

# 第8章 泛型程序设计

- ▲ 为什么要使用泛型程序设计
- ▲ 定义简单泛型类
- ▲ 泛型方法
- ▲ 类型变量的限定
- ▲ 泛型代码和虚拟机
- ▲ 限制与局限性
- ▲ 泛型类型的继承规则
- ▲ 通配符类型
- ▲ 反射和泛型

泛型类和泛型方法有类型参数，这使得它们可以准确地描述用特定类型实例化时会发生什么。在有泛型类之前，程序员必须使用 `Object` 编写适用于多种类型的代码。这很烦琐，也很不安全。

随着泛型的引入，Java 有了一个表述能力很强的类型系统，允许设计者详细地描述变量和方法的类型要如何变化。对于简单的情况，你会发现实现泛型代码很容易。不过，在更高级的情况下，对于实现者来说这会相当复杂。其目标是提供让其他程序员可以轻松使用的类和方法而不会出现意外。

Java 5 中泛型的引入成为 Java 程序设计语言自最初发行以来最显著的变化。Java 的一个主要设计目标是支持与之前版本的向后兼容性。因此，Java 的泛型有一些让人不快的局限性。在本章中，你会了解泛型程序设计的优点以及存在的问题。

## 8.1 为什么要使用泛型程序设计

泛型程序设计（generic programming）意味着编写的代码可以对多种不同类型的对象重用。例如，你并不希望为收集 `String` 和 `File` 对象分别编写不同的类。实际上，也不需要这样做，因为一个 `ArrayList` 类就可以收集任何类的对象。这就是泛型程序设计的一个例子。

实际上，在 Java 有泛型类之前已经有一个 `ArrayList` 类。下面来研究泛型程序设计的机制是如何演变的，另外还会介绍这对于用户和实现者来说意味着什么。

### 8.1.1 类型参数的好处

在 Java 中增加泛型类之前，泛型程序设计是用继承（inheritance）实现的。`ArrayList` 类只维护一个 `Object` 引用的数组：

```
public class ArrayList // before generic classes
{
    private Object[] elementData;
    ...
}
```

```
public Object get(int i) { . . . }
public void add(Object o) { . . . }
}
```

这种方法存在两个问题。获取一个值时必须进行强制类型转换：

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

此外，这里没有错误检查。可以向数组列表中添加任何类的值：

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

对于这个调用，编译和运行都不会出错。不过在其他地方，如果将 `get` 的结果强制类型转换为 `String` 类型，就会产生一个错误。

泛型提供了一个更好的解决方案：类型参数（type parameter）。`ArrayList` 类现在有一个类型参数用来指示元素的类型：

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

这使得代码具有更好的可读性。人们一看就知道这个数组列表中包含的是 `String` 对象。

**注释：**如果用一个明确的类型而不是 `var` 声明一个变量，则可以通过使用“菱形”语法省略构造器中的类型参数：

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

省略的类型可以从变量的类型推断得出。

Java 9 扩展了菱形语法的使用范围，原先不接受这种语法的地方现在也可以使用了。例如，现在可以对匿名子类使用菱形语法：

```
ArrayList<String> passwords = new ArrayList<>() // diamond OK in Java 9
{
    public String get(int n) { return super.get(n).replaceAll(".", "*"); }
};
```

编译器也可以充分利用这个类型信息。调用 `get` 的时候，不再需要强制类型转换。编译器知道返回值类型为 `String`，而不是 `Object`：

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

编译器还知道 `ArrayList<String>` 的 `add` 方法有一个类型为 `String` 的参数，这比有一个 `Object` 类型的参数要安全得多。现在，编译器会检查，防止你插入错误类型的对象。例如，以下语句

```
files.add(new File(".")); // can only add String objects to an ArrayList<String>
```

是无法通过编译的。不过，得到编译错误要比运行时出现类的强制类型转换异常好得多。

这正是类型参数的魅力所在：它们会让你的程序更易读，也更安全。

### 8.1.2 谁想成为泛型程序员

使用类似 `ArrayList` 的泛型类很容易。大多数 Java 程序员使用 `ArrayList<String>` 之类的类

型时就好像它们是 Java 语言内置的类型一样（就像 `String[]` 数组）。（当然，数组列表比数组更好，因为数组列表可以自动扩展。）

但是，实现一个泛型类可没有那么容易。使用你的代码的程序员可能会插入各种各样的类作为类型参数。他们希望一切都能正常工作，不会有恼人的限制，也不会有让人混乱的错误消息。因此，作为一个泛型程序员，你的任务就是要预计到你的泛型类将来所有可能的用法。

这个任务会有多难呢？下面来看让标准类库的设计者饱受折磨的一个典型问题。`ArrayList` 类有一个方法 `addAll`，用来添加另一个集合的全部元素。一个程序员可能想要将一个 `ArrayList<Manager>` 中的所有元素添加到一个 `ArrayList<Employee>` 中去。不过，当然反过来应该不合法。如何允许前一个调用，而不允许后一个调用呢？Java 语言的设计者发明了一个具有独创性的新概念来解决这个问题，即通配符类型（wildcard type）。通配符类型非常抽象，不过，利用通配符类型，构建类库的程序员可以编写出尽可能灵活的方法。

泛型程序设计可以分为 3 个能力水平。基本水平是，仅仅使用泛型类（比较典型的是像 `ArrayList` 这样的集合），而不考虑它们如何工作以及为什么这样做。大多数应用程序员都希望保持在这一水平，除非出现了问题。不过，当混合使用不同的泛型类时，或者要与对类型参数一无所知的遗留代码交互时，你可能会看到让人困惑的错误消息。那时你就需要对 Java 泛型有足够的了解，才能系统地解决问题，而不是胡乱地猜测。当然，最终你可能想要实现自己的泛型类与泛型方法。

应用程序员很可能不会编写太多的泛型代码。JDK 开发人员已经做出了很大的努力，为所有的集合类提供了类型参数。凭经验来说，只有原本涉及大量通用类型（如 `Object` 或 `Comparable` 接口）的强制类型转换的代码才会因使用类型参数而受益。

本章将介绍实现自己的泛型代码所需了解的全部知识。不过，希望大多数读者主要利用这些知识来帮助排除代码的问题，以及满足想要了解参数化集合类内部工作原理的好奇心。

## 8.2 定义简单泛型类

泛型类（generic class）就是有一个或多个类型变量的类。本章使用一个简单的 `Pair` 类作为例子。这个类使我们可以只关注泛型，而不用为数据存储的细节而分心。下面是泛型 `Pair` 类的代码：

```
public class Pair<T>
{
    private T first;
    private T second;

    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; }
}
```

`Pair` 类引入了一个类型变量 `T`, 用尖括号 (`< >`) 括起来, 放在类名的后面。泛型类可以有多个类型变量。例如, 可以定义 `Pair` 类, 其中第一个字段和第二个字段使用不同的类型:

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

类型变量在整个类定义中用于指定方法的返回类型以及字段和局部变量的类型。例如,

```
private T first; // uses the type variable
```

**注释:** 常见的做法是类型变量使用大写字母, 而且很简短。Java 类库使用变量 `E` 表示集合的元素类型, `K` 和 `V` 分别表示表的键和值的类型。`T` (必要时还可以用相邻的字母 `U` 和 `S`) 表示“任意类型”。

可以用具体的类型替换类型变量来实例化 (instantiate) 泛型类型, 例如:

```
Pair<String>
```

可以把结果想象成一个普通类, 它有以下构造器:

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

以及以下方法:

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

换句话说, 泛型类相当于普通类的工厂。

程序清单 8-1 中的程序具体使用了 `Pair` 类。静态方法 `minmax` 会遍历数组并同时计算出最小值和最大值。它用一个 `Pair` 对象同时返回两个结果。回想一下: 用 `compareTo` 方法比较两个字符串, 如果字符串相同则返回 0; 按照字典顺序, 如果第一个字符串比第二个字符串靠前, 就返回一个负整数, 否则返回一个正整数。

### 程序清单 8-1 pair1/PairTest1.java

```
1 package pair1;
2
3 /**
4  * @version 1.01 2012-01-26
5  * @author Cay Horstmann
6  */
7 public class PairTest1
8 {
9     public static void main(String[] args)
10    {
11        String[] words = { "Mary", "had", "a", "little", "lamb" };
12        Pair<String> mm = ArrayAlg.minmax(words);
13        System.out.println("min = " + mm.getFirst());
14        System.out.println("max = " + mm.getSecond());
15    }
16 }
```

```

17
18 class ArrayAlg
19 {
20     /**
21      * Gets the minimum and maximum of an array of strings.
22      * @param a an array of strings
23      * @return a pair with the min and max values, or null if a is null or empty
24      */
25     public static Pair<String> minmax(String[] a)
26     {
27         if (a == null || a.length == 0) return null;
28         String min = a[0];
29         String max = a[0];
30         for (int i = 1; i < a.length; i++)
31         {
32             if (min.compareTo(a[i]) > 0) min = a[i];
33             if (max.compareTo(a[i]) < 0) max = a[i];
34         }
35         return new Pair<>(min, max);
36     }
37 }

```

**C++ 注释：**从表面上看，Java 的泛型类类似于 C++ 的模板类。唯一明显的不同是 Java 没有特殊的 template 关键字。但是，在本章中你将会看到，这两种机制有着本质的区别。

### 8.3 泛型方法

上一节已经介绍了如何定义一个泛型类。还可以定义一个带有类型参数的方法。

```

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

```

这个方法是在普通类中定义的，而不是在泛型类中。不过，这是一个泛型方法，可以从尖括号和类型变量看出这一点。注意，类型变量放在修饰符（这里的修饰符就是 public static）的后面，并在返回类型的前面。

可以在普通类中定义泛型方法，也可以在泛型类中定义。

当调用一个泛型方法时，可以把具体类型包围在尖括号中，放在方法名前面：

```
String middle = ArrayAlg.<String>getMiddle("John", "Q.", "Public");
```

在这种情况下（实际也是大多数情况下），方法调用中可以省略 `<String>` 类型参数。编译器有足够的信息推断出你想要的方法。它将参数的类型与泛型类型 `T...` 进行匹配，推断出 `T` 一定是 `String`。也就是说，可以简单地调用

```
String middle = ArrayAlg.getMiddle("John", "Q.", "Public");
```

几乎在所有情况下，泛型方法的类型推导都能正常工作。偶尔，编译器也会提示错误，此时你就需要解读错误报告。考虑下面这个示例：

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

错误消息以晦涩的方式指出（不同的编译器版本给出的错误消息可能有所不同）：解释这个代码有两种方式，而且这两种方式都是合法的。简单地说，编译器将把参数自动装箱为 1 个 Double 和 2 个 Integer 对象，然后寻找这些类的共同超类型。事实上，它找到了 2 个超类型：Number 和 Comparable 接口，Comparable 接口本身也是一个泛型类型。在这种情况下，可以采取的补救措施是将所有的参数都写为 double 值。

 **提示：**如果想知道编译器对一个泛型方法调用最终推断出哪种类型，Peter von der Ahé 推荐了这样一个窍门：故意引入一个错误，然后研究所得到的错误消息。例如，考虑调用 `ArrayAlg.getMiddle("Hello", 0, null)`。将结果赋给 JButton，这肯定是不对的。将会得到一个错误报告：

