Глава 12

ОБОБЩЕННОЕ ПРОГРАММИРОВАНИЕ

В ЭТОЙ ГЛАВЕ...

- Назначение обобщенного программирования
- Определение простого обобщенного класса
- Обобщенные методы
- Ограничения переменных типов
- Обобщенный код и виртуальная машина
- Ограничения и лимиты
- Правила наследования обобщенных типов
- Подстановочные типы
- Рефлексия и обобщения

Обобщения (generics) представляют собой наиболее существенное изменение в языке программирования Java со времен версии 1.0. Появление обобщений в Java 5.0 стало результатом первых требований к спецификации Java (Java Specification Requests – JSR 14), которые были сформулированы в 1999 г. Группа экспертов потратила около пяти лет на разработку спецификаций и тестовые реализации.

Обобщения понадобились потому, что они позволяют писать более безопасный код, который легче читается, чем код, перегруженный переменными `Object` и приведениями типов. Обобщения особенно полезны для классов коллекций, таких как вездесущий `ArrayList`.

Обобщения похожи – по крайней мере, внешне – на шаблоны в C++. В C++, как и в Java, шаблоны впервые были добавлены для поддержки строго типизированных коллекций. Однако с годами были открыты и другие их применения. После прочтения этой главы, возможно, вы найдете новые, ранее неизвестные применения для обобщений в своих программах.

Назначение обобщенного программирования

Обобщенное программирование означает написание кода, который может быть многократно использован с объектами многих различных типов. Например, вы не хотите программировать отдельные классы для коллекционирования объектов `String` и `File`. И вам не нужно этого делать – единственный `ArrayList` может собирать объекты любого класса. Это лишь один пример обобщенного программирования.

До появления Java SE 5.0 обобщенное программирование в Java всегда реализовывалось посредством наследования. Класс `ArrayList` просто поддерживал массив ссылок на `Object`:

```
public class ArrayList // до Java SE 5.0
{
  public Object get(int i) { . . . }
  public void add(Object o) { . . . }
  . . .
  private Object[] elementData;
}
```

Такой подход порождает две проблемы. Всякий раз, когда извлекается значение, необходимо выполнить приведение типа:

```
ArrayList files = new ArrayList();
. . .
String filename = (String) names.get(0);
```

Более того, здесь нет проверки ошибок. Вы можете добавить значение любого класса:

```
files.add(new File(". . ."));
```

Этот вызов компилируется и запускается без ошибок. Но затем попытка выполнить приведение результата метода `get()` к типу `String` вызовет ошибку.

Обобщения предлагают лучшее решение: параметры типа. Класс `ArrayList` теперь принимает параметр типа, указывающий тип хранимых элементов:

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

Это облегчает чтение вашего кода. Вы можете сразу понять, что этот конкретный массив-список содержит объекты `String`.

Компилятор также может успешно использовать эту информацию. Не требуется никакого приведения при вызове `get()`. Компилятор знает, что типом возврата этого метода является `String`, а не `Object`:

```
String filename = files.get(0);
```

Компилятор также знает, что метод `add()` в `ArrayList<String>` имеет параметр типа `String`. Это намного безопаснее, чем иметь дело с параметром типа `Object`. Теперь компилятор может проконтролировать, чтобы вы не вставили объект неверного типа. Например, следующий оператор не скомпилируется:

```
files.add(new File(". . ."); // в ArrayList<String> можно добавлять
                             // только объекты String
```

Ошибка компиляции — это намного лучше, чем исключение приведения типа во время выполнения. В этом привлекательность параметров типа: они делают вашу программу легче читаемой и более безопасной.

Кто хочет быть обобщенным программистом?

Такие обобщенные классы, как `ArrayList`, использовать легко. И большинство программистов Java просто применяют типы вроде `ArrayList<String>` — как если бы они были частью языка, подобно массивам `String[]`. (Разумеется, массивы-списки лучше простых массивов, поскольку могут автоматически расширяться.)

Однако реализовать обобщенный класс не так просто. Программисты, которые будут использовать ваш код, попытаются подставлять всевозможные классы вместо ваших параметров типа. Они ожидают, что все будет работать без досадных ограничений и запутанных сообщений об ошибках. Ваша задача как обобщенного программиста — предвидеть все возможные будущие применения вашего класса.

