецификатор формата. Обычно соответствие между спецификаторами и аргументами, на которые они воздействуют, устанавливается в соответствии с порядком их следования, слева направо. Это означает, что первый спецификатор формата соответствует первому аргументу, второй спецификатор — второму аргументу и т. д. Однако, используя *порядковый номер* (*argument index*) или *индекс аргумента*, Вы можете указать явно, какой спецификатор формата соответствует какому аргументу.

Порядковый номер аргумента указан за знаком *%* в спецификаторе формата и имеет следующий формат:

```
   n$
```

Символ *n* обозначает порядковый номер нужного аргумента, нумерация аргументов начинается с единицы. Рассмотрим пример:

```
fmt. format (" %3$d  %l$d %2$ “, 10, 20, 30 )
```

Он формирует следующую строку вывода:

```
30 10 20
```

Листинг 9.10. Применение относительных номеров аргументов для упрощения создания пользовательского формата вывода 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);
  }
}
```

В следующей строке приведен вывод результатов работы программы из листинга 9.10:

```
Today is day 8 of may, 2004
```

Благодаря относительному индексированию аргумент cal нужно передать всего один раз вместо трех.

Применение метода *printf()* языка Java

Хотя с технической точки зрения нет ничего неправильного в непосредственном использовании класса *Formatter*, как было сделано в предыдущих примерах, при формировании вывода на консоль, версия Java 2 5.0 прелагает более удобную альтернативу: метод *printf* (). Этот метод автоматически использует объект типа *Formatter* для создания форматированной строки. Затем она выводится как строка в стандартный поток вывода по умолчанию на консоль. Метод *printf* () определен в классах *PrintStream* и *PrintWriter*.

В классе *PrintStream* у метода *printf* () две синтаксические формы записи:

```
PrintStream printf(String fmtString, Object...args)
PrintStream printf(Local loc, String fmtString, Object...args)
```

В первом варианте аргументы *&rgs* записываются в стандартный поток вывода в формате, заданном в параметре *fmtString*, используя региональные настройки (*locale*), принятые по умолчанию. Во втором варианте Вам предлагается задать региональные настройки. Оба варианта возвращают вызывающий объект типа Printstream. Поскольку *system.out* — объект типа *PrintStream*, Вы можете вызывать метол *printf* () прямо для объекта *system.out*.

Варианты метода *printf* () для класса *Printwriter* приведены в следующих строках:

```
PrintWriter printf(String fmtString, Object...args)
PrintWriter printf(Local loc, String fmtString, Object...args)
```

Они выполняются точно так же, как и в классе printstream за исключением того, что возвращают объект типа *PrintWriter*.

В листинге 9.11 приведен пример программы, использующей метод *printf*() для вывода числовых значений в различных форматах. В прошлом подобное форматирование требовало существенной работы. С появлением метода *printf* () оно превратилось в легко решаемую задачу.

Листинг 9.11.Демонстрация применения метода printf()

```
class PrintfDemo {
  public static void main(String args[]) {
    System.out.println("Here are some numeric values " +
                       "in different formats.\n");

    System.out.printf("Various Integer formats: ");
    System.out.printf("%d %(d %+d %05d\n", 3, -3, 3, 3);

    System.out.println();

    System.out.printf("Default floating-point format: %f\n",
                      1234567.123);
    System.out.printf("Floating-point with commas: %,f\n",
                      1234567.123);
```

```
    System.out.printf("Negative floating-point default: %,f\n",
                      -1234567.123);
    System.out.printf("Negative floating-point option: %,(f\n",
                      -1234567.123);

    System.out.println();

    System.out.printf("Line-up positive and negative values:\n");
    System.out.printf("% ,.2f\n% ,.2f\n",
                      1234567.123, -1234567.123);

  }
}
```

Далее приведен вывод результатов программы из листинга 9.11:

```
Here are some numeric values in defferent formats.

Various Integer formats: 3 (3) +3

Default floating-point format: 1234567.123000 Floating-point with
commas: 1,234,567.123000

Negative floating-point default: -1,234,567.123000 Negative floating-
point option: (1,234,567.123000)

Line-up positive and negative values:
1, 234,567.123000
-1, 234,567.123000
```

Класс Scanner

Класс *Scanner* — это дополнение к классу *Formatter*. Объекты класса *Scanner* читают форматированный ввод и преобразуют его в двоичное представление. Они могут использоваться для чтения данных с консоли, из файла, строки или любого другого источника, реализующего Интерфейсы *Readable* (добавленный в Java 2 версии 5.0) или *ReadabieByteChannei*. Например, можно применять класс *scanner* для чтения числа, введенного с клавиатуры и присваивания этого значения переменной. Хотя такие операции всегда были возможны, класс *scanner* существенно упрощает этот процесс. Как Вы увидите, класс *scanner*, несмотря на свои богатые функциональные возможности, очень прост в использовании.

Класс *scanner* включен в пакет *java.util*.

Конструкторы класса *Scanner*

В классе *Scanner* определены конструкторы, перечисленные в табл. 9.6. Как правило, объект типа *Scanner* может быть создан для объектов типа *String*, *InputStream* или любого другого типа, реализующего интерфейсы *Readable* или *ReadabieByteChannel*.

В приведенном далее фрагменте создается объект класса scanner, который читает файл *Test.txt*:

```
FileReader fin = new FileReader("Test.txt";
```

```
Scanner src = new Scanner(fin);
```

Этот код выполняется, поскольку класс *FileReader* реализует интерфейс *Readable*. Таким образом, вызов конструктора разрешается для *Scanner* (*Readable*).

В следующей строке создается объект класса scanner, который читает из стандартного потока ввода, по умолчанию с клавиатуры:

```
Scanner conin = new Scanner(System.in);
```

Приведенный код— действующий, так как *System.in*— объект типа *InputStream*. Следовательно, вызов конструктора отображается в *Scanner* (*input stream*) .

Таблица 9.6. *Конструкторы класса scanner*

Метод	**Описание**
`static Scanner create(File` *`from)`* `throws FileNotFoundException`	Создает объект типа scanner, который использует файл, заданный параметром *from* как источник входных данных
`static Scanner create(File` *`from,`* `String` *`charset)`*	Создает объект типа scanner, который использует как источник входных данных поток, заданный параметром from с кодировкой, заданной параметром charset
`Scanner(InputStream` *`from,`* `String charset)`	Создает обьект типа Scanner, который использует поток, заданный параметром from как источник входных данных
`Scanner(Readable from)`	Создает объект типа Scanner, который использует объект, реализующий интерфейс Readable и заданный параметром *from* как источник входных данных
`Scanner(ReadableByteChannel from)`	Создает обьект типа Scanner, который использует объект, реализующий интерфейс ReadableByteChannel и заданный параметром from как источник входных данных
`Scanner(ReadableByteChannel from, String charset)`	Создает объект типа Scanner, который использует объект, реализующий интерфейс ReadableByteChannel и заданный параметром from как источник входных данных
`Scanner(ReadableByteChannel` *`from,`* `String charset)`	Создает объект типа Scanner, который использует как источник входных данных объект, реализующий интерфейс ReadableByteChannel и заданный параметром from , с кодировкой, указанной в параметре *charset*
`Scanner(String from)`	Создает объект типа scanner, который использует строку, заданную параметром from как источник входных данных

В следующем фрагменте создается объект типа scanner, который читает данные из строки:

```
String instr = "10 99.88 scanning is easy.";
Scanner conin = new Scanner(instr);
```

Описание форматирования входных данных

Создав объект типа *Scanner*, очень просто использовать его для чтения форматированных входных данных. Как правило, объект класса Scanner читает лексемы (*tokens*) из базового источника, который Вы задали при создании объекта типа *Scanner*. Применительно к классу *Scanner*, *лексема* — это порция вводимых данных, обособленная набором разделителей, по умолчанию пробелами. При считывании лексема сопоставляется с конкретным *регулярным выражением* (*regular expression*), задающим формат ввода. Хотя класс *scanner* разрешает определять собственный тип выражения, соответствие которому будет проверяться в последующих операциях ввода, в нем содержится много предопределенных образцов, соответствующих базовым типам, таким как *int* и *double*, и строкам. Следовательно, зачастую Вам не придется формировать образец для проверки соответствия вводимых данных.

Обычно для использования класса *scanner* необходимо выполнить следующие шаги.

1. С помощью одного из методов *hasNextX* класса *Scanner* определить, доступна ли для ввода порция данных типа *x*.
2. Если да, считать ее с помощью одного из методов *nextX* класса *scanner*.
3. Повторять процесс, пока не исчерпан поток ввода.

Как показывает описанный процесс ввода, в классе scanner определены два набора методов, позволяющих читать входные данные. Первый содержит методы *hasNextX*, перечисленные в табл. 9.7. Эти методы определяют, доступны ли для ввода данные заданного типа. Например, вызов метода *hasNextint*() возвращает *true*, только если следующая лексема, предназначенная для считывания, — целое число. Если данные указанного типа доступны, они считываются с помощью одного из методов *nextX* Класса *Scanner*, перечисленных в табл. 9.8, Например, для считывания очередного целого числа вызовите метод *NextInt*(). В приведенном далее фрагменте показано, как прочесть последовательность целых значений с клавиатуры:

```
Scanner conin = new Scanner(System.in);
int i;