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

大致的意思是：可以将结果赋给 Object、Serializable 或 Comparable。

 **C++ 注释：**在 C++ 中，要将类型参数放在方法名后面。这有可能会导致烦人的解析二义性。例如，`g(f<a, b>(c))` 可以理解为“用 `f<a, b>(c)` 的结果调用 `g`”，或者理解为“用两个布尔值 `f<a` 和 `b>(c)` 调用 `g`”。

## 8.4 类型变量的限定

有时，类或方法需要对类型变量加以约束。下面是一个典型的例子。我们要计算数组中的最小元素：

```
class ArrayAlg
{
    public static <T> T min(T[] a) // almost correct
    {
        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;
    }
}
```

但是，这里有一个问题。请看 `min` 方法的代码。变量 `smallest` 的类型为 `T`，这意味着它可以是任何一个类的对象。如何知道 `T` 所属的类有一个 `compareTo` 方法呢？

解决这个问题的办法是限制 T 只能是实现了 Comparable 接口（包含一个方法 compareTo 的标准接口）的一个类。通过对类型变量 T 设置一个限定（bound）来实现这一点：

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

实际上 Comparable 接口本身就是一个泛型类型。目前，我们先忽略其复杂性以及编译器产生的警告。8.8 节会讨论如何在 Comparable 接口中适当地使用类型参数。

现在，泛型方法 min 只能在实现了 Comparable 接口的类（如 String、LocalDate 等）的数组上调用。因为 Rectangle 类没有实现 Comparable 接口，所以在 Rectangle 数组上调用 min 将会得到一个编译错误。

**C++ 注释：**在 C++ 中，不能对模板参数的类型加以限制。如果程序员用一个不适当的类型实例化一个模板，将会在模板代码中报告一个（通常含糊不清的）错误消息。

你或许会感到奇怪——在这里我们为什么使用关键字 extends 而不是 implements？毕竟，Comparable 是一个接口。下面的记法

```
<T extends BoundingType>
```

表示 T 应该是限定类型（bounding type）的子类型（subtype）。T 和限定类型可以是类，也可以是接口。选择关键字 extends 的原因是它更接近子类型的概念，并且 Java 的设计者也不打算在语言中再添加一个新的关键字（如 sub）。