Насколько это трудно сделать? Приведем пример типичной проблемы, с которой сталкивались проектировщики стандартной библиотеки классов. Класс `ArrayList` содержит метод `addAll()`, предназначенный для добавления элементов другой коллекции. Программист может пожелать добавить все элементы из `ArrayList<Manager>` в `ArrayList<Employee>`. Но, конечно же, обратная задача не должна быть разрешена. Как разрешить один вызов и запретить другой? Проектировщики языка Java изобрели остроумную новую концепцию — *подстановочный тип* (wildcard type), чтобы решить эту проблему. Подстановочные типы довольно абстрактны, но они позволяют разработчику библиотеки сделать методы насколько возможно гибкими.

Обобщенное программирование разделяется на три уровня квалификации. На базовом уровне вы просто используете обобщенные классы — обычно это коллекции вроде `ArrayList` — причем делаете это, не задумываясь о том, как они работают. Большинство прикладных программистов предпочитают оставаться на этом уровне до тех пор, пока что-то не пойдет не так. Вы можете столкнуться с непонятным сообщением об ошибке, смешивая разные обобщенные классы, или же имея дело с унаследованным кодом, который ничего не знает о параметрах типа. В такой момент вам нужно знать достаточно об обобщениях Java, чтобы решить проблему системно, а не "методом тыка". И, наконец, конечно, вы можете решить реализовать свои собственные обобщенные классы и методы.

Прикладным программистам, вероятно, не придется писать много обобщенного кода. Разработчики из Sun уже выполнили самую тяжелую работу и предусмотрели параметры типа для всех классов коллекций. В качестве эмпирического правила можно сказать так: от применения параметров типа выигрывает только тот код, в котором традиционно присутствует много приведений от очень общих типов (таких как `Object` или интерфейс `Comparable`).

В этой главе мы расскажем вам все, что необходимо знать для реализации вашего собственного обобщенного кода. Однако вы надеемся, что наши читатели используют свои знания, прежде всего, для отыскания причин неполадок, а также для удовлетворения любопытства относительно внутреннего устройства параметризованных классов коллекций.

Определение простого обобщенного класса

Обобщенный класс — это класс с одной или более переменной типа. В этой главе мы используем в качестве примера простой класс `Pair`. Этот класс позволит сосредоточиться на механизме обобщений, не вдаваясь в подробности хранения данных. Код обобщенного класса Pair приведен ниже.

```
public class Pair<T>
{
    public Pair() { first = null; second = null; }
    public Pair(T first, T second) { this.first = first; this.second = second; }
    public T getFirst() { return first; }
    public T getSecond() { return second; }
    public void setFirst(T newValue) { first = newValue; }
    public void setSecond(T newValue) { second = newValue; }
    private T first;
    private T second;
}
```

Класс `Pair` вводит переменную типа `T`, заключенную в угловые скобки <>, следующие после имени класса. Обобщенный класс может иметь более одной переменной типа. Например, мы могли бы определить класс `Pair` с разными типами для первого и второго поля:

```
public class Pair<T, U> { . . . }
```

Переменные типа используются по всему определению класса для спецификации типа возврата методов и типов полей и локальных переменных. Например:

```
private T first; // используется переменная типа
```

На заметку! Использование заглавных букв для имен переменных типа — общепринятая практика, как и их краткость. Библиотека Java использует переменную `E` для обозначения типа элемента коллекции, `K` и `V` — для типов ключа и значения в таблице, а `T` (и соседние буквы `S` и `U` при необходимости) — для обозначения “любого типа вообще”.

Вы *создаете экземпляр* обобщенного типа, подставляя имя типа вместо переменной типа:

```
Pair<String>
```

Вы можете воспринимать результат как обычный класс с конструкторами:

```
Pair<String>()
Pair<String>(String, String)
```

и методами:

```
String getFirst()
String getSecond()
void setFirst(String)
void setSecond(String)
```

Другими словами, обобщенный класс действует как фабрика обычных классов.

Программа в листинге 12.1 запускает класс `Pair` в работу. Статический метод `minmax()` проходит по элементам массива и одновременно вычисляет минимальное и максимальное значения. Он использует объект `Pair` для возврата результата. Напомним, что метод `compareTo()` сравнивает две строки, возвращая `0`, если строки идентичны, отрицательное число — если первая строка идет перед второй в алфавитном порядке, и положительное — в противоположном случае.