// Считывает последовательность целых чисел
whilefconin.hasMextlnt()) {
  i = conin.nextlnt();
}
```

Таблица 9.7. *Методы hasNext класса Scanner*

Метод	**Описание**
boolean hasNext()	Возвращает true, если доступна для чтения лексема какого-либо типа. Возвращает false в противном случае
boolean hasNext(Pattern pattern)	Возвращает true, если лексема, соответствующая образцу, переданному в параметре pattern, доступна для чтения. Возвращает false в противном случае
boolean hasNext(String pattern)	Возвращает true, если лексема, соответствующая образцу, переданному в параметре pattern, доступна для чтения. Возвращает false в противном случае
boolean hasNextBigDecimal()	Возвращает true, если значение, которое можно сохранить в объекте типа Big-Decimal, доступно

	для считывания, Возвращает false в противном случае
boolean hasNextBiglnteger()	Возвращает true, если значение, которое можно сохранить в объекте типа Biglnteger, доступно для считывания. Возвращает false в противном случае. Использует основание системы счисления, принятое по умолчанию (равно 10, если не менялось)
boolean hasNextBiglnteger (int *radix)*	Возвращает true, если значение, которое можно сохранить в объекте типа Biglnteger, доступно для считывания. Возвращает false в противном случае
boolean hasNextBoolean()	Возвращает true, если значение типа boolean доступно для считывания. Возвращает false в противном случае
boolean hasNextByte ()	Возвращает true, если значение типа byte доступно для считывания, Возвращает false в противном случае. Использует основание системы счисления, принятое по умолчанию (равно 10, если не менялось)
boolean hasNextByte(int radix)	Возвращает true, если значение типа byte доступно для считывания. Возвращает false в противном случае
boolean hasNextDouble()	Возвращает true, если значение типа double доступно для считывания. Возвращает false в противном случае
boolean hasNextFloat()	Возвращает true, если значение типа float доступно для считывания. Возвращает false в противном случае
boolean hasNextlnt()	Возвращает true, если значение типа int доступно для считывания. Возвращает false в противном случае. Использует основание системы счисления, принятое по умолчанию (равно 10, если не менялось)
boolean hasNextlnt(int radix)	Возвращает true, если значение типа int доступно для считывания. Возвращает false в противном случае
boolean hasNextLong()	Возвращает true, если значение типа long доступно для считывания. Возвращает false в противном случае. Использует основание системы счисления, принятое по умолчанию (равно 10, если не менялось)
boolean hasNextLong(int radix)	Возвращает true, если значение типа long доступно для считывания. Возвращает false в противном случае
boolean hasNextShort()	Возвращает true, если значение типа short доступно для считывания. Возвращает false в противном случае. Использует основание системы счисления, принятое по умолчанию (равно 10, если не менялось)
boolean hasNextShort(int radix)	Возвращает true, если значение типа short доступно для считывания. Возвращает false в

	противном случае

Цикл *while* прекращается, как только очередная лексема — нецелочисленная. Следовательно, цикл прерывает чтение целых чисел, как только обнаруживает нецелое число в потоке ввода.

Если метод next не может найти данные того типа, которой он ищет, он генерирует исключение типа *NoSuchElementException*. Следовательно, лучше убедиться в том, что данные искомого типа доступны для считывания с помощью метода *hasNext* прежде, чем вызывать соответствующий метод *next*.

Несколько примеров применения класса *Scanner*

Включение класса *Scanner* в язык Java превратило прежде утомительное и скучное занятие в легкое и приятное. Для того чтобы понять, почему это произошло, рассмотрим несколько примеров. В листинге 9.12 вычисляется среднее арифметическое последовательности целых значений, вводимых с клавиатуры.

Листинг 9.12. Использование класса Scanner для вычисления среднего арифметического

```
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("Enter numbers to average.");

    // Read and sum numbers.
    while(conin.hasNext()) {
      if(conin.hasNextDouble()) {
        sum += conin.nextDouble();
        count++;
      }
      else {
        String str = conin.next();
        if(str.equals("done")) break;
        else {
          System.out.println("Data format error.");
          return;
        }
      }
    }

    System.out.println("Average is " + sum / count);
  }
}
```

В листинге 9.12 числа считываются с клавиатуры и суммируются до тех пор, пока пользователь не введет строку "*done*". Эта строка прекращает ввод и на экран выводится среднее арифметическое введенных чисел. Далее приведен образец вывода программы из листинга 9.12:

```
Enter numbers to average.
1.2
2
3.4
4
done
Average is 2.65
```

В программе из листинга 9.12 считываются числа, пока не обнаружена лексема, которая не соответствует корректному значению типа *double*. Когда это происходит, программа проверяет, не равна ли эта лексема строке "*done*". Если это так, программа завершается корректно. В противном случае выводится сообщение об ошибке.

Обратите внимание на то, что числа читаются с помощью вызова метода *nextDouble*(). Этот метод считывает любое число, которое может быть преобразовано в тип *double*, включая целые значения, такие как 2, и значения с плавающей точкой, такие как 3.4. Таким образом, у числа, считываемого методом *nextDouble*(), необязательно должна быть десятичная точка. Этот базовый принцип положен в основу всех методов *next*. Они могут находить соответствие и читать данные любого формата, который способен представлять данные требуемого типа.

Приятная особенность, свойственная классу *Scanner*, заключается в том, что способ, применяемый для считывания данных из одного источника, используется для чтения и из другого источника. В листинге 9.13 приведен пример, в котором подсчитывается среднее арифметическое последовательности чисел, содержащихся в текстовом файле.

Листинг 9.13. Применение класса Scanner для расчета среднего арифметического чисел, считываемых 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;

    // Write output to a file.
    FileWriter fout = new FileWriter("test.txt");
    fout.write("2 3.4 5 6 7.4 9.1 10.5 done");
    fout.close();

    FileReader fin = new FileReader("Test.txt");

    Scanner src = new Scanner(fin);

    // Read and sum numbers.
    while(src.hasNext()) {
      if(src.hasNextDouble()) {
        sum += src.nextDouble();
        count++;
      }
      else {
        String str = src.next();
```

```
        if(str.equals("done")) break;
        else {
          System.out.println("File format error.");
          return;
        }
      }
    }

    fin.close();
    System.out.println("Average is " + sum / count);
  }
}
```

В следующей строке приведен вывод программы из листинга 9.13:

```
Average is 6.2
```

Вы можете использовать класс *Scanner* для чтения данных разных типов, даже если порядок их следования заранее неизвестен. У вас есть возможность проверить тип данных, доступных для чтения, перед их считыванием. Рассмотрим программу, приведенную в листинге 9.14.