一个类型变量或通配符可以有多个限定，例如：

```
T extends Comparable & Serializable
```

限定类型用“&”分隔，而逗号用来分隔类型变量。

按照 Java 继承机制，可以根据需要拥有多个接口超类型，但最多有一个限定可以是类。如果有一个类作为限定，它必须是限定列表中的第一个限定。

在程序清单 8-2 中，我们把 minmax 重写为一个泛型方法。这个方法可以计算泛型数组的最大值和最小值，并返回一个 Pair<T>。

### 程序清单 8-2 pair2/PairTest2.java

```
1 package pair2;
2
3 import java.time.*;
4
5 /**
6  * @version 1.02 2015-06-21
7  * @author Cay Horstmann
8 */
9 public class PairTest2
10 {
11     public static void main(String[] args)
12     {
13         LocalDate[] birthdays =
14             {
15                 LocalDate.of(1906, 12, 9), // G. Hopper
16                 LocalDate.of(1919, 3, 18), // Grace Hopper
17                 LocalDate.of(1920, 1, 26), // Alan Turing
18                 LocalDate.of(1928, 6, 28), // Steve Jobs
19                 LocalDate.of(1936, 12, 28), // Bill Gates
20                 LocalDate.of(1955, 9, 1), // Steve Jobs
21                 LocalDate.of(1956, 12, 30), // Steve Jobs
22                 LocalDate.of(1957, 1, 1), // Steve Jobs
23                 LocalDate.of(1962, 1, 24), // Steve Jobs
24                 LocalDate.of(1964, 1, 24), // Steve Jobs
25                 LocalDate.of(1965, 1, 24), // Steve Jobs
26                 LocalDate.of(1967, 1, 24), // Steve Jobs
27                 LocalDate.of(1971, 1, 24), // Steve Jobs
28                 LocalDate.of(1972, 1, 24), // Steve Jobs
29                 LocalDate.of(1972, 1, 24), // Steve Jobs
30                 LocalDate.of(1972, 1, 24), // Steve Jobs
31                 LocalDate.of(1972, 1, 24), // Steve Jobs
32                 LocalDate.of(1972, 1, 24), // Steve Jobs
33                 LocalDate.of(1972, 1, 24), // Steve Jobs
34                 LocalDate.of(1972, 1, 24), // Steve Jobs
35                 LocalDate.of(1972, 1, 24), // Steve Jobs
36                 LocalDate.of(1972, 1, 24), // Steve Jobs
37                 LocalDate.of(1972, 1, 24), // Steve Jobs
38                 LocalDate.of(1972, 1, 24), // Steve Jobs
39                 LocalDate.of(1972, 1, 24), // Steve Jobs
40                 LocalDate.of(1972, 1, 24), // Steve Jobs
41                 LocalDate.of(1972, 1, 24), // Steve Jobs
42                 LocalDate.of(1972, 1, 24), // Steve Jobs
43                 LocalDate.of(1972, 1, 24), // Steve Jobs
44                 LocalDate.of(1972, 1, 24), // Steve Jobs
45                 LocalDate.of(1972, 1, 24), // Steve Jobs
46                 LocalDate.of(1972, 1, 24), // Steve Jobs
47                 LocalDate.of(1972, 1, 24), // Steve Jobs
48                 LocalDate.of(1972, 1, 24), // Steve Jobs
49                 LocalDate.of(1972, 1, 24), // Steve Jobs
50                 LocalDate.of(1972, 1, 24), // Steve Jobs
51                 LocalDate.of(1972, 1, 24), // Steve Jobs
52                 LocalDate.of(1972, 1, 24), // Steve Jobs
53                 LocalDate.of(1972, 1, 24), // Steve Jobs
54                 LocalDate.of(1972, 1, 24), // Steve Jobs
55                 LocalDate.of(1972, 1, 24), // Steve Jobs
56                 LocalDate.of(1972, 1, 24), // Steve Jobs
57                 LocalDate.of(1972, 1, 24), // Steve Jobs
58                 LocalDate.of(1972, 1, 24), // Steve Jobs
59                 LocalDate.of(1972, 1, 24), // Steve Jobs
60                 LocalDate.of(1972, 1, 24), // Steve Jobs
61                 LocalDate.of(1972, 1, 24), // Steve Jobs
62                 LocalDate.of(1972, 1, 24), // Steve Jobs
63                 LocalDate.of(1972, 1, 24), // Steve Jobs
64                 LocalDate.of(1972, 1, 24), // Steve Jobs
65                 LocalDate.of(1972, 1, 24), // Steve Jobs
66                 LocalDate.of(1972, 1, 24), // Steve Jobs
67                 LocalDate.of(1972, 1, 24), // Steve Jobs
68                 LocalDate.of(1972, 1, 24), // Steve Jobs
69                 LocalDate.of(1972, 1, 24), // Steve Jobs
70                 LocalDate.of(1972, 1, 24), // Steve Jobs
71                 LocalDate.of(1972, 1, 24), // Steve Jobs
72                 LocalDate.of(1972, 1, 24), // Steve Jobs
73                 LocalDate.of(1972, 1, 24), // Steve Jobs
74                 LocalDate.of(1972, 1, 24), // Steve Jobs
75                 LocalDate.of(1972, 1, 24), // Steve Jobs
76                 LocalDate.of(1972, 1, 24), // Steve Jobs
77                 LocalDate.of(1972, 1, 24), // Steve Jobs
78                 LocalDate.of(1972, 1, 24), // Steve Jobs
79                 LocalDate.of(1972, 1, 24), // Steve Jobs
80                 LocalDate.of(1972, 1, 24), // Steve Jobs
81                 LocalDate.of(1972, 1, 24), // Steve Jobs
82                 LocalDate.of(1972, 1, 24), // Steve Jobs
83                 LocalDate.of(1972, 1, 24), // Steve Jobs
84                 LocalDate.of(1972, 1, 24), // Steve Jobs
85                 LocalDate.of(1972, 1, 24), // Steve Jobs
86                 LocalDate.of(1972, 1, 24), // Steve Jobs
87                 LocalDate.of(1972, 1, 24), // Steve Jobs
88                 LocalDate.of(1972, 1, 24), // Steve Jobs
89                 LocalDate.of(1972, 1, 24), // Steve Jobs
90                 LocalDate.of(1972, 1, 24), // Steve Jobs
91                 LocalDate.of(1972, 1, 24), // Steve Jobs
92                 LocalDate.of(1972, 1, 24), // Steve Jobs
93                 LocalDate.of(1972, 1, 24), // Steve Jobs
94                 LocalDate.of(1972, 1, 24), // Steve Jobs
95                 LocalDate.of(1972, 1, 24), // Steve Jobs
96                 LocalDate.of(1972, 1, 24), // Steve Jobs
97                 LocalDate.of(1972, 1, 24), // Steve Jobs
98                 LocalDate.of(1972, 1, 24), // Steve Jobs
99                 LocalDate.of(1972, 1, 24), // Steve Jobs
100                LocalDate.of(1972, 1, 24), // Steve Jobs
101                LocalDate.of(1972, 1, 24), // Steve Jobs
102                LocalDate.of(1972, 1, 24), // Steve Jobs
103                LocalDate.of(1972, 1, 24), // Steve Jobs
104                LocalDate.of(1972, 1, 24), // Steve Jobs
105                LocalDate.of(1972, 1, 24), // Steve Jobs
106                LocalDate.of(1972, 1, 24), // Steve Jobs
107                LocalDate.of(1972, 1, 24), // Steve Jobs
108                LocalDate.of(1972, 1, 24), // Steve Jobs
109                LocalDate.of(1972, 1, 24), // Steve Jobs
110                LocalDate.of(1972, 1, 24), // Steve Jobs
111                LocalDate.of(1972, 1, 24), // Steve Jobs
112                LocalDate.of(1972, 1, 24), // Steve Jobs
113                LocalDate.of(1972, 1, 24), // Steve Jobs
114                LocalDate.of(1972, 1, 24), // Steve Jobs
115                LocalDate.of(1972, 1, 24), // Steve Jobs
116                LocalDate.of(1972, 1, 24), // Steve Jobs
117                LocalDate.of(1972, 1, 24), // Steve Jobs
118                LocalDate.of(1972, 1, 24), // Steve Jobs
119                LocalDate.of(1972, 1, 24), // Steve Jobs
120                LocalDate.of(1972, 1, 24), // Steve Jobs
121                LocalDate.of(1972, 1, 24), // Steve Jobs
122                LocalDate.of(1972, 1, 24), // Steve Jobs
123                LocalDate.of(1972, 1, 24), // Steve Jobs
124                LocalDate.of(1972, 1, 24), // Steve Jobs
125                LocalDate.of(1972, 1, 24), // Steve Jobs
126                LocalDate.of(1972, 1, 24), // Steve Jobs
127                LocalDate.of(1972, 1, 24), // Steve Jobs
128                LocalDate.of(1972, 1, 24), // Steve Jobs
129                LocalDate.of(1972, 1, 24), // Steve Jobs
130                LocalDate.of(1972, 1, 24), // Steve Jobs
131                LocalDate.of(1972, 1, 24), // Steve Jobs
132                LocalDate.of(1972, 1, 24), // Steve Jobs
133                LocalDate.of(1972, 1, 24), // Steve Jobs
134                LocalDate.of(1972, 1, 24), // Steve Jobs
135                LocalDate.of(1972, 1, 24), // Steve Jobs
136                LocalDate.of(1972, 1, 24), // Steve Jobs
137                LocalDate.of(1972, 1, 24), // Steve Jobs
138                LocalDate.of(1972, 1, 24), // Steve Jobs
139                LocalDate.of(1972, 1, 24), // Steve Jobs
140                LocalDate.of(1972, 1, 24), // Steve Jobs
141                LocalDate.of(1972, 1, 24), // Steve Jobs
142                LocalDate.of(1972, 1, 24), // Steve Jobs
143                LocalDate.of(1972, 1, 24), // Steve Jobs
144                LocalDate.of(1972, 1, 24), // Steve Jobs
145                LocalDate.of(1972, 1, 24), // Steve Jobs
146                LocalDate.of(1972, 1, 24), // Steve Jobs
147                LocalDate.of(1972, 1, 24), // Steve Jobs
148                LocalDate.of(1972, 1, 24), // Steve Jobs
149                LocalDate.of(1972, 1, 24), // Steve Jobs
150                LocalDate.of(1972, 1, 24), // Steve Jobs
151                LocalDate.of(1972, 1, 24), // Steve Jobs
152                LocalDate.of(1972, 1, 24), // Steve Jobs
153                LocalDate.of(1972, 1, 24), // Steve Jobs
154                LocalDate.of(1972, 1, 24), // Steve Jobs
155                LocalDate.of(1972, 1, 24), // Steve Jobs
156                LocalDate.of(1972, 1, 24), // Steve Jobs
157                LocalDate.of(1972, 1, 24), // Steve Jobs
158                LocalDate.of(1972, 1, 24), // Steve Jobs
159                LocalDate.of(1972, 1, 24), // Steve Jobs
160                LocalDate.of(1972, 1, 24), // Steve Jobs
161                LocalDate.of(1972, 1, 24), // Steve Jobs
162                LocalDate.of(1972, 1, 24), // Steve Jobs
163                LocalDate.of(1972, 1, 24), // Steve Jobs
164                LocalDate.of(1972, 1, 24), // Steve Jobs
165                LocalDate.of(1972, 1, 24), // Steve Jobs
166                LocalDate.of(1972, 1, 24), // Steve Jobs
167                LocalDate.of(1972, 1, 24), // Steve Jobs
168                LocalDate.of(1972, 1, 24), // Steve Jobs
169                LocalDate.of(1972, 1, 24), // Steve Jobs
170                LocalDate.of(1972, 1, 24), // Steve Jobs
171                LocalDate.of(1972, 1, 24), // Steve Jobs
172                LocalDate.of(1972, 1, 24), // Steve Jobs
173                LocalDate.of(1972, 1, 24), // Steve Jobs
174                LocalDate.of(1972, 1, 24), // Steve Jobs
175                LocalDate.of(1972, 1, 24), // Steve Jobs
176                LocalDate.of(1972, 1, 24), // Steve Jobs
177                LocalDate.of(1972, 1, 24), // Steve Jobs
178                LocalDate.of(1972, 1, 24), // Steve Jobs
179                LocalDate.of(1972, 1, 24), // Steve Jobs
180                LocalDate.of(1972, 1, 24), // Steve Jobs
181                LocalDate.of(1972, 1, 24), // Steve Jobs
182                LocalDate.of(1972, 1, 24), // Steve Jobs
183                LocalDate.of(1972, 1, 24), // Steve Jobs
184                LocalDate.of(1972, 1, 24), // Steve Jobs
185                LocalDate.of(1972, 1, 24), // Steve Jobs
186                LocalDate.of(1972, 1, 24), // Steve Jobs
187                LocalDate.of(1972, 1, 24), // Steve Jobs
188                LocalDate.of(1972, 1, 24), // Steve Jobs
189                LocalDate.of(1972, 1, 24), // Steve Jobs
190                LocalDate.of(1972, 1, 24), // Steve Jobs
191                LocalDate.of(1972, 1, 24), // Steve Jobs
192                LocalDate.of(1972, 1, 24), // Steve Jobs
193                LocalDate.of(1972, 1, 24), // Steve Jobs
194                LocalDate.of(1972, 1, 24), // Steve Jobs
195                LocalDate.of(1972, 1, 24), // Steve Jobs
196                LocalDate.of(1972, 1, 24), // Steve Jobs
197                LocalDate.of(1972, 1, 24), // Steve Jobs
198                LocalDate.of(1972, 1, 24), // Steve Jobs
199                LocalDate.of(1972, 1, 24), // Steve Jobs
200                LocalDate.of(1972, 1, 24), // Steve Jobs
201                LocalDate.of(1972, 1, 24), // Steve Jobs
202                LocalDate.of(1972, 1, 24), // Steve Jobs
203                LocalDate.of(1972, 1, 24), // Steve Jobs
204                LocalDate.of(1972, 1, 24), // Steve Jobs
205                LocalDate.of(1972, 1, 24), // Steve Jobs
206                LocalDate.of(1972, 1, 24), // Steve Jobs
207                LocalDate.of(1972, 1, 24), // Steve Jobs
208                LocalDate.of(1972, 1, 24), // Steve Jobs
209                LocalDate.of(1972, 1, 24), // Steve Jobs
210                LocalDate.of(1972, 1, 24), // Steve Jobs
211                LocalDate.of(1972, 1, 24), // Steve Jobs
212                LocalDate.of(1972, 1, 24), // Steve Jobs
213                LocalDate.of(1972, 1, 24), // Steve Jobs
214                LocalDate.of(1972, 1, 24), // Steve Jobs
215                LocalDate.of(1972, 1, 24), // Steve Jobs
216                LocalDate.of(1972, 1, 24), // Steve Jobs
217                LocalDate.of(1972, 1, 24), // Steve Jobs
218                LocalDate.of(1972, 1, 24), // Steve Jobs
219                LocalDate.of(1972, 1, 24), // Steve Jobs
220                LocalDate.of(1972, 1, 24), // Steve Jobs
221                LocalDate.of(1972, 1, 24), // Steve Jobs
222                LocalDate.of(1972, 1, 24), // Steve Jobs
223                LocalDate.of(1972, 1, 24), // Steve Jobs
224                LocalDate.of(1972, 1, 24), // Steve Jobs
225                LocalDate.of(1972, 1, 24), // Steve Jobs
226                LocalDate.of(1972, 1, 24), // Steve Jobs
227                LocalDate.of(1972, 1, 24), // Steve Jobs
228                LocalDate.of(1972, 1, 24), // Steve Jobs
229                LocalDate.of(1972, 1, 24), // Steve Jobs
230                LocalDate.of(1972, 1, 24), // Steve Jobs
231                LocalDate.of(1972, 1, 24), // Steve Jobs
232                LocalDate.of(1972, 1, 24), // Steve Jobs
233                LocalDate.of(1972, 1, 24), // Steve Jobs
234                LocalDate.of(1972, 1, 24), // Steve Jobs
235                LocalDate.of(1972, 1, 24), // Steve Jobs
236                LocalDate.of(1972, 1, 24), // Steve Jobs
237                LocalDate.of(1972, 1, 24), // Steve Jobs
238                LocalDate.of(1972, 1, 24), // Steve Jobs
239                LocalDate.of(1972, 1, 24), // Steve Jobs
240                LocalDate.of(1972, 1, 24), // Steve Jobs
241                LocalDate.of(1972, 1, 24), // Steve Jobs
242                LocalDate.of(1972, 1, 24), // Steve Jobs
243                LocalDate.of(1972, 1, 24), // Steve Jobs
244                LocalDate.of(1972, 1, 24), // Steve Jobs
245                LocalDate.of(1972, 1, 24), // Steve Jobs
246                LocalDate.of(1972, 1, 24), // Steve Jobs
247                LocalDate.of(1972, 1, 24), // Steve Jobs
248                LocalDate.of(1972, 1, 24), // Steve Jobs
249                LocalDate.of(1972, 1, 24), // Steve Jobs
250                LocalDate.of(1972, 1, 24), // Steve Jobs
251                LocalDate.of(1972, 1, 24), // Steve Jobs
252                LocalDate.of(1972, 1, 24), // Steve Jobs
253                LocalDate.of(1972, 1, 24), // Steve Jobs
254                LocalDate.of(1972, 1, 24), // Steve Jobs
255                LocalDate.of(1972, 1, 24), // Steve Jobs
256                LocalDate.of(1972, 1, 24), // Steve Jobs
257                LocalDate.of(1972, 1, 24), // Steve Jobs
258                LocalDate.of(1972, 1, 24), // Steve Jobs
259                LocalDate.of(1972, 1, 24), // Steve Jobs
260                LocalDate.of(1972, 1, 24), // Steve Jobs
261                LocalDate.of(1972, 1, 24), // Steve Jobs
262                LocalDate.of(1972, 1, 24), // Steve Jobs
263                LocalDate.of(1972, 1, 24), // Steve Jobs
264                LocalDate.of(1972, 1, 24), // Steve Jobs
265                LocalDate.of(1972, 1, 24), // Steve Jobs
266                LocalDate.of(1972, 1, 24), // Steve Jobs
267                LocalDate.of(1972, 1, 24), // Steve Jobs
268                LocalDate.of(1972, 1, 24), // Steve Jobs
269                LocalDate.of(1972, 1, 24), // Steve Jobs
270                LocalDate.of(1972, 1, 24), // Steve Jobs
271                LocalDate.of(1972, 1, 24), // Steve Jobs
272                LocalDate.of(1972, 1, 24), // Steve Jobs
273                LocalDate.of(1972, 1, 24), // Steve Jobs
274                LocalDate.of(1972, 1, 24), // Steve Jobs
275                LocalDate.of(1972, 1, 24), // Steve Jobs
276                LocalDate.of(1972, 1, 24), // Steve Jobs
277                LocalDate.of(1972, 1, 24), // Steve Jobs
278                LocalDate.of(1972, 1, 24), // Steve Jobs
279                LocalDate.of(1972, 1, 24), // Steve Jobs
280                LocalDate.of(1972, 1, 24), // Steve Jobs
281                LocalDate.of(1972, 1, 24), // Steve Jobs
282                LocalDate.of(1972, 1, 24), // Steve Jobs
283                LocalDate.of(1972, 1, 24), // Steve Jobs
284                LocalDate.of(1972, 1, 24), // Steve Jobs
285                LocalDate.of(1972, 1, 24), // Steve Jobs
286                LocalDate.of(1972, 1, 24), // Steve Jobs
287                LocalDate.of(1972, 1, 24), // Steve Jobs
288                LocalDate.of(1972, 1, 24), // Steve Jobs
289                LocalDate.of(1972, 1, 24), // Steve Jobs
290                LocalDate.of(1972, 1, 24), // Steve Jobs
291                LocalDate.of(1972, 1, 24), // Steve Jobs
292                LocalDate.of(1972, 1, 24), // Steve Jobs
293                LocalDate.of(1972, 1, 24), // Steve Jobs
294                LocalDate.of(1972, 1, 24), // Steve Jobs
295                LocalDate.of(1972, 1, 24), // Steve Jobs
296                LocalDate.of(1972, 1, 24), // Steve Jobs
297                LocalDate.of(1972, 1, 24), // Steve Jobs
298                LocalDate.of(1972, 1, 24), // Steve Jobs
299                LocalDate.of(1972, 1, 24), // Steve Jobs
300                LocalDate.of(1972, 1, 24), // Steve Jobs
301                LocalDate.of(1972, 1, 24), // Steve Jobs
302                LocalDate.of(1972, 1, 24), // Steve Jobs
303                LocalDate.of(1972, 1, 24), // Steve Jobs
304                LocalDate.of(1972, 1, 24), // Steve Jobs
305                LocalDate.of(1972, 1, 24), // Steve Jobs
306                LocalDate.of(1972, 1, 24), // Steve Jobs
307                LocalDate.of(1972, 1, 24), // Steve Jobs
308                LocalDate.of(1972, 1, 24), // Steve Jobs
309                LocalDate.of(1972, 1, 24), // Steve Jobs
310                LocalDate.of(1972, 1, 24), // Steve Jobs
311                LocalDate.of(1972, 1, 24), // Steve Jobs
312                LocalDate.of(1972, 1, 
```

```

16     LocalDate.of(1815, 12, 10), // A. Lovelace
17     LocalDate.of(1903, 12, 3), // J. von Neumann
18     LocalDate.of(1910, 6, 22), // K. Zuse
19   };
20   Pair<LocalDate> mm = ArrayAlg.minmax(birthdays);
21   System.out.println("min = " + mm.getFirst());
22   System.out.println("max = " + mm.getSecond());
23 }
24 }
25
26 class ArrayAlg
27 {
28   /**
29    * Gets the minimum and maximum of an array of objects of type T.
30    * @param a an array of objects of type T
31    * @return a pair with the min and max values, or null if a is null or empty
32   */
33   public static <T extends Comparable> Pair<T> minmax(T[] a)
34   {
35     if (a == null || a.length == 0) return null;
36     T min = a[0];
37     T max = a[0];
38     for (int i = 1; i < a.length; i++)
39     {
40       if (min.compareTo(a[i]) > 0) min = a[i];
41       if (max.compareTo(a[i]) < 0) max = a[i];
42     }
43     return new Pair<>(min, max);
44   }
45 }