На заметку! На первый взгляд обобщенные классы Java похожи на шаблонные классы C++. Единственное очевидное отличие состоит в том, что в Java не предусмотрено специальное ключевое слово `template`. Однако, как вы увидите далее в этой главе, между этими двумя механизмами есть тонкое различие.

Листинг 12.1. Содержимое файла `PairTest.java`

```
1. /**
2.  * @version 1.00 2004-05-10
3.  * @author Cay Horstmann
4.  */
5. public class PairTest1
6. {
7.  public static void main(String[] args)
8.   {
9.      String[] words = { "Mary", "had", "a", "little", "lamb" };
10.     Pair<String> mm = ArrayAlg.minmax(words);
11.     System.out.println("min = " + mm.getFirst());
12.     System.out.println("max = " + mm.getSecond());
13.  }
14.}
15.class ArrayAlg
16.{
17./**
18. * Получает минимальное и максимальное значение массива строк.
19. * @param a Массив строк
20. * @return a Пара минимального и максимального значений или null,
21. * если a пуст
22. */
23.  public static Pair<String> minmax(String[] a)
24.  {
25.     if (a == null || a.length == 0) return null;
26.     String min = a[0];
27.     String max = a[0];
28.     for (int i = 1; i < a.length; i++)
29.     {
30.         if (min.compareTo(a[i]) > 0) min = a[i];
31.         if (max.compareTo(a[i]) < 0) max = a[i];
32.     }
33.     return new Pair<String>(min, max);
34.  }
35.}
```

Обобщенные методы

В предыдущем разделе вы видели, как определяется обобщенный класс. Вы можете также определить отдельный метод с параметрами типа.

```
class ArrayAlg
{
  public static <T> T getMiddle(T[] a)
  {
    return a[a.length / 2];
  }
}
```

Этот метод определен внутри обычного класса, а не внутри обобщенного. Однако этот метод — обобщенный, на что указывают угловые скобки и переменная типа. Обратите внимание, что переменная типа вставляется после модификаторов (`public static` в нашем случае) и перед типом возврата.

Вы можете определять обобщенные методы как внутри обычных классов, так и внутри обобщенных. Когда вызывается обобщенный метод, вы можете передать ему действительные типы, заключая их в угловые скобки перед именем метода:

```
String[] names = { "John", "Q.", "Public" };
String middle = ArrayAlg.<String>getMiddle(names);
```

В этом случае (как и в большинстве случаев) вы можете пропустить параметр типа `String` при вызове метода. У компилятора есть достаточно информации, чтобы предположить, какой метод вам нужен. Он сравнивает тип `names` (т.е. `String[]` с типом `T[]`) и соображает, что `T` должно быть `String`. То есть, вы можете просто вызвать так:

```
String middle = ArrayAlg.getMiddle(names);
```

Почти во всех случаях определение типа в обобщенных методах работает гладко. Иногда компилятор ошибается и вам приходится расшифровывать его сообщение об ошибке. Рассмотрим следующий пример:

```
double middle = ArrayAlg.getMiddle(3.14, 1729, 0);
```

Сообщение об ошибке будет таким: `found: java.lang.Number&java.lang.Comparable<? extends java.lang.Number&java.lang.Comparable<?>>, required: double`. Позднее в этой главе вы узнаете, как расшифровывается объявление типа “found”. Вкратце скажем, что компилятор выполняет автоматическую упаковку (autoboxing) параметра в один объект `Double` и два объекта `Integer`, а затем пытается найти общий супертип для этих классов. Он находит их два: `Number` и интерфейс `Comparable`, который сам по себе является обобщенным типом. В этом случае решение заключается в передаче всех параметров как значений `double`.

Совет. Петер Ван Дер Ахе (Peter von der Ahe) рекомендует следующий трюк, если вы хотите увидеть, какой тип компилятор выводит при вызове обобщенного метода: намеренно допустите ошибку и изучите полученное в результате сообщение об ошибке. Например, рассмотрим вызов `ArrayAlg.getMiddle("Hello", 0, null)`. Присвоим результат `JButton`, что не может быть правильно. Мы получим такое сообщение об ошибке:

```
java.lang.Object&java.io.Serializable&java.lang.Comparable<? extends
java.lang.Object&java.io.Serializable&java.lang.Comparable<?>>."
```

Говоря просто, вы можете присвоить результат объектам типа `Object`, `Serializable` или `Comparable`.