Листинг 9.14. Применение класса Scanner для считывания данных разных типов, хранящихся в файле

```
import java.util.*;
import java.io.*;

class ScanMixed {
  public static void main(String args[])
    throws IOException {

    int i;
    double d;
    boolean b;
    String str;

    // Write output to a file.
    FileWriter fout = new FileWriter("test.txt");
    fout.write("Testing Scanner 10 12.2 one true two false");
    fout.close();

    FileReader fin = new FileReader("Test.txt");

    Scanner src = new Scanner(fin);


    // Read to end.
    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();
  }
}
```

Далее приведен вывод результатов работы программы из листинга 9.14:

```
String: Testing
String: Scanner
int: 10
double: 12.2
String: one
boolean: true
String: two
boolean: false
```

При считывании данных разных типов, как в листинге 9.14, необходимо внимательно следить за порядком вызова методов *next*. Например, если в цикле поменять порядок вызова методов *nextInt*() и *nextDouble*(), оба числа будут считаны как значения типа *double*, поскольку метод *nextDouble*() приводит в соответствие любую строку, которая может быть представлена как тип *double*.

Установка разделителей

Объект класса *scanner* находит начало и конец лексемы, основываясь на наборе *разделителей* (*delimiters*). По умолчанию в качестве разделителей применяются пробелы, и именно они использовались в предыдущих примерах. Однако можно изменить разделители, вызвав метод *useDelimiter*(), формы синтаксической записи которого приведены в следующих строках:

```
Scanner useDelimiter(String pattern)
Scanner useDelimiter(Pattern pattern)
```

В этой записи параметр pattern — это регулярное выражение, определяющее набор разделителей.

В листинге 9.15 приведена новая версия программы из предыдущих листингов, которая считывает последовательность чисел, разделенных запятыми и любым количеством пробелов.

Листинг 9.15. Применение класса scanner для вычисления среднего арифметического последовательности значений, разделенных запятыми

```
import java.util.*;
import java.io.*;
```

```
class SetDelimiters {
  public static void main(String args[])
    throws IOException {

    int count = 0;
    double sum = 0.0;

    // Write output to a file.
    FileWriter fout = new FileWriter("test.txt");

    // Now, store values in comma-separated list.
    fout.write("2, 3.4,    5,6, 7.4, 9.1, 10.5, done");
    fout.close();

    FileReader fin = new FileReader("Test.txt");

    Scanner src = new Scanner(fin);

    // Set delimiters to space and comma.
    src.useDelimiter(", *");

    // Read and sum numbers.
    while(src.hasNext()) {
      if(src.hasNextDouble()) {
        sum += src.nextDouble();
        count++;
      }
      else {
        String str = src.next();
        if(str.equals("done")) break;
        else {
          System.out.println("File format error.");
          return;
        }
      }
    }
    fin.close();
    System.out.println("Average is " + sum / count);
  }
}
```

В листинге 9.15 числа, записанные в файл *test.txt*, разделены запятыми и пробелами. Использование образца для разделителей ", * " сообщает объекту класса *Scanner* о том, что запятую и 0 или более пробелов следует интерпретировать как разделитель. Вывод у версии программы из листинга 9.15 такой же, как и в предыдущих примерах.

Вы можете получить текущий образец разделителя, вызвав метод *delimiter*(), синтаксическая запись которого приведена в следующей строке:

```
Pattern delimiter()
```

Другие свойства класса *Scanner*

В классе *Scanner* определены и другие методы в дополнение к уже рассмотренным. Один из них, очень полезный в некоторых ситуациях, — *findInLine*(). Два варианта его синтаксической записи приведены в следующих строках:

```
String findInLine(Pattern pattern)
String findInLine(String pattern)
```

Этот метол ищет заданный образец в очередной строке текста. Если образец найден, соответствующая ему лексема извлекается из строки ввода и возвращается. В противном случае возвращается *null*. Этот метод действует независимо от установленного набора разделителей, Он полезен, когда нужно определить местоположение конкретного образца (подстроки в строке). В листинге 9.16 определяется местоположение поля *Age* во входной строке и затем выводится возраст.

Листинг 9.16. Демонстрация применения метода findInLine()

```
import java.util.*;

class FindInLineDemo {
  public static void main(String args[]) {
    String instr = "Name: Tom Age: 28 ID: 77";

    Scanner conin = new Scanner(instr);

    // Find and display age.
    conin.findInLine("Age:"); // find Age

    if(conin.hasNext())
      System.out.println(conin.next());
    else
      System.out.println("Error!");

  }
}
```

Вывод программы из листинга 9.16 — 28. Метод *findInLine*() применяется в программе для поиска подстроки "*Age*". После того как заданная подстрока найдена, считывается следующая за ней лексема, а она представляет возраст.

Варианты синтаксической записи родственного методу *findInLine*() метода *findWithinHorizon*() приведены в следующих строках:

```
String findWithinHorizon(Pattern pattern, int count)
String findWithinHorizon(String pattern, int count)
```

Этот метод пытается найти заданный образец в ближайших *count* символах. Если поиск удачен, метод возвращает найденный образец, в противном случае он возвращает *null*. Если количество символов count равно 0, поиск ведется во всей строке ввода пока не найдена подстрока, соответствующая образцу, или не обнаружен конец строки ввода.