```

## 8.5 泛型代码和虚拟机

虚拟机没有泛型类型对象——所有对象都属于普通类。在泛型实现的早期版本中，甚至能够将使用泛型的程序编译为在 1.0 虚拟机上运行的类文件！在下面的小节中你会看到编译器如何“擦除”类型参数，以及这个过程对 Java 程序员有什么影响。

### 8.5.1 类型擦除

无论何时定义一个泛型类型，都会自动提供一个相应的原始类型（raw type）。这个原始类型的名字就是去掉类型参数后的泛型类型名。类型变量会被擦除（erased），并替换为其限定类型（或者，对于无限定的变量则替换为 Object）。

例如，Pair<T> 的原始类型如下所示：

```

public class Pair
{
  private Object first;
  private Object second;

  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; }
}

```

因为 T 是一个无限定的类型变量，所以直接替换为 Object。

其结果是一个普通类，就好像 Java 语言中引入泛型之前实现的类一样。

在程序中可以包含不同类型的 Pair，例如，Pair<String> 或 Pair<LocalDate>。不过擦除类型后，它们都会变成原始的 Pair 类型。

**C++ 注释：**就这点而言，Java 泛型与 C++ 模板有很大的区别。C++ 会为每个模板的实例化生成不同的类型，这一现象称为“模板代码膨胀”。Java 不受这个问题的困扰。

原始类型用第一个限定来替换类型变量，或者，如果没有给定限定，就替换为 Object。例如，类 Pair<T> 中的类型变量没有显式的限定，因此，原始类型用 Object 替换 T。假定我们声明了一个稍有不同的类型：

```

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

```

原始类型 Interval 如下所示：

```

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

```

**注释：**你可能想要知道限定切换为 class Interval<T extends Serializable & Comparable> 会发什么。如果这样做，原始类型会用 Serializable 替换 T，而且编译器会在必要时插入转换为 Comparable 的强制类型转换。为了提高效率，应该将标记（tagging）接口（即没有方法的接口）放在限定列表的末尾。

### 8.5.2 转换泛型表达式

编写一个泛型方法调用时，如果擦除了返回类型，编译器会插入强制类型转换。例如，对于下面这个语句序列：

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

`getFirst` 擦除类型后的返回类型是 `Object`。编译器自动插入转换到 `Employee` 的强制类型转换。也就是说，编译器把这个方法调用转换为两条虚拟机指令：

- 调用原始方法 `Pair.getFirst`。
- 将返回的 `Object` 类型强制转换为 `Employee` 类型。

访问一个泛型字段时也会插入强制类型转换。假设 `Pair` 类的 `first` 字段和 `second` 字段都是公共的（也许这不是一种好的编程风格，但在 Java 中是合法的）。以下表达式

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

也会在结果字节码中插入强制类型转换。

### 8.5.3 转换泛型方法

类型擦除也会出现在泛型方法中。程序员通常认为类似下面的泛型方法

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

是整个一组方法，而擦除类型之后，只剩下一个方法：

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

注意，类型参数 `T` 已经被擦除了，只留下了它的限定类型 `Comparable`。

方法的擦除带来了两个复杂问题。考虑下面这个示例：

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

日期区间是一对 `LocalDate` 对象，而且我们想覆盖这个方法来确保第二个值永远不小于第一个值。这个类擦除后变成

```
class DateInterval extends Pair // after erasure
{
    public void setSecond(LocalDate second) { . . . }
    . .
}
```

令人感到奇怪的是，还有另一个从 `Pair` 继承的 `setSecond` 方法，即

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

这显然是一个不同的方法，因为它有一个不同类型的参数——Object，而不是 LocalDate。不过，它不应该不一样。考虑下面的语句序列：

```
var interval = new DateInterval(...);
Pair<LocalDate> pair = interval; // OK--assignment to superclass
pair.setSecond(aDate);
```

这里，我们希望 setSecond 调用具有多态性，应该调用适当的方法。因为 pair 引用一个 DateInterval 对象，所以应该调用 DateInterval.setSecond。问题在于类型擦除与多态发生了冲突。为了解决这个问题，编译器在 DateInterval 类中生成一个桥方法（bridge method）：

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

要想了解为什么这样可行，请仔细跟踪以下语句的执行：

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

变量 pair 已经声明为类型 Pair<LocalDate>，并且这个类型只有一个名为 setSecond 的方法，即 setSecond(Object)。虚拟机在 pair 引用的对象上调用这个方法。这个对象是 DateInterval 类型，因而将会调用 DateInterval.setSecond(Object) 方法。这个方法是合成的桥方法。它会调用 DateInterval.setSecond(LocalDate)，这正是我们想要的。

桥方法可能会变得更奇怪。假设 DateInterval 类也覆盖了 getSecond 方法：

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

在 DateInterval 类中，有两个 getSecond 方法：

```
LocalDate getSecond() // defined in DateInterval
Object getSecond() // overrides the method defined in Pair to call the first method
```

你不能编写这样的 Java 代码（两个方法有相同的参数类型是不合法的，在这里，两个方法都没有参数）。但是，在虚拟机中，会由参数类型以及返回类型共同指定一个方法。因此，编译器可以为两个仅返回类型不同的方法生成字节码，虚拟机能够正确地处理这种情况。

**注释：**桥方法不只是用于泛型类型。第 5 章已经讲过，一个方法覆盖另一个方法时，可以指定一个更严格的返回类型，这是合法的。例如：

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

Object.clone 和 Employee.clone 方法被称为有协变的返回类型（covariant return type）。

实际上，Employee 类有两个克隆方法：

```
Employee clone() // defined above
Object clone() // synthesized bridge method, overrides Object.clone
```

合成的桥方法会调用新定义的方法。

总之，对于 Java 泛型的转换，需要记住以下几点：

- 虚拟机中没有泛型，只有普通的类和方法。
- 所有的类型参数都会替换为它们的限定类型。
- 会合成桥方法来保持多态。
- 为保持类型安全性，必要时会插入强制类型转换。

#### 8.5.4 调用遗留代码

设计 Java 泛型时，主要目标是允许泛型代码和遗留代码之间能够互操作。下面看有关遗留代码的一个具体示例。Swing 用户界面工具包提供了一个 `JSlider` 类，它的“刻度”（tick）可以定制为包含文本或图像的标签。这些标签用以下调用设置：

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

`Dictionary` 类将整数映射到标签。在 Java 5 之前，这个类实现为一个 `Object` 实例映射。Java 5 把 `Dictionary` 实现为一个泛型类，不过 `JSlider` 从未更新。此时，没有类型参数的 `Dictionary` 是一个原始类型。这里就存在兼容性问题。

填充字典时，可以使用泛型类型。

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

将 `Dictionary<Integer, Component>` 对象传递给 `setLabelTable` 时，编译器会发出一个警告。

```
slider.setLabelTable(labelTable); // warning
```

毕竟，编译器无法确定 `setLabelTable` 可能会对 `Dictionary` 对象做什么操作。这个方法可能会把字典的所有键替换为字符串。这就打破了键类型必须为 `Integer` 的承诺，未来的操作有可能导致糟糕的强制类型转换异常。

要仔细考虑这个问题，想想看 `JSlider` 到底会用 `Dictionary` 对象做什么。在这里十分清楚，`JSlider` 只读取这个信息，因此可以忽略这个警告。

现在看一个相反的情形，由一个遗留类得到一个原始类型的对象。可以将它赋给一个类型使用了泛型的变量，当然，这样做会得到一个警告。例如：

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

没关系。查看这个警告，确保标签表确实包含 `Integer` 和 `Component` 对象。当然，从来没有绝对的保证。恶意的程序员可能会在滑动条中安装一个不同的 `Dictionary`。不过，这种情况并不会比有泛型之前的情况更糟糕。最差的情况也就是程序抛出一个异常。

考虑了这个警告之后，可以使用注解（annotation）使之消失。可以对一个局部变量加注解，如下所示：

```
@SuppressWarnings("unchecked")
Dictionary<Integer, Components> labelTable = slider.getLabelTable(); // no warning
```

或者，可以对整个方法加注解，如下所示：

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

这个注解会关闭对方法中所有代码的检查。

## 8.6 限制与局限性

在下面几节中，我们将讨论使用 Java 泛型时需要考虑的一些限制。大多数限制都是由类型擦除引起的。

### 8.6.1 不能用基本类型实例化类型参数

不能用基本类型代替类型参数。因此，没有 `Pair<double>`，只有 `Pair<Double>`。当然，其原因就在于类型擦除。擦除之后，`Pair` 类含有 `Object` 类型的字段，而 `Object` 不能存储 `double` 值。

这的确令人烦恼。但是，这样做与 Java 语言中基本类型的独立状态相一致。这并不是一个致命的缺陷——只有 8 种基本类型，而且即使不能接受包装器类型（wrapper type），也可以使用单独的类和方法来处理。