На заметку! В C++ вы помещаете параметр типа после имени метода. Это может привести к неприятной неоднозначности при разборе кода. Например, `g(f<a,b>(c))` может означать "вызвать `g` с результатом вызова `f<a,b>(c)`", или же "вызвать `g` с двумя булевскими значениями `f<a` и `b>(c)`".

Ограничения переменных типов

Иногда класс или метод нуждается в наложении ограничений на переменные типов. Приведем типичный пример. Нужно вычислить минимальный элемент массива:

```
class ArrayAlg
{
  public static <T> T min(T[] a) // почти правильно
  {
    if (a == null || a.length == 0) return null;
    T smallest = a[0];
    for (int i = 1; i < a.length; i++)
       if (smallest.compareTo(a[i]) > 0) smallest = a[i];
    return smallest;
  }
}
```

Но здесь есть проблема. Взгляните на код метода `min()`. Переменная `smallest` имеет тип `T`, а это означает, что она может быть объектом произвольного класса. Откуда мы знаем, имеет ли класс `T` метод `compareTo()`?

Решение заключается в том, чтобы наложить ограничение на тип `T`, чтобы вместо него можно было подставлять только класс, реализующий `Comparable` — стандартный интерфейс с единственным методом `compareTo()`. Это делается добавлением ограничения (`bound`) переменной типа `T`:

```
public static <T extends Comparable> T min(T[] a) . . .
```

В действительности интерфейс `Comparable` сам является обобщенным типом. Пока мы проигнорируем эту сложность и предупреждения, которые генерирует компилятор. Далее в разделе "Подстановочные типы" мы обсудим, как правильно использовать параметры типа с интерфейсом `Comparable`.

Теперь обобщенный метод `min()` может вызываться только с массивами классов, реализующих интерфейс `Comparable`, таких как `String`, `Date` и т.п. Вызов `min()` с массивом `Rectangle` даст ошибку во время компиляции, поскольку класс `Rectangle` не реализует `Comparable`.

На заметку! В C++ вы не можете ограничить типы в параметрах шаблонов. Если программист создаст экземпляр шаблона с неправильным типом, сообщение об ошибке (часто невнятное) появится в шаблонном коде.

Вас может удивить, почему здесь используется ключевое слово `extends` вместо `implements`, ведь `Comparable` — это интерфейс? Обозначение

```
<T extends Ограничивающий_тип>
```

говорит о том, что T должен быть *подтипом* ограничивающего типа. Ключевое слово `extends` выбрано потому, что это резонное приближение концепции подтипа, и проектировщики Java не хотели добавлять в язык новое ключевое поле (такое как `sub`).

Переменная типа или тип подстановки могут иметь несколько ограничений. Например:

```
T extends Comparable & Serializable
```

Ограничивающие типы разделяются знаком &, потому что запятые используются для разделения переменных типа.

Как и в наследовании Java, вы можете иметь столько интерфейсных подтипов, сколько хотите, но только один из ограничивающих типов может быть классом. Если вы используете для ограничения класс, он должен быть первым в списке ограничений.

В следующем примере программы (листинг 12.2) мы перепишем метод `minmax()`, сделав его обобщенным. Метод вычисляет минимум и максимум в обобщенном массиве, возвращая `Pair<T>`.

Листинг 12.2. Содержимое файла `PairTest2.java`

```
1. import java.util.*;
2. /**
3.  * @version 1.00 2004-05-10
4.  * @author Cay Horstmann
5.  */
6. public class PairTest2
7. {
8.      public static void main(String[] args)
9.      {
10.        GregorianCalendar[] birthdays =
11.        {
12.          new GregorianCalendar(1906, Calendar.DECEMBER, 9), // G. Hopper
13.          new GregorianCalendar(1815, Calendar.DECEMBER, 10), // A. Lovelace
14.          new GregorianCalendar(1903, Calendar.DECEMBER, 3), // J. von Neumann
15.          new GregorianCalendar(1910, Calendar.JUNE, 22), // K. Zuse
16.        };
17.        Pair<GregorianCalendar> mm = ArrayAlg.minmax(birthdays);
18.        System.out.println("min = " + mm.getFirst().getTime());
19.        System.out.println("max = " + mm.getSecond().getTime());
20.     }
21. }
22. class ArrayAlg
23. {
24.   /**
25.    * Получает минимум и максимум из массива объектов типа T.
26.    * @param a Массив объектов типа T
27.    * @return Пара с минимальным и максимальным значениями
28.    *         или null, если a пуст или равен null
29.    */
30.   public static <T extends Comparable> Pair<T> minmax(T[] a)
31.   {
32.     if (a == null || a.length == 0) return null;
33.     T min = a[0];
34.     T max = a[0];
35.     for (int i = 1; i < a.length; i++)
36.     {
37.            if (min.compareTo(a[i]) > 0) min = a[i];
38.            if (max.compareTo(a[i]) < 0) max = a[i];
39.     }
40.     return new Pair<T>(min, max);
41.   }
42. }
```