Вы можете пропустить или обойти образец с помощью метода *skip*(), варианты синтаксической записи которого приведены в следующих строках:

```
Scanner skip(Pattern pattern)
Scanner skip(String pattern)
```

Если в строке ввода найдена подстрока, соответствующая образцу *pattern*, метод *skip()* просто перемещается за нее в строке ввода и возвращает ссылку на вызывающий объект. Если подстрока не найдена, метод *skip ()* генерирует исключение типа *NoSuchElementException*.

Кроме того, в класс Scanner включены методы: `radix()`, возвращающий текущее основание системы счисления, используемое классом *Scanner*, `useRadix()`, устанавливающий основание системы счисления, и метод *close()*, закрывающий объект класса scanner.

Глава 10

Изменения в API

Главное содержание этой книги — добавленные в язык Java новые функциональные возможности, которым были посвящены предыдущие девять глав. Но версия Java 2 5.0 вносит много изменений и в API языка. Как и во всех предыдущих обновлениях языка Java, в существующую библиотеку внесено много мелких, включающих незначительные улучшения корректировок и подчисток, но есть несколько значительных изменений, влияющих на всех программистов, пишущих на языке Java. Они включают как добавление важных новых средств, так и основательное обновление существующих. Именно эти крупные изменения и станут предметом обсуждения в этой главе.

К ним следует отнести следующие изменения:

- модификацию подсистемы Collections Framework (и других частей API) для применения настраиваемых типов;
- включение новых классов и интерфейсов в пакет java.iang;
- добавление методов побитной обработки для классов Integer и Long;
- поддержку 32-битных кодов символов Unicode в классах String и Character;
- добавление новых подпакетов в пакет java.iang;
- добавление новых подпакетов в пакет Java.util.

Благодаря усилиям команды разработчиков языка Java эти обширные добавления и обновления были выполнены так, что позволили сохранить работоспособность уже существующих программ и расширить функциональные возможности API. Обзору перечисленных изменений посвящена оставшаяся часть этой главы.

Возможность применения настраиваемых типов при работе с коллекциями

Наиболее значительное изменение *API* кроется в классах, интерфейсах и методах, формирующих подсистему *Collections Framework*. Каждый из них был полностью перестроен и модифицирован для работы с параметризованным типом, вместо ссылок на тип *Object*. Превращение *Collections Framework* в подсистему настраиваемых типов стало важнейшим достижением, потому что все операции с коллекциями теперь обладают типовой безопасностью. Настраиваемые типы избавили от необходимости обратного преобразования в надлежащий тип элемента, содержащегося в коллекции, при его извлечении.

В общем, все классы и интерфейсы коллекций получили параметры типа, которые описывают тип элемента, хранящегося и обрабатываемого в коллекции. Далее приведен перечень новых объявлений интерфейсов коллекций:

```
interface  Collection<E>                    interface Comparator<T>
```

```
interface  Iterator<E>                    interface ListIterator<E>
interface  List<E>                        interface Map<K, V>
interface  Queue<E>                       interface Set<E>
interface  SortedMap<K, V>                interface SortedSet<E>
```

В приведенном перечне интерфейс *Queue* добавлен в версии Java 2 5.0. Далее приведен перечень новых объявлений классов:

```
abstract class AbstractCollection<E>

abstract class AbstractList<E>

abstract class AbstractMap<K, V>

abstract class AbstractSeguentialList<E>

class ArrayList<E>

class SnuinMap<K extends Enum<K>, V>

class HashMap<K, V>

class HashTable<K, V>

class LinkedHashMap<K, V>

class LinkedList<E>

class Stack<E>

class TreeSet<E>

class WeakHashMap<K, V>

abstract class AbstractQueue<E>

abstract class AbstractSet<E>

class Collections

class EnumSet<E extends enum<E>>

classHashSet<E>

class IdentifyHashMap<K, V>

class LinkedHashSet<E>

class PriorityQueue<E>

class TreeMap<k, V>

class Vector<E>
```

В приведенном списке классы *EnumMap*, *EnumSet*, *AbstractQueue* и *PriorityQueue* добавлены в версии Java 2 5.0.

Теперь, когда *Collections Framework* превратилась в подсистему настраиваемых типов, при создании коллекции Вы задаете тип данных, которые будут в ней храниться. Например:

```
ArrayList<Integer> list = new Arraylist<Integer>();
```

объявляет переменную *list* как ссылку на объект класса *ArrayList*, содержащий объекты класса Integer.

Поскольку интерфейс *Iterator* теперь тоже настраиваемый тип, Вы должны задавать как параметр тип данных, для которых создается итератор. Например, если иметь в виду приведенное объявление переменной *list*, нужно

```
Iterator<Integer> itr =list.iterator();
```

Переменная *list* содержит объекты типа *Integer*, поэтому переменная *itr* должна быть объявлена как итератор для объектов типа *Integer*.

Обновление класса *Collections*

Если Вы посмотрите на приведенный перечень классов коллекций, то увидите, что у класса *Collections* нет параметра типа. Причина заключается в том, что класс *Collections* целиком состоит из статических методов, которые реализуют алгоритмы обработки коллекций. Следовательно, нет смысла делать этот класс настраиваемым. Однако его методы модифицированы для применения настраиваемых типов благодаря передаче в метод и возврату из метода настраиваемых типов.