### 8.6.2 运行时类型查询只适用于原始类型

虚拟机中的对象总是有一个特定的非泛型类型。因此，所有的类型查询只生成原始类型。例如，

```
if (a instanceof Pair<String>) // ERROR
```

实际上仅仅测试 `a` 是否是任意类型的一个 `Pair`。下面的测试同样如此：

```
if (a instanceof Pair<T>) // ERROR
```

或以下强制类型转换也是如此：

```
Pair<String> p = (Pair<String>) a; // warning--can only test that a is a Pair
```

为了提醒这一风险，如果试图查询一个对象是否属于某个泛型类型，你会得到一个编译器错误（使用 `instanceof` 时），或者得到一个警告（使用强制类型转换时）。

同样的道理，`getClass` 方法总是返回原始类型。例如：

```
Pair<String> stringPair = . . .;
Pair<Employee> employeePair = . . .;
if (stringPair.getClass() == employeePair.getClass()) // they are equal
```

这个比较的结果是 `true`，因为两个 `getClass` 调用都返回 `Pair.class`。

### 8.6.3 不能创建参数化类型的数组

不能实例化参数化类型的数组，例如：

```
var table = new Pair<String>[10]; // ERROR
```

这有什么问题呢？擦除之后，`table` 的类型是 `Pair[]`。可以把它转换为 `Object[]`：

```
Object[] objarray = table;
```

数组会记住它的元素类型，如果试图存储类型不正确的元素，就会抛出一个 `ArrayStoreException` 异常：

```
objarray[0] = "Hello"; // ERROR--component type is Pair
```

不过对于泛型类型，擦除会使这种机制无效。以下赋值

```
objarray[0] = new Pair<Employee>();
```

尽管能够通过数组存储的检查，但仍会导致一个类型错误。出于这个原因，不允许创建参数化类型的数组。

需要说明的是，只是不允许创建这些数组，而声明类型为 `Pair<String>[]` 的变量仍是合法的。不过不能用 `new Pair<String>[10]` 初始化这个变量。

 **注释：**可以声明通配类型的数组，然后进行强制类型转换：

```
var table = (Pair<String>[]) new Pair<?>[10];
```

结果将是不安全的。如果在 `table[0]` 中存储一个 `Pair<Employee>`，然后对 `table[0].getFirst()` 调用一个 `String` 方法，会得到一个 `ClassCastException` 异常。

 **提示：**如果需要收集参数化类型对象，可以直接使用 `ArrayList<Pair<String>>` 很安全也很有效。

#### 8.6.4 Varargs 警告

上一节中已经了解到，Java 不支持泛型类型的数组。这一节中我们再来讨论一个相关的问题：向参数个数可变的方法传递一个泛型类型的实例。

考虑下面这个简单的方法，它的参数个数是可变的：

```
public static <T> void addAll(Collection<T> coll, T... ts)
{
    for (T t : ts) coll.add(t);
}
```

回忆一下，实际上参数 `ts` 是一个数组，包含提供的所有实参。

现在考虑以下调用：

```
Collection<Pair<String>> table = . . .;
Pair<String> pair1 = . . .;
Pair<String> pair2 = . . .;
addAll(table, pair1, pair2);
```

为了调用这个方法，Java 虚拟机必须建立一个 `Pair<String>` 数组，这就违反了规则。不过，对于这种情况，规则有所放松，你只会得到一个警告，而不是错误。

可以采用两种方法来抑制这个警告。一种方法是为包含 `addAll` 调用的方法增加注解 `@SuppressWarnings("unchecked")`。或者在 Java 7 中，还可以用 `@SafeVarargs` 直接注解 `addAll` 方法：

```
@SafeVarargs
public static <T> void addAll(Collection<T> coll, T... ts)
```

现在就可以提供泛型类型来调用这个方法了。对于任何只需要读取参数数组元素的方法（这肯定是最常见的情况），都可以使用这个注解。

`@SafeVarargs` 只能用于声明为 `static`、`final` 或（Java 9 中）`private` 的构造器和方法。所有其他方法都可能被覆盖，这会使这个注解失去意义。

 **注释：**可以使用 `@SafeVarargs` 注解来消除创建泛型数组的有关限制，方法如下：

```
@SafeVarargs static <E> E[] array(E... array) { return array; }
```

现在可以调用：

```
Pair<String>[] table = array(pair1, pair2);
```

这看起来很方便，不过隐藏着危险。以下代码

```
Object[] objarray = table;
objarray[0] = new Pair<Employee>();
```

能顺利运行而不会出现 `ArrayStoreException` 异常（因为数组存储只会检查擦除后的类型），但在处理 `table[0]` 时，你会在别处得到一个异常。

### 8.6.5 不能实例化类型变量

不能在类似 `new T(...)` 的表达式中使用类型变量。例如，下面的 `Pair<T>` 构造器就是非法的：

```
public Pair() { first = new T(); second = new T(); } // ERROR
```

类型擦除将 `T` 变成 `Object`，而你肯定不希望调用 `new Object()`。

在 Java 8 之后，最好的解决办法是让调用者提供一个构造器表达式。例如：

```
Pair<String> p = Pair.makePair(String::new);
```

`makePair` 方法接收一个 `Supplier<T>`，这是一个函数式接口，表示一个无参数而且返回类型为 `T` 的函数：

```
public static <T> Pair<T> makePair(Supplier<T> constr)
{
    return new Pair<T>(constr.get(), constr.get());
}
```

比较传统的解决方法是通过反射调用 `Constructor.newInstance` 方法来构造泛型对象。

遗憾的是，细节有点复杂。不能如下调用：

```
first = T.class.getConstructor().newInstance(); // ERROR
```

表达式 `T.class` 是不合法的，因为它会擦除为 `Object.class`。必须适当地设计 API 以便得到一个 `Class` 对象，如下所示：

```
public static <T> Pair<T> makePair(Class<T> cl)
{
    try
```

```

    {
        return new Pair<T>(cl.getConstructor().newInstance(),
            cl.getConstructor().newInstance());
    }
    catch (Exception e) { return null; }
}

```

这个方法可以如下调用：

```
Pair<String> p = Pair.makePair(String.class);
```

注意，`Class`类本身是泛型的。例如，`String.class`是`Class<String>`的一个实例（事实上，它是唯一的实例）。因此，`makePair`方法能够推断出所建立的对组（pair）的类型。

### 8.6.6 不能构造泛型数组

就像不能实例化泛型实例一样，也不能实例化数组。不过原因有所不同，毕竟数组可以填充`null`值，看上去好像可以安全地构造数组。不过，数组本身也带有类型，用来监控虚拟机中的数组存储。这个类型会被擦除。例如，考虑下面的例子：

```

public static <T extends Comparable> T[] minmax(T... a)
{
    T[] mm = new T[2]; // ERROR
    ...
}

```

类型擦除会让这个方法总是构造`Comparable[2]`数组。

如果数组仅仅作为一个类的私有实例字段，那么可以将这个数组的元素类型声明为擦除后的类型并使用强制类型转换。例如，`ArrayList`类可以如下实现：

```

public class ArrayList<E>
{
    private Object[] elements;
    ...
    @SuppressWarnings("unchecked") public E get(int n) { return (E) elements[n]; }
    public void set(int n, E e) { elements[n] = e; } // no cast needed
}

```

但实际的实现没有这么清晰：

```

public class ArrayList<E>
{
    private E[] elements;
    ...
    public ArrayList() { elements = (E[]) new Object[10]; }
}

```

这里，强制类型转换`E[]`是一个假象，而类型擦除使其无法察觉。

这个技术并不适用于我们的`minmax`方法，因为`minmax`方法返回一个`T[]`数组，如果我们对类型“作假”，使用擦除后的类型，就会得到运行时错误结果。假设实现以下代码：

```

public static <T extends Comparable> T[] minmax(T... a)
{
    var result = new Comparable[2]; // array of erased type
}

```

```

    ...
    return (T[]) result; // compiles with warning
}

```

以下调用

```
String[] names = ArrayAlg.minmax("Tom", "Dick", "Harry");
```

编译时不会有任何警告。当方法返回后 Comparable[] 引用被强制转换为 String[] 时，将会出现 ClassCastException 异常。

在这种情况下，最好让用户提供一个数组构造器表达式：

```
String[] names = ArrayAlg.minmax(String[]::new, "Tom", "Dick", "Harry");
```

构造器表达式 String::new 指示一个函数，给定所需的长度，会构造一个指定长度的 String 数组。minmax 方法使用这个参数生成一个有正确类型的数组：

```

public static <T extends Comparable> T[] minmax(IntFunction<T[]> constr, T... a)
{
    T[] result = constr.apply(2);
    ...
}

```

比较老式的方法是利用反射，并调用 Array.newInstance：

```

public static <T extends Comparable> T[] minmax(T... a)
{
    var result = (T[]) Array.newInstance(a.getClass().getComponentType(), 2);
    ...
}

```

ArrayList 类的 toArray 方法就没有这么幸运。它需要生成一个 T[] 数组，但没有元素类型。因此，有下面两种不同的形式：

```
Object[] toArray()
T[] toArray(T[] result)
```

第二个方法接收一个数组参数。如果数组足够大，就使用这个数组。否则，用 result 的元素类型构造一个足够大的新数组。

### 8.6.7 泛型类的静态上下文中类型变量无效

不能在静态字段或方法中引用类型变量。例如，下面的做法看起来很聪明，但实际上行不通：

```

public class Singleton<T>
{
    private static T singleInstance; // ERROR

    public static T getSingleInstance() // ERROR
    {
        if (singleInstance == null) construct new instance of T
        return singleInstance;
    }
}

```

如果这样可行，程序就可以声明一个 Singleton<Random> 以共享一个随机数生成器，另外声明一个 Singleton<JFileChooser> 以共享一个文件选择器对话框。但是，这样是行不通的。类型

擦除之后，只剩下 Singleton 类，它只包含一个 singleInstance 字段。因此，带有类型变量的静态字段和方法是完全非法的。

### 8.6.8 不能抛出或捕获泛型类的实例

既不能抛出也不能捕获泛型类的对象。实际上，甚至泛型类扩展 Throwable 都是不合法的。例如，以下定义就不能编译：

```
public class Problem<T> extends Exception { /* . . . */ }
    // ERROR--can't extend Throwable
```

catch 子句中不能使用类型变量。例如，以下方法将不能编译：

```
public static <T extends Throwable> void doWork(Class<T> t)
{
    try
    {
        do work
    }
    catch (T e) // ERROR--can't catch type variable
    {
        Logger.getGlobal().info(. . .);
    }
}
```

不过，在异常规范中使用类型变量是允许的。以下方法是合法的：

```
public static <T extends Throwable> void doWork(T t) throws T // OK
{
    try
    {
        do work
    }
    catch (Throwable realCause)
    {
        t.initCause(realCause);
        throw t;
    }
}
```

### 8.6.9 可以取消对检查型异常的检查

Java 异常处理的一个基本原则是，必须为所有检查型异常提供一个处理器。不过可以利用泛型取消这个机制。关键在于以下方法：

```
@SuppressWarnings("unchecked")
static <T extends Throwable> void throwAs(Throwable t) throws T
{
    throw (T) t;
}
```