Обобщенный код и виртуальная машина

Виртуальная машина не имеет дела с объектами обобщенных типов — все объекты принадлежат обычным классам. Ранние версии реализаций обобщения могли даже компилировать программу, использующую обобщения в файлы классов, которые могла исполнять виртуальная машина версии 1.0! Такая обратная совместимость позднее, в процессе работы над средствами обобщения Java, была отброшена. Если вы применяете компилятор Sun для компиляции кода, использующего средства обобщения Java, то результирующие файлы классов *не* будут выполняться на виртуальных машинах версий, предшествующих 5.0.

На заметку! Если вы хотите воспользоваться преимуществами обобщений, сохраняя совместимость байт-кода с более старыми виртуальными машинами, загляните на `http://sourceforge.net/projects/retroweaver`. Программа Retroweaver перепишет файлы классов, так что они будут совместимыми со старыми виртуальными машинами.

Всякий раз при определении обобщенного типа автоматически создается соответствующий ему "сырой" (raw) тип. Имя этого типа совпадает с именем обобщенного типа с удаленными параметрами типа. Переменные типа удаляются и заменяются ограничивающими типами (или типом `Object`, если переменная не имеет ограничений).

Например, "сырой" тип для `Pair<T>` выглядит, как показано ниже.

```
public class Pair
{
    public Pair(Object first, Object second)
    {
        this.first = first;
        this.second = second;
    }
    public Object getFirst() { return first; }
    public Object getSecond() { return second; }
    public void setFirst(Object newValue) { first = newValue; }
    public void setSecond(Object newValue) { second = newValue; }
    private Object first;
    private Object second;
}
```

Поскольку `T` — неограниченная переменная типа, она просто заменяется `Object`.

В результате получается обычный класс вроде тех, что вы могли реализовать раньше, до появления средств обобщений в язык программирования Java.

Ваши программы содержат разные варианты `Pair`, такие как `Pair<String>` или `Pair<GregorianCalendar>`, но такая "подчистка" превращает их в "сырой" тип `Pair`.

На заметку! В этом отношении обобщения Java очень отличаются от шаблонов C++. C++ производит разные типы для каждого экземпляра шаблона — этот феномен называется "разбуханием шаблонного кода". Java не страдает от этой проблемы.

"Сырой" тип заменяет переменные типов первым ограничением или же `Object`, если никаких ограничений не предусмотрено. Например, переменная типа в классе `Pair<T>` не имеет явных ограничений, а потому сырой тип заменяет `T` на `Object`.

Предположим, что объявлен немного отличающийся тип:

```
public class Interval<T extends Comparable & Serializable> implements Serializable
{
    public Interval(T first, T second)
    {
        if (first.compareTo(second) <= 0) { lower = first; upper = second; }
        else { lower = second; upper = first; }
    }
    . . .
    private T lower;
    private T upper;
}
```

Сырой тип `Interval` выглядит следующим образом:

```
public class Interval implements Serializable
{
    public Interval(Comparable first, Comparable second) { . . . }
    . . .
    private Comparable lower;
    private Comparable upper;
}
```

На заметку! Вы можете спросить, что случится, если поменять местами ограничения: `class Interval<Serializable & Comparable>`. В этом случае сырой тип заменит `T` на `Serializable`, и компилятор вставит, где нужно, приведения к `Comparable`. Поэтому для эффективности вы должны помещать пустые интерфейсы (т.е. интерфейсы без методов) в конец списка ограничений.