В класс *collections* также включено несколько новых методов. Возможно самый важный среди них набор перегруженных методов *checked*, таких как *checkedCollection*(), который возвращает представление коллекции, названное в документации "представлением времени выполнения, обеспечивающим типовую безопасность". Это означает ссылку на коллекцию, в которой динамически во время выполнения проверяется типовое соответствие каждого включаемого в нее элемента. Попытка вставить в коллекцию несовместимый элемент вызывает генерацию исключения типа *ClassCastException*. Применение такого представления коллекции очень полезно во время отладки, потому что гарантирует наличие в коллекции только подходящих элементов. К родственным методам относятся также методы: *CheckedSet*(), *CheckedList*(), *CheckedMap*() и т.д. Они возвращают представление, наделенное типовой безопасностью, для указанной коллекции.

Кроме того, в класс *Collections* включены следующие новые методы:

- метод frequency (), возвращающий число вхождений элемента;
- метод disjoint (), возвращающий значение *true*, если у двух коллекций нет общих элементов;
- метод addAll(), вставляющий содержимое массива в коллекцию;
- метод *reverseOrder*(), возвращающий объект типа *comparator* для обратного порядка следования элементов.

Почему настраиваемые коллекции

Подробное обсуждение преимуществ использования настраиваемых типов применительно к коллекциям можно найти *в главе 3*. Здесь же приведено краткое резюме.

В прошлом в коллекции содержались ссылки на объекты класса *Object*, которые могли указывать на объекты любого типа. Теперь Вы явно задаете тип данных, хранящийся в коллекции. Например, вызов:

```
list.add("Collections");
```

корректен, т. к. строка в кавычках — это тип *String*. Приведенный далее оператор содержит ошибку, поскольку делается попытка сохранить объект класса *Integer* в переменной *list*:

```
list.add(new Integer(10) ); // Ошибка
```

Этот вызов порождает ошибку несовместимости типов, потому что класс *Integer* несовместим с типом *String*.

Еще одно преимущество применения настраиваемых типов в коллекциях заключается в том, что становится ненужным приведение типов при извлечении элемента из коллекции. В исходном тексте старого стиля для получения строки из коллекции *list* пришлось бы написать оператор, подобный следующему:

```
String str = (String) list.get(0); //код старого стиля
```

Раньше приведение типов было необходимо, так как вызов *list.get* () возвращал тип *Object*, а не *String*. Теперь, с появлением настраиваемых типов, Вы можете написать следующий оператор:

```
String str = list.get(0); // код нового стиля, автоматически извлекается
                          // тип String
```

поскольку переменная list — типа *LinkedList<String>*, компилятору автоматически становится известно о том, что у ссылки, возвращаемой методом *list.get*(), тип *String*, и никакого явного приведения типов не требуется.

Помимо удобства, устранение явных преобразований типов также препятствует появлению ошибок. Невозможно случайно преобразовать ссылку, возвращенную методом, таким как *get* (), в

несовместимый тип. В прошлом такие некорректные приведения типов порождали ошибки времени выполнения. С появлением настраиваемых типов подобные недопустимые преобразования типов обнаруживаются на этапе компиляции и могут быть устранены до того, как код будет введен в эксплуатацию.

Модернизация других классов и интерфейсов для применения настраиваемых типов

Некоторые другие классы, интерфейсы и методы в *API* языка Java были модернизированы для применения настраиваемых типов. Например, в следующей строке перечислены интерфейсы и классы из пакета *java.lang*, ставшие теперь настраиваемыми:

```
Comparable<T>        Class<T>              ThreadLocal<T>
```

Новый класс *Enum* и интерфейс *Iterable* — теперь тоже настраиваемые. В пакете *java.lang.reflect* класс *Constructor* — также настраиваемый.

Новые классы и интерфейсы, добавленные в пакет java.lang

Версия Java 2 5.0 добавляет в пакет *java.lang* три новых класса и три новых интерфейса.

Далее перечислены новые классы:

```
Enum            ProcessBuilder          StringBuilder
```

В следующей строке приведены новые интерфейсы:

```
Appendable      Iterable<T>             Readable
```

Класс Enum подробно рассмотрен *в главе 6*. Оставшиеся классы и интерфейсы описаны в последующих разделах этой главы.

Класс ProcessBulider

Это новый класс, добавленный в пакет *java.lang*. Он обеспечивает запуск процессов (т. е. программ) и управление ими. Как Вы помните, все процессы представлены классом *Process*, и до выхода Java 2, v5.0 процесс запускался с помощью вызова метода *Runtime.exec*(). Класс *ProcessBuilder* предлагает больше возможностей контроля над процессами, чем раньше. Например, можно установить текущий рабочий каталог и изменить параметры окружения.

В классе *ProcessBuilder* определены следующие конструкторы:

```
ProcessBuilder(List<String> args)
ProcessBuilder(String...args)
```

Параметр *args* — список аргументов, содержащий имя программы для выполнения вместе с любыми требуемыми аргументами командной строки. В первом варианте конструктора аргументы передаются в объект класса *List*. Во втором — задаются как параметр переменной длины. В табл. 10.1 приведены методы, определенные в классе *ProcessBuilder*.