假设这个方法包含在接口 Task 中，如果有一个检查型异常 e，并调用

```
Task.<RuntimeException>throwAs(e);
```

编译器就会认为 e 是一个非检查型异常。以下代码会把所有异常都转换为编译器所认为的非检查型异常：

```

try
{
    do work
}
catch (Throwable t)
{
    Task.<RuntimeException>throwAs(t);
}

```

下面使用这个技术解决一个棘手的问题。要在一个线程中运行代码，需要把代码放在一个实现了 Runnable 接口的类的 run 方法中。不过这个方法不允许抛出检查型异常。我们将提供一个从 Task 到 Runnable 的适配器，它的 run 方法可以抛出任意的异常。

```

interface Task
{
    void run() throws Exception;

    @SuppressWarnings("unchecked")
    static <T extends Throwable> void throwAs(Throwable t) throws T
    {
        throw (T) t;
    }

    static Runnable asRunnable(Task task)
    {
        return () ->
        {
            try
            {
                task.run();
            }
            catch (Exception e)
            {
                Task.<RuntimeException>throwAs(e);
            }
        };
    }
}

```

例如，以下程序运行了一个线程，它会抛出一个检查型异常。

```

public class Test
{
    public static void main(String[] args)
    {
        var thread = new Thread(Task.asRunnable(() ->
        {
            Thread.sleep(1000);
            System.out.println("Hello, World!");
            throw new Exception("Check this out!");
        }));
        thread.start();
    }
}

```

Thread.sleep 方法声明为抛出一个 InterruptedException，我们不再需要捕获这个异常。因为我们没有中断这个线程，所以不会抛出这个异常。不过，程序会抛出一个检查型异常。运行

程序时，你会得到一个栈轨迹。

这有什么意义呢？正常情况下，你必须捕获一个 `Runnable` 的 `run` 方法中的所有检查型异常，把它们“包装”到非检查型异常中，因为 `run` 方法声明为不抛出任何检查型异常。

不过在这里并没有做这种“包装”。我们只是抛出异常，并“哄骗”编译器，让它相信这不是一个检查型异常。

通过使用泛型类、擦除和 `@SuppressWarnings` 注解，我们就能消除 Java 类型系统的一个基本限制。

### 8.6.10 注意擦除后的冲突

擦除泛型类型后，不允许创建引发冲突的条件。下面来看一个示例。假定为 `Pair` 类增加一个 `equals` 方法，如下所示：

```
public class Pair<T>
{
    public boolean equals(T value) { return first.equals(value) && second.equals(value); }
    ...
}
```

考虑一个 `Pair<String>`。从概念上讲，它有两个 `equals` 方法：

```
boolean equals(String) // defined in Pair<T>
boolean equals(Object) // inherited from Object
```

但是，直觉把我们引入歧途。方法

```
boolean equals(T)
```

擦除后就是

```
boolean equals(Object)
```

这会与 `Object.equals` 方法发生冲突。

当然，补救的办法是重新命名引发冲突的方法。

泛型规范还指出了另外一个规则：“为了支持擦除转换，我们要施加一个限制：倘若两个接口类型是同一接口的不同参数化，一个类或类型变量就不能同时作为这两个接口类型的子类。”例如，下面的代码是非法的：

```
class Employee implements Comparable<Employee> { . . . }
class Manager extends Employee implements Comparable<Manager> { . . . } // ERROR
```

如果以上代码可行，`Manager` 就会实现 `Comparable<Employee>` 和 `Comparable<Manager>`，而它们是同一接口的不同参数化。

这一限制与类型擦除的关系并不十分明显。毕竟，以下非泛型版本是合法的。

```
class Employee implements Comparable { . . . }
class Manager extends Employee implements Comparable { . . . }
```

其原因非常微妙，这有可能与合成的桥方法产生冲突。实现了 `Comparable<X>` 的类会获得一个桥方法：

```
public int compareTo(Object other) { return compareTo((X) other); }
```

不可能对不同的类型 `X` 有两个这样的方法。

## 8.7 泛型类型的继承规则

使用泛型类时，需要了解有关继承和子类型的一些规则。下面先从许多程序员感觉不太直观的情况开始介绍。考虑一个类和一个子类，如 Employee 和 Manager。Pair<Manager> 是 Pair<Employee> 的一个子类型吗？或许人们会感到奇怪，答案是“不是”。例如，下面的代码将不能成功编译：

```
Pair<Employee> buddies = new Pair<Manager>(ceo, cfo); // illegal
```

一般来讲，无论 S 与 T 有什么关系，Pair<S> 与 Pair<T> 都没有任何关系（如图 8-1 所示）。

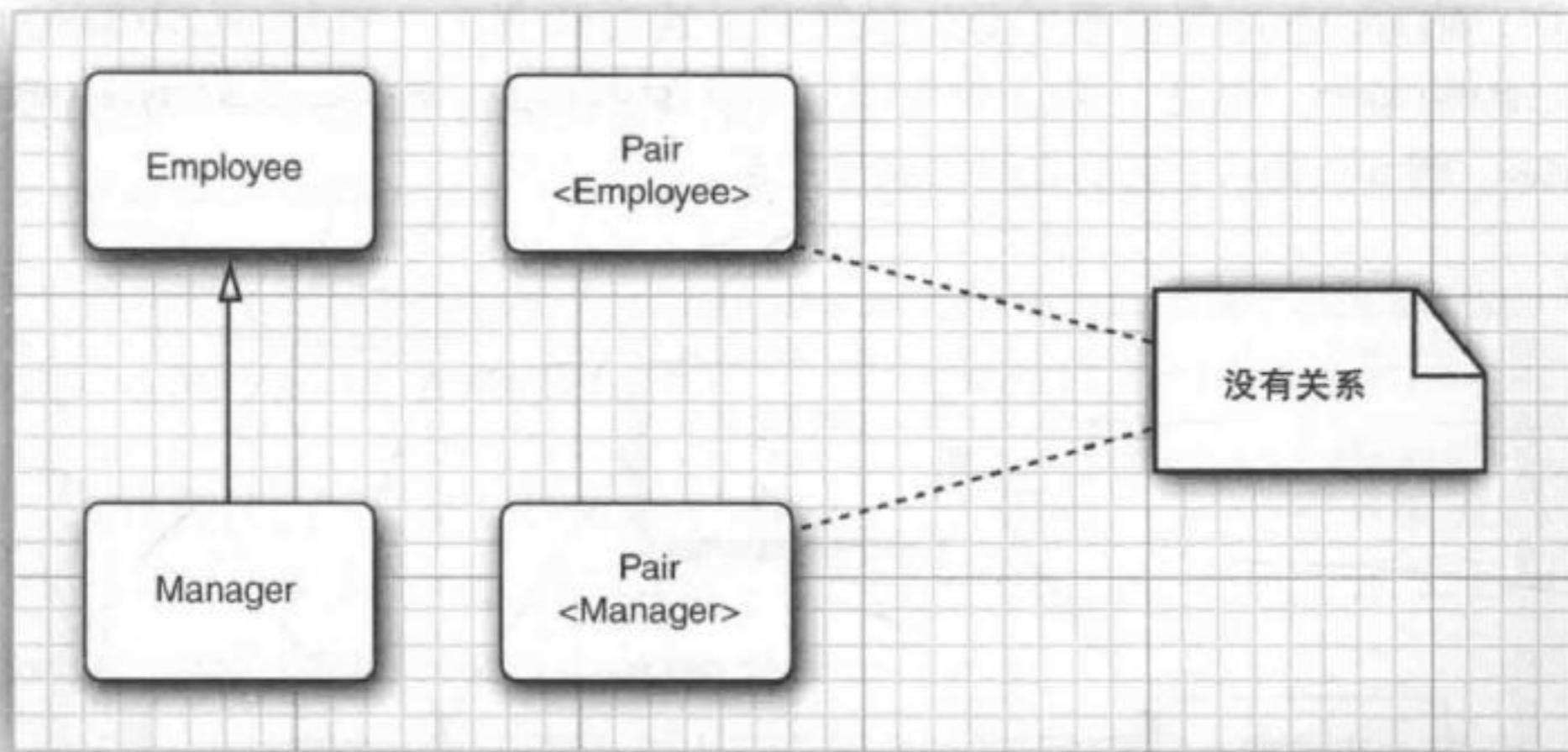


图 8-1 pair 类之间没有继承关系

这看起来是一个很严格的限制，不过对于类型安全非常必要。假设允许将 Pair<Manager> 转换为 Pair<Employee>。考虑下面的代码：

```
var managerBuddies = new Pair<Manager>(ceo, cfo);
Pair<Employee> employeeBuddies = managerBuddies; // illegal, but suppose it wasn't
employeeBuddies.setFirst(lowlyEmployee);
```

显然，最后一句是合法的。但是 employeeBuddies 和 managerBuddies 引用了同样的对象。现在我们会把 CFO 和一个底层员工组成一对，这对于 Pair<Manager> 来说应该是不可能的。

**注释：**前面看到的是泛型类型与 Java 数组之间的一个重要区别。可以将一个 Manager[] 数组赋给一个类型为 Employee[] 的变量：

```
Manager[] managerBuddies = { ceo, cfo };
Employee[] employeeBuddies = managerBuddies; // OK
```

不过，数组有特别的保护。如果试图将一个底层员工存储到 employeeBuddies[0]，虚拟机将会抛出 ArrayStoreException 异常。

总是可以将参数化类型转换为一个原始类型。例如，`Pair<Employee>` 是原始类型 `Pair` 的一个子类型。在与遗留代码交互时，这个转换非常必要。

转换成原始类型会导致类型错误吗？很遗憾，会！看一看下面这个示例：

```
var managerBuddies = new Pair<Manager>(ceo, cfo);
Pair rawBuddies = managerBuddies; // OK
rawBuddies.setFirst(new File("...")); // only a compile-time warning
```

听起来有点吓人。但是，请记住现在的状况不会比更老版本的 Java 更糟糕。虚拟机的安全性还没有到生死攸关的程度。当使用 `getFirst` 获得外来对象并赋给 `Manager` 变量时，与以往一样，会抛出 `ClassCastException` 异常。这里失去的只是泛型程序设计提供的附加安全性。

最后，泛型类可以扩展或实现其他的泛型类。就这一点而言，它们与普通的类没有什么区别。例如，`ArrayList<T>` 类实现了 `List<T>` 接口。这意味着，一个 `ArrayList<Manager>` 可以转换为一个 `List<Manager>`。但是，如前面所见，`ArrayList<Manager>` 不是一个 `ArrayList<Employee>` 或 `List<Employee>`。图 8-2 展示了它们之间的这种关系。