Трансляция обобщенных выражений

Когда ваша программа вызывает обобщенный метод, компилятор вставляет приведения, когда удаляется возвращаемый тип. Например, рассмотрим следующую последовательность операторов:

```
Pair<Employee> buddies = . . .;
Employee buddy = buddies.getFirst();
```

Подчистка `getFirst()` приведет к возврату типа `Object`. Компилятор автоматически вставит приведение к `Employee`. То есть компилятор транслирует вызов метода в две инструкции виртуальной машины.

- Вызов сырого метода `Pair.getFirst()`.
- Приведение возвращенного объекта `Object` к типу `Employee`.

Приведения также вставляются при обращении к обобщенному полю. Предположим, что поля `first` и `second` были бы общедоступными, т.е. `public`. (Возможно, это нехороший стиль программирования, но вполне допустимый в Java.) Тогда в операторе

```
Employee buddy = buddies.first;
```

в результирующем байт-коде также будут вставлены приведения.

Трансляция обобщенных методов

Подчистка типов также происходит и с обобщенными методами. Программисты обычно воспринимают обобщенные методы вроде этого:

```
public static <T extends Comparable> T min(T[] a)
```

как целое семейство методов. Но после подчистки остается только один метод:

```
public static Comparable min(Comparable[] a)
```

Обратите внимание, что параметр типа `T` удален, а остается только ограничивающий тип `Comparable`.

Подчистка методов вносит пару сложностей. Рассмотрим пример:

```
class DateInterval extends Pair<Date>
{
    public void setSecond(Date second)
    {
        if (second.compareTo(getFirst()) >= 0)
            super.setSecond(second);
    }
    . . .
}
```

Интервал дат задается парой объектов `Date`, и мы хотим переопределить эти методы, чтобы гарантировать, что второе значение никогда не будет меньше первого. Этот класс очищается до следующего:

```
class DateInterval extends Pair // после чистки
{
    public void setSecond(Date second) { . . . }
    . . .
}
```

Возможно, это неожиданно, но существует и другой метод `setSecond()`, унаследованный от `Pair`, а именно:

```
public void setSecond(Object second)
```

Это явно другой метод, поскольку он имеет параметр другого типа — `Object` вместо `Date`. Но он *не должен* быть другим. Рассмотрим последовательность операторов:

```
DateInterval interval = new DateInterval(. . .);
Pair<Date> pair = interval; // OK—присвоение суперклассу
pair.setSecond(aDate);
```

Мы ожидаем, что вызов `setSecond()` является полиморфным и потому будет вызван соответствующий метод. Поскольку `pair` ссылается на объект `DateInterval`, это должен быть `DateInterval.setSecond()`. Но проблема в том, что подчистка типов пересекается с полиморфизмом. Чтобы решить эту проблему, компилятор генерирует *метод-мост* (bridge method) в классе `DateInterval`:

```
public void setSecond(Object second) { setSecond((Date) second); }
```

Чтобы понять, почему это работает, тщательно проследим за выполнением приведенного ниже оператора:

```
pair.setSecond(aDate)
```

Переменная `pair` объявлена с типом `Pair<Date>`, и этот тип имеет только один метод по имени `setSecond`, а именно — `setSecond(Object)`. Виртуальная маши-

на вызывает этот метод на объекте, на который ссылается pair. Этот объект имеет тип DateInterval, потому вызывается метод DateInterval.setSecond(Object). Этот метод и есть синтезированный метод-мост. Он вызывает DateInterval.setSecond(Date), что нам и нужно.

Метод-мост может быть еще более странным. Предположим, что метод DateInterval также переопределяет getSecond:

```
class DateInterval extends Pair<Date>
{
    public Date getSecond() { return (Date) super.getSecond().clone(); }
    . . .
}
```

В очищенном типе есть два метода getSecond:

```
Date getSecond()     // определен в DateInterval
Object getSecond()   // определен в Pair
```

Вы не сможете написать такой Java-код без параметров — было бы некорректно иметь два метода с одинаковыми типами параметров. Однако в виртуальной машине типы параметров *и тип возврата* специфицируют метод. Поэтому компилятор может произвести байт-код двух методов, отличающихся только типом возврата, и виртуальная машина корректно справляется с такой ситуацией.