Метод	Описание
`List<String> command()`	Возвращает ссылку на объект класса List, который содержит имя программы и ее аргументы. Изменения в этом списке воздействуют на

	вызывающий процесс
`ProcessBuilder command(List<String> args)`	Устанавливает имя программы и ее аргументы, передавая их в параметре переменной длины *args.* Изменения в этом списке воздействуют на вызывающий процесс
`ProcessBuilder command(String...args)`	Устанавливает имя программы и ее аргументы, передавая их в параметре переменной длины *args*
`ProcessBuilder directory(File dir)`	Устанавливает текущий рабочий каталог вызывающего обьекта класса ProcessBuilder
`Map<String, String> environment()`	Возвращает переменные окружения, связанные с вызывающим объектом типа ProcessBuilder в виде пар: ключ/значение
`boolean redirectErrorStream()`	Возвращает значение true, если стандартный поток сообщений об ошибках был перенаправлен в стандартный поток вывода. Возвращает false, если потоки разделены
`ProcessBuilder redirectErrorStream(boolean merge)`	Если параметр *merge* равен true, стандартный поток сообщений об ошибках перенаправляется в стандартный поток вывода. Если параметр *merge* равен false, потоки разделены, такое состояние принято по умолчанию
`Process start() throws IOException`	Запускает процесс, заданный вызывающим объектом типа ProcessBuilder
`File directory()`	Возвращает текущий рабочий каталог вызывающего объекта типа ProcessBuilder. Это значение будет равно null, если каталог тот же, что и у программы, запустившей процесс

Для формирования процесса с помощью класса *ProcessBuilder*, Вы просто создаете экземпляр типа *ProcessBuilder*, указывая имя программы и любые необходимые аргументы. Для того чтобы запустить программу, вызовите метод *start* () для этого экземпляра. В листинге 10.1 приведен простой пример, запускающий *Notepad*, текстовый редактор операционной системы Windows. Обратите внимание на то, что имя файла для редактирования передается как аргумент.

Листинге 10.1 Применение класса ProcessBuilder

```
import java.io.*;

class PBDemo {
  public static void main(String args[])
    throws IOException {

    ProcessBuilder proc = new ProcessBuilder("notepad.exe", "testfile");
    proc.start();
  }
}
```

Класс StringBuilder

Класс *StringBuilder* идентичен хорошо известному в языке Java классу *StringBuffer* за одним важным исключением: он не синхронизирован, что означает отсутствие потоковой безопасности. Преимущество класса *StringBuilder* — более быстрая обработка. Но если Вы используете многопоточность, то вместо класса *StringBuilder* следует применять класс *StringBuffer*.

В классе String определен новый конструктор, позволяющий создавать объект типа *String* из объекта типа *StringBuilder*. Его синтаксическая запись приведена в следующей строке:

```
String(StringBuilder strBuildObj)
```

Интерфейс *Appendable*

К объекту класса, реализующего интерфейс *Appendable*, можно добавлять символы или последовательности символов. В интерфейсе *Appendable* определены следующие два метода:

```
Appendable append(char ch) throws IOException
Appendable append(CharSequence chars) throws IOException
```

В первом варианте к вызывающему объекту добавляется символ *ch*. Во втором — последовательность символов *chars*. В обоих случаях возвращается ссылка на вызывающий объект.

Интерфейс *Iterable*

Интерфейс *Iterable* должен реализовываться любым классом, объекты которого будут обрабатываться в цикле *for* стиля *for-each*. Другими словами, для того чтобы корректно обрабатывать объект в цикле *for* стиля *for-each*, в его классе должен быть реализован интерфейс *Iterable*. Этот интерфейс — настраиваемый тип, общий вид его объявления приведен в следующей строке:

```
interface Iterable<T>
```

В интерфейсе *Iterable* определен один метод *iterator*(), синтаксическая запись которого показана в следующей строке:

```
Iterator<T> iterator()
```

Метод возвращает итератор для набора элементов, содержащихся в вызывающем объекте.

Интерфейс *Readable*

Интерфейс *Readable* показывает, что объект можно использовать как поставщик символов. В нем определен метод *read* (), приведенный в следующей строке:

```
int read(CharBuffer cb) throws lOException
```

Этот метод читает символы в переменную *cb*. Он возвращает количество прочитанных символов или -1, если встретился конец файла (EOF).

Новые методы побитной обработки классов *Integer и Long*

Несколько методов побитной обработки добавлены в числовые оболочки типов *Integer* и *Long*. Методы побитных манипуляций типа *Integer* приведены в табл. 10.2. В тип *Long* включены аналогичные методы за исключением того, что они обрабатывают данные типа *long*. Их перечень дан в табл. 10.3

Таблица 10.2. *Методы побитной обработки, добавленные в тип* Integer

Метод	Описание
`static int bitCount(int num)`	Возвращает количество единичных битов в **num**

`static int highestOneBit (int num)`	Определяет **позицию** самого старшего разряда с единичным битом в **num.** Возвращает значение, в котором установлен только найденный самый старший бит. Если нет единичных битов, возвращает 0
`static int lowestOneBit (int num)`	Определяет позицию самого младшего разряда с единичным битом в ***num.*** Возвращает значение, в котором установлен только найденный самый младший бит. Если нет единичных битов, возвращает 0