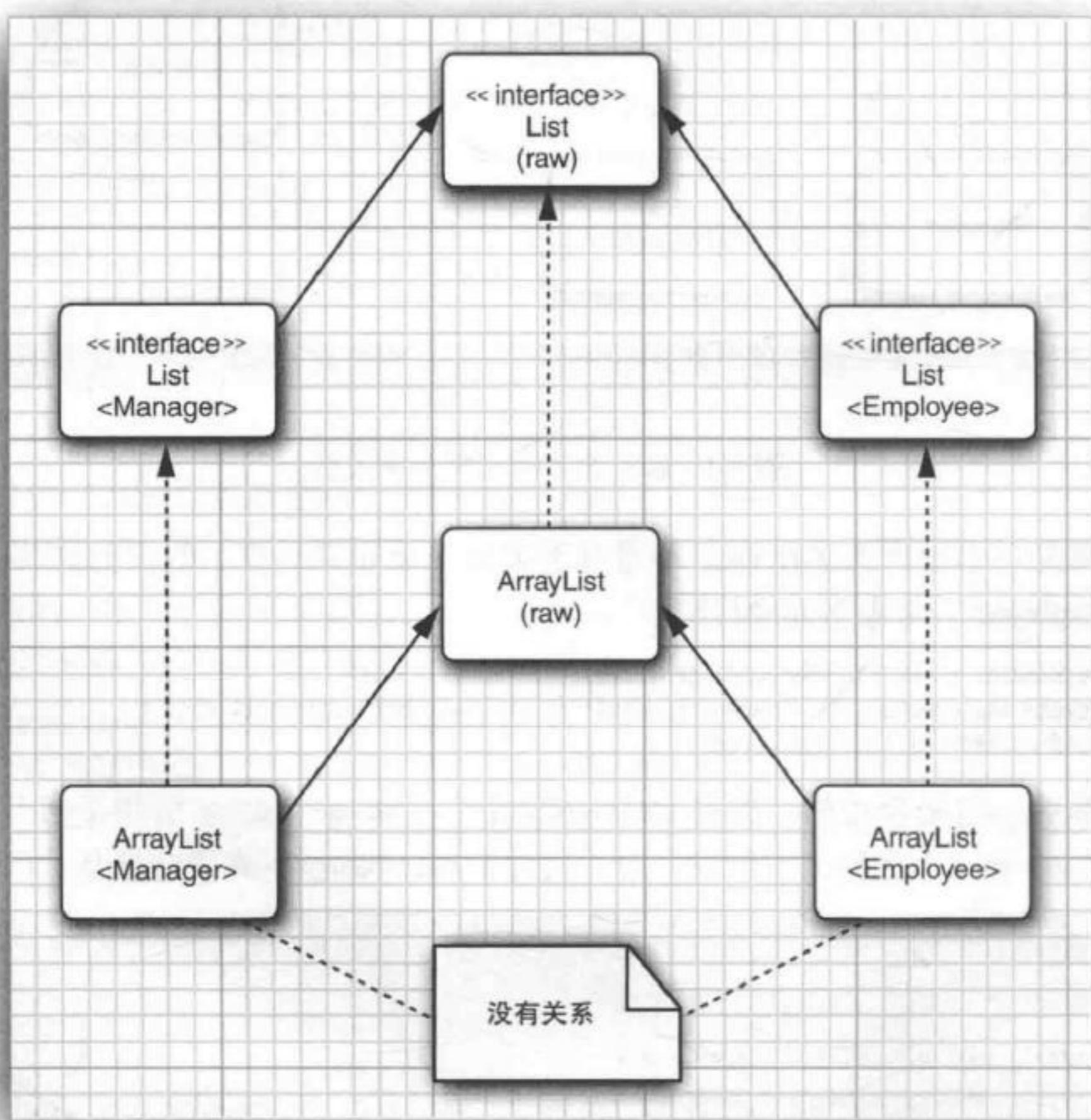


图 8-2 泛型列表类型之间的子类型关系

## 8.8 通配符类型

严格的泛型类型系统使用起来并不那么令人愉快，类型系统的研究人员知道这一点已经有一段时间了。Java 的设计者发明了一种巧妙的（但很安全的）“逃生出口”：通配符类型（wildcard type）。下面几小节会介绍如何使用通配符。

### 8.8.1 通配符概念

在通配符类型中，允许类型参数变化。例如，通配符类型

```
Pair<? extends Employee>
```

表示任何泛型 Pair 类型，它的类型参数是 Employee 的子类，如 Pair<Manager>，但不能是 Pair<String>。

假设要编写一个打印员工对的方法，如下所示：

```
public static void printBuddies(Pair<Employee> p)
{
    Employee first = p.getFirst();
    Employee second = p.getSecond();
    System.out.println(first.getName() + " and " + second.getName() + " are buddies.");
}
```

正如前面讲到的，不能将 Pair<Manager> 传递给这个方法，这一点很有限制。不过解决的方法很简单——可以使用一个通配符类型：

```
public static void printBuddies(Pair<? extends Employee> p)
```

类型 Pair<Manager> 是 Pair<? extends Employee> 的子类型（如图 8-3 所示）。

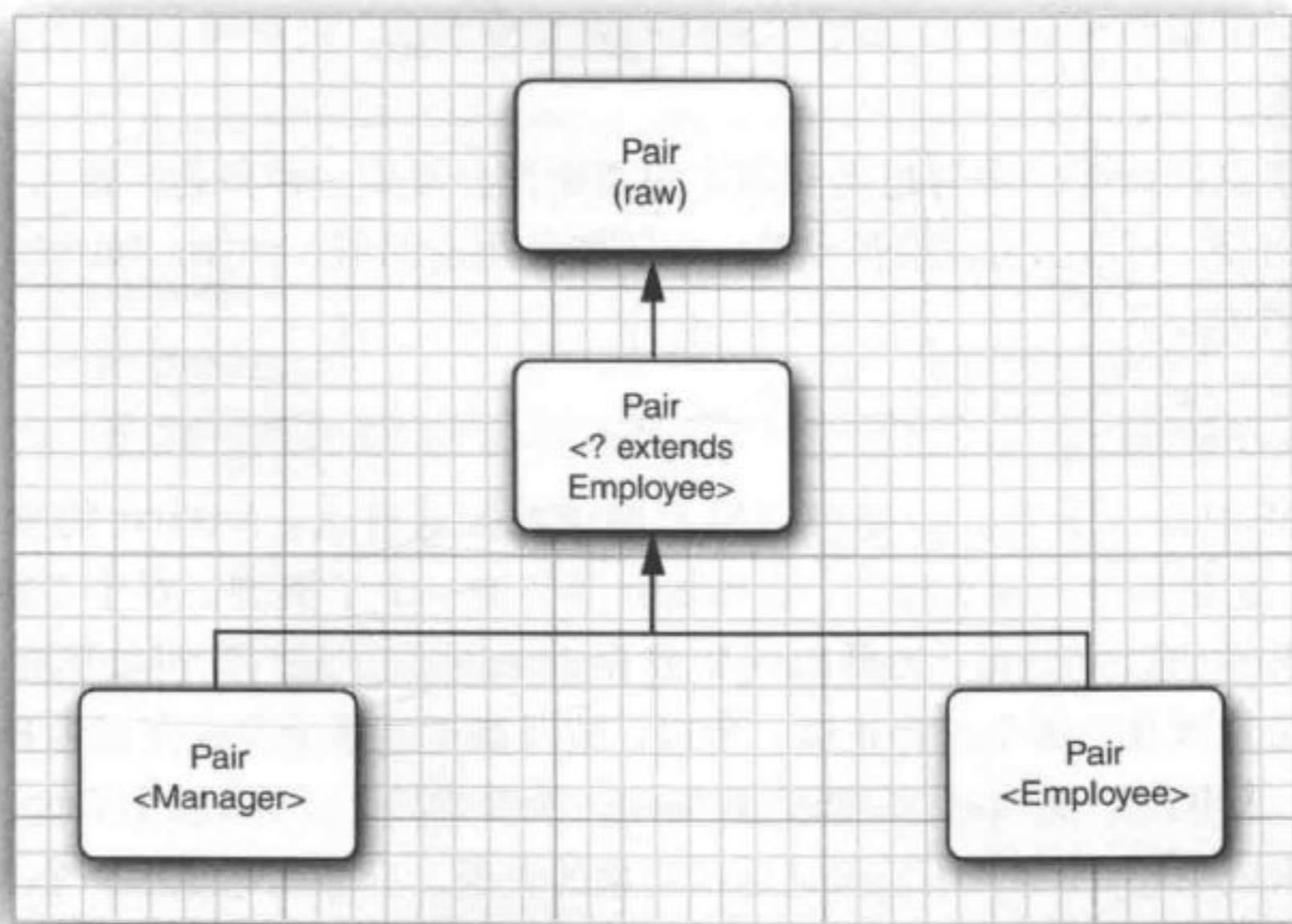


图 8-3 使用通配符的子类型关系

使用通配符会通过 `Pair<? extends Employee>` 的引用破坏 `Pair<Manager>` 吗？

```
var managerBuddies = new Pair<Manager>(ceo, cfo);
Pair<? extends Employee> wildcardBuddies = managerBuddies; // OK
wildcardBuddies.setFirst(lowlyEmployee); // compile-time error
```

这不可能引起破坏。对 `setFirst` 的调用有一个类型错误。要了解其中的缘由，请仔细看一看类型 `Pair<? extends Employee>`。它的方法如下：

```
? extends Employee getFirst()
void setFirst(? extends Employee)
```

不可能调用 `setFirst` 方法。考虑调用 `wildcardBuddies.setFirst(lowlyEmployee)`，编译器知道 `setFirst` 的参数有某个特定的类型，这个类型扩展了 `Employee`。这个特定类型是 `Employee` 吗？是 `Manager` 吗？还是另外某个子类？编译器无法知道。因此，编译器不能接受 `lowlyEmployee`。出于同样的原因，调用 `wildcardBuddies.setFirst(cio)`（其中 `cio` 是一个 `Manager` 实例）也会出错。除了 `null`，编译器必须拒绝传入 `setFirst` 的所有参数。

`getFirst` 方法则可以继续工作。`getFirst` 的返回值是某个特定类型的实例，这是 `Employee` 的一个子类型。编译器不知道这个特定类型是什么，但它可以保证对 `Employee` 引用的赋值是安全的。

这就是引入有限定的通配符的关键之处。现在我们已经有办法区分安全的访问器方法和不安全的更改器方法了。

### 8.8.2 通配符的超类型限定

通配符限定与类型变量限定十分类似，但是，它们还有一个附加的能力，你可以指定一个超类型限定（supertype bound），如下所示：

```
? super Manager
```

这个通配符限制为 `Manager` 的所有超类型。（真是很幸运，已有的 `super` 关键字十分准确地描述了这种关系。）

为什么想要这样做呢？带有超类型限定的通配符会提供一种行为，这与 8.8 节介绍的通配符行为正好相反。可以为方法提供参数，但不能使用返回值。例如，`Pair<? super Manager>` 有一些方法可以描述如下：

```
void setFirst(? super Manager)
? super Manager getFirst()
```

这不是真正的 Java 语法，但是可以展示编译器知道什么。`setFirst` 的参数类型表示为 `? super Manager`，这是某个特定类型 `T`，而 `Manager` 是 `T` 的一个子类型。对于 `T` 实际上有 3 种选择：`Object`、`Employee` 或 `Manager`。（如果 `Manager` 或 `Employee` 实现了接口，可能还有更多选择）。不过，编译器无法知道其中哪个选择正确。所以，编译器不能接受参数类型为 `Employee` 或 `Object` 的调用。毕竟，`T` 可能是 `Manager`。只能传递 `Manager` 类型或某个子类型（如 `Executive`）的对象。