На заметку! Методы-мосты не ограничиваются обобщенными типами. Мы уже упоминали в главе 5 о том, что, начиная с версии Java SE 5.0, допускается, чтобы метод специфицировал более ограниченный тип возврата, когда он переопределяет другой метод. Например:

```
public class Employee implements Cloneable
{
  public Employee clone() throws CloneNotSupportedException { ... }
}
```

О методах Object.clone и Employee.clone говорят, что это *ковариантные типы*. В действительности класс Employee имеет два таких метода:

```
Employee clone() // определен выше
Object clone() // синтезированный метод-мост, переопределяет Object.clone
```

Синтезированный метод-мост вызывает вновь определенный метод.

В итоге вам нужно помнить следующие факты о трансляции обобщений Java.

- Для виртуальной машины обобщений не существует, есть только обычные классы и методы.
- Все параметры типа заменяются типами-ограничителями.
- Методы-мосты синтезируются для предохранения полиморфизма.
- Приведения вставляются по необходимости для обеспечения безопасности типов.

Вызов унаследованного кода

До появления Java SE 5.0 был написан огромный объем кода Java. Если бы обобщенные классы не могли взаимодействовать с этим кодом, они не получили бы широкого распространения. К счастью, достаточно просто использовать обобщенные классы вместе с их "сырыми" эквивалентами из унаследованного API.

Рассмотрим конкретный пример. Чтобы установить метки JSlider, вы используете метод

```
void setLabelTable(Dictionary table)
```

В главе 9 мы применяли следующий код для наполнения таблицы меток:

```
Dictionary<Integer, Component> labelTable = new Hashtable<Integer, Component>();
labelTable.put(0, new JLabel(new ImageIcon("nine.gif")));
labelTable.put(20, new JLabel(new ImageIcon("ten.gif")));
. . .
slider.setLabelTable(labelTable); // WARNING
```

В Java SE 5.0 классы `Dictionary` и `Hashtable` превратились в обобщенные классы. Поэтому можно сформировать `Dictionary<Integer, Component>` вместо использования "сырого" `Dictionary`. Однако когда вы передаете объект `Dictionary<Integer, Component>` методу `setLabelTable`, компилятор выдает предупреждение:

```
Dictionary<Integer, Component> labelTable = . . .;
slider.setLabelTable(labelTable); // предупреждение
```

В конце концов, компилятор не знает о том, что `setLabelTable()` может делать с объектом `Dictionary`. Этот метод может заменить все ключи строками. Это нарушит гарантию того, что ключи имеют тип `Integer`, и тогда будущие операции могут вызвать исключения приведения.

С этим предупреждением мало что можно сделать, кроме как обдумать его и спросить себя, что, скорее всего, `JSlider` собирается делать с объектом `Dictionary`? В нашем случае вполне ясно, что `JSlider` только читает информацию, так что мы можем проигнорировать предупреждение.

Теперь рассмотрим противоположный случай, когда вы получаете объект "сырого" типа от унаследованного класса. Вы можете присвоить его переменной параметризованного типа, но конечно, при этом получите предупреждение. Например:

```
Dictionary<Integer, Components> labelTable =
     slider.getLabelTable(); // предупреждение
```

Все хорошо — просмотрите предупреждение и убедитесь, что таблица меток действительно содержит объекты `Integer` и `Component`. Конечно, абсолютной гарантии нет. Злоумышленник может инсталлировать другой `Dictionary` в `JSlider`. Но, опять-таки, эта ситуация на хуже той, что была до версии Java SE 5.0. В худшем случае ваша программа сгенерирует исключение.

После того, как вы обратили внимание на предупреждение, вы можете использовать *аннотацию* для того, чтобы оно исчезло. Аннотацию следует поместить перед методом, чей код генерирует предупреждение, как показано ниже:

```
@SuppressWarnings("unchecked")
public void configureSlider() { . . . }
```

К сожалению, эта аннотация отключает проверку всего кода внутри метода. Будет хорошей идеей изолировать потенциально небезопасный код в отдельных методах, чтобы его можно было легко просмотреть.

На заметку! `Hashtable` — конкретный подкласс абстрактного класса `Dictionary`. И `Dictionary`, и `Hashtable` объявлены устаревшими, с тех пор как в Java SE 1.2 для них появилась замена в виде интерфейса `Map` и класса `HashMap`. Несмотря на это, упомянутые устаревшие классы до сих пор живут и здравствуют. В конце концов, класс `JSlider` был добавлен только в Java 1.3. Знали ли его разработчики о классе `Map` в то время? Можно ли рассчитывать, что они адаптируют его к обобщениям в будущем? В принципе, подобное всегда происходит с унаследованным кодом.