Листинг 10.2. Демонстрация нескольких новых методов побитных манипуляций в классе Integer

```
class Bits {
  public static void main(String args[]) {
    int n = 170; // 10101010

    System.out.println("Value in binary: 10101010");

    System.out.println("Number of one bits: " +
                       Integer.bitCount(n));

    System.out.println("Highest one bit: " +
                       Integer.highestOneBit(n));

    System.out.println("Lowest one bit: " +
                       Integer.lowestOneBit(n));

    System.out.println("Number of leading zeros : " +
                       Integer.numberOfLeadingZeros(n));

    System.out.println("Number of trailing zeros : " +
                       Integer.numberOfTrailingZeros(n));

    System.out.println("\nBeginning with the value 1, " +
                       "rotate left 16 times.");
    n = 1;
    for(int i=0; i < 16; i++) {
      n = Integer.rotateLeft(n, 1);
      System.out.println(n);
    }
  }
}
```

Далее приведен вывод результатов работы программы из листинга I0.2:

```
Value in binary: 10101010
```

```
Number of one bits: 4
Highest one bit: 128
Lowest one bit: 2
Number of leading zeros: 24
Number of trailing zeros: 1

Begining with the value X, rotate left 16 times.
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
```

Методы signum() u reverseBytes()

В числовые оболочки типов *Integer* и *Long*, добавлены методы *signum*() и *reverseBytes* (). Далее приведены синтаксические записи этих методов для типа *Integer*:

```
static int signum(int num)
static int reverseBytes(int num)
```

Метод *signum*() возвращает -1, если число num отрицательное, 0, если нулевое, и 1, если оно положительное. Метод *reverseBytes* () меняет на противоположный порядок байтов в числе *num* и возвращает результат.

Далее приведены версии описанных методов для оболочки типа *Long*:

```
static int signum(long num)
static long reverseBytes(long num)
```

Поддержка 32-битных кодовых точек для символов Unicode

В версию Java 2 5.0 внесены значительные дополнения в типы *character* и *String* для поддержки 32-битных символов *Unicode*. В прошлом все символы *Unicode* могли храниться в шестнадцати битах, которые равны размеру значения типа *char* (и размеру значения, содержащегося в объекте типа *character*), поскольку эти значения лежали в диапазоне от 0 до FFFF. Но недавно набор символов *Unicode* был расширен и теперь требует больше 16 бит для хранения символа. Новая версия набора символов *Unicode* включает в себя символы, лежащие в диапазоне от 0 до 10FFFF.

Приведу три важных термина: *кодовая точка* или позиция (code point), *кодовая единица* или кодовое значение (*code unit*) и дополнительный символ (supplemental character), Применительно к языку Java кодовая точка — это код символа из диапазона от 0 до 10FFFF. В языке Java термин "кодовая единица" используется для ссылки на 16-битные символы. Символы, имеющие значения, большие, чем FFFF, называются дополнительными.

Расширение набора символов *Unicode* создало фундаментальные проблемы для языка Java, Поскольку у дополнительного символа значение больше, чем может вместить тип char, потребовались некоторые средства для хранения и обработки дополнительных символов. В версии Java 2 5.0 эта проблема решена двумя способами. Во-первых, язык Java использует два значения типа char для представления дополнительного символа. Первое из них называется *верхним суррогатом* (*high surrogate*), а второе — *нижним суррогатом* (*low surrogate*). Разработаны новые методы, такие как *codePointAt*(), для преобразований кодовых точек в дополнительные символы и обратно.

Во-вторых, в языке Java перегружены некоторые из существовавших ранее методов в классах *Character и String*. В перегруженных вариантах методов используются данные типа *int* вместо *char*. Поскольку размер переменной или константы типа *int* достаточно велик для размещения любого символа как единичного значения, этот тип может использоваться для хранения любого символа. Например, у метода *isDigit*() теперь два варианта, приведенные далее:

```
static boolean isDigit(char ch)
static boolean isDigit(int cp)
```

Новые подпакеты пакета java.lang

В Java 2, v5.0 добавлены следующие подпакеты пакета java.lang:

- java.lang.annotation
- java.lang.instrument
- java.lang.management

Далее дано краткое описание каждого из них:

java.lang.annotation

Новая функциональная возможность включения аннотаций в исходные тексты программ обеспечивается подпакетом *java.lang.annotation*. В нем определен интерфейс *Annotation* и перечислимые типы *ElementType* и *Retention*-Policy (аннотации подробно описаны в *главе 7).*

java.lang. instrument

В этом подпакете определены функциональные возможности для оснащения средствами контроля различных аспектов выполнения программ. Он содержит интерфейсы *Instrumentation* и *ClassFileTransformer* и класс *ClaasDefinition*.

java.lang.management

Пакет *java.lang.management* обеспечивает поддержку управления виртуальной машины Java (JVM) и среды выполнения программы. Пользуясь функциональными средствами этого подпакета, Вы можете наблюдать за различными аспектами выполнения программы и управлять ими.

Классы Formatter и Scanner

В версии Java 2 5.0 существенно усовершенствованы средства форматированного ввода/вывода с помощью классов *Formatter* и *scanner*, добавленных в пакет *java.util*. Они подробно описаны в *главе 9.*