另外，如果调用 `getFirst`，不能保证返回对象的类型。只能把它赋给一个 `Object`。

下面是一个典型的示例。我们有一个经理数组，并且想把奖金最高和最低的经理放在一个 `Pair` 对象中。`Pair` 的类型是什么？在这里，`Pair<Employee>` 是合理的，或者对此而言，

`Pair<Object>`也是合理的（如图 8-4 所示）。下面的方法将接受任何合适的 `Pair`:

```
public static void minmaxBonus(Manager[] a, Pair<? super Manager> result)
{
    if (a.length == 0) return;
    Manager min = a[0];
    Manager max = a[0];
    for (int i = 1; i < a.length; i++)
    {
        if (min.getBonus() > a[i].getBonus()) min = a[i];
        if (max.getBonus() < a[i].getBonus()) max = a[i];
    }
    result.setFirst(min);
    result.setSecond(max);
}
```

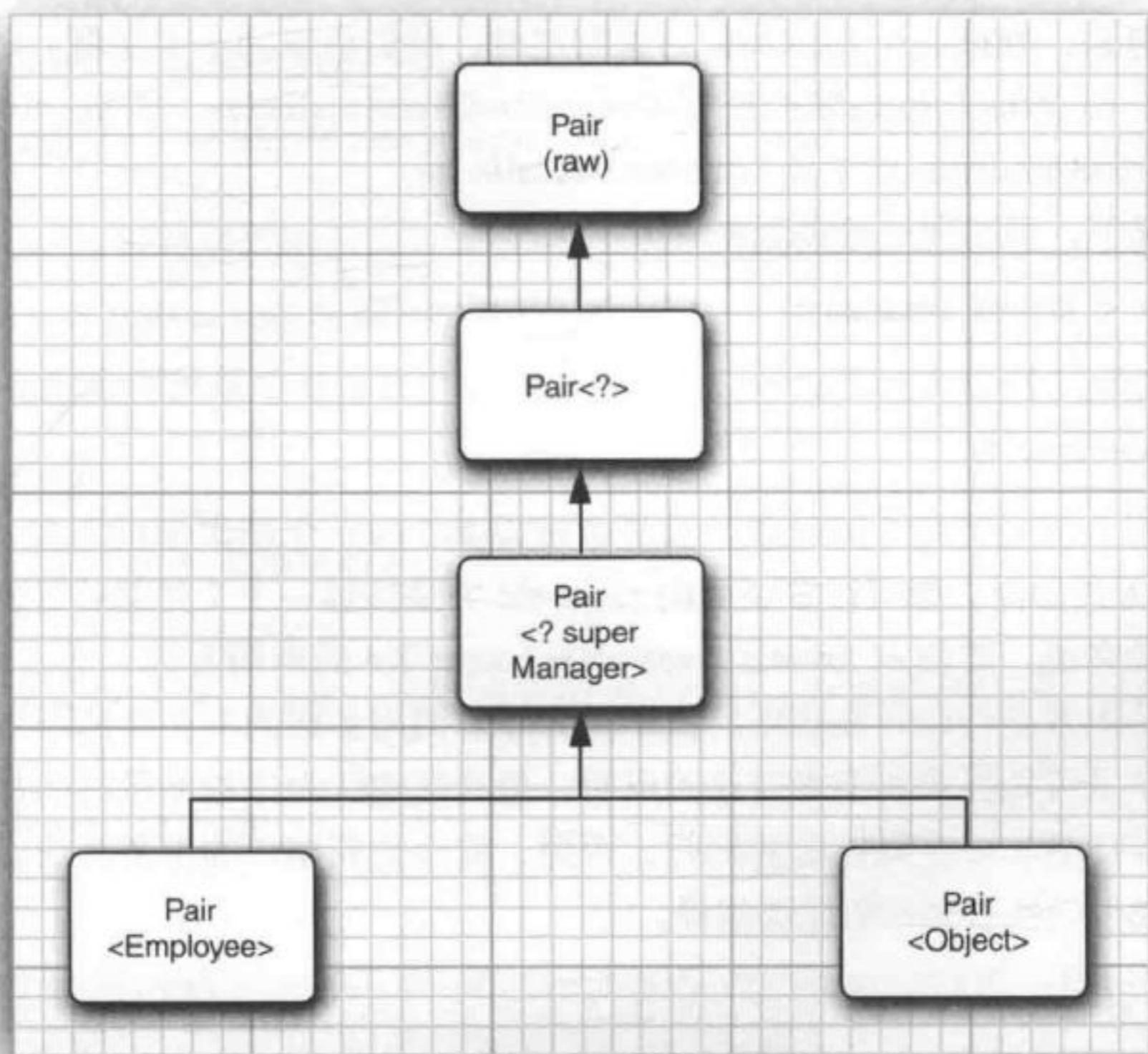


图 8-4 带有超类型限定的通配符

直观地讲，带有超类型限定的通配符允许你写入一个泛型对象，而带有子类型限定的通配符允许你读取一个泛型对象。

下面是超类型限定的另一种应用。`Comparable` 接口本身就是一个泛型类型。声明如下：

```
public interface Comparable<T>
{
    public int compareTo(T other);
}
```

在这里，类型变量指示了 `other` 参数的类型。例如，`String` 类实现了 `Comparable<String>`，它的 `compareTo` 方法声明为

```
public int compareTo(String other)
```

这很好，显式参数有正确的类型。接口是泛型接口之前，`other` 是一个 `Object`，这个方法的实现中必须有一个强制类型转换。

由于 `Comparable` 是一个泛型类型，对于 `ArrayAlg` 类的 `minmax` 方法，也许我们还能做得更好一些？可以将它声明为：

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

看起来，这样比只使用 `T extends Comparable` 更彻底，而且对于很多类都能很好地工作。例如，如果计算一个 `String` 数组的最小值，`T` 就是类型 `String`，而 `String` 是 `Comparable<String>` 的一个子类型。但是，处理一个 `LocalDate` 对象数组时，我们会遇到一个问题。`LocalDate` 实现了 `ChronoLocalDate`，而 `ChronoLocalDate` 扩展了 `Comparable<ChronoLocalDate>`。因此，`LocalDate` 实现的是 `Comparable<ChronoLocalDate>` 而不是 `Comparable<LocalDate>`。

在这种情况下，可以利用超类型来解决：

```
public static <T extends Comparable<? super T>> Pair<T> minmax(T[] a)
```

现在 `compareTo` 方法形式如下：

```
int compareTo(? super T)
```

它可以声明为接受类型 `T` 的对象，或者也可以是 `T` 的一个超类型的对象（例如，当 `T` 是 `LocalDate` 时）。无论如何，都可以安全地向 `compareTo` 方法传递一个 `T` 类型的对象。

对于初学者来说，类似 `<T extends Comparable<? super T>>` 的声明看起来有点吓人。很遗憾，因为这个声明的本意是帮助开发应用的程序员去除对调用参数的不必要的限制。对泛型没有兴趣的应用程序员可能很快就会略过这些声明，想当然地认为库程序员做的都是正确的。如果你是一名库程序员，一定要熟悉通配符，否则，就会受到用户的责备，他们要在代码中随机地添加强制类型转换直至代码能够编译。

**注释：**超类型限定的另一个常见的用法是作为一个函数式接口的参数类型。例如，`Collection` 接口有一个方法：

```
default boolean removeIf(Predicate<? super E> filter)
```

这个方法会删除所有满足给定谓词条件的元素。例如，如果你不喜欢有奇怪散列码的员工，就可以如下将他们删除：

```
ArrayList<Employee> staff = . . .;
Predicate<Object> oddHashCode = obj -> obj.hashCode() % 2 != 0;
staff.removeIf(oddHashCode);
```

你希望能够传入一个 `Predicate<Object>`，而不只是 `Predicate<Employee>`。`super` 通配符可以使这个愿望成真。

### 8.8.3 无限定通配符

甚至还可以使用根本无限定的通配符，例如，`Pair<?>`。初看起来，这好像与原始的 `Pair` 类型一样。实际上，这两种类型有很大的不同。类型 `Pair<?>` 有以下方法：

```
? getFirst()
void setFirst(?)
```

`getFirst` 的返回值只能赋给一个 `Object`。`setFirst` 方法不能调用，甚至不能用 `Object` 调用。`Pair<?>` 和 `Pair` 本质的不同在于：你可以用任意 `Object` 对象调用原始 `Pair` 类的 `setFirst` 方法。

 **注释：**可以调用 `setFirst(null)`。

为什么要使用这样一个脆弱的类型？它对于很多简单操作很有用。例如，下面这个方法可用来测试一个对组是否包含一个 `null` 引用，它不需要具体的类型。

```
public static boolean hasNulls(Pair<?> p)
{
    return p.getFirst() == null || p.getSecond() == null;
}
```

通过将 `hasNulls` 转换成泛型方法，可以避免使用通配符类型：

```
public static <T> boolean hasNulls(Pair<T> p)
```

但是，带有通配符的版本可读性更好。

### 8.8.4 通配符捕获

下面编写一个方法来交换对组的元素：

```
public static void swap(Pair<?> p)
```

通配符不是类型变量，因此，不能编写使用 `?` 作为一种类型的代码。也就是说，下面的代码是非法的：

```
? t = p.getFirst(); // ERROR
p.setFirst(p.getSecond());
p.setSecond(t);
```

这里有一个问题，因为在交换的时候，必须临时保存第一个元素。幸运的是，这个问题有一个有趣的解决方案。我们可以写一个辅助方法 `swapHelper`，如下所示：

```
public static <T> void swapHelper(Pair<T> p)
{
    T t = p.getFirst();
    p.setFirst(p.getSecond());
    p.setSecond(t);
}
```

注意，`swapHelper` 是一个泛型方法，而 `swap` 不是，它有一个固定的 `Pair<?>` 类型的参数。现在可以由 `swap` 调用 `swapHelper`：

```
public static void swap(Pair<?> p) { swapHelper(p); }
```

在这种情况下，`swapHelper` 方法的参数 `T` 捕获通配符。并不知道通配符指示哪种类型，但

是，这是一个明确的类型，并且从 `<T>swapHelper` 的定义可以清楚地看到 `T` 指示那个类型。

当然，在这种情况下，并不是一定要使用通配符。我们也可以直接把 `<T> void swap(Pair<T> p)` 实现为一个没有通配符的泛型方法。不过，考虑下面这个例子，这里通配符类型很自然地出现在一个计算中间：

```
public static void maxminBonus(Manager[] a, Pair<? super Manager> result)
{
    minmaxBonus(a, result);
    PairAlg.swapHelper(result); // OK--swapHelper captures wildcard type
}
```

在这里，通配符捕获机制是不可避免的。

通配符捕获只有在非常有限的情况下是合法的。编译器必须能够保证通配符表示单个确定的类型。例如，`ArrayList<Pair<T>>` 中的 `T` 绝对不能捕获 `ArrayList<Pair<?>>` 中的通配符。数组列表可能包含两个 `Pair<?>`，其中的 `?` 可能分别有不同的类型。

程序清单 8-3 中的测试程序将前几节讨论的各种方法综合在一起，以便我们了解它们的具体使用。

### 程序清单 8-3 pair3/PairTest3.java

```
1 package pair3;
2
3 /**
4 * @version 1.01 2012-01-26
5 * @author Cay Horstmann
6 */
7 public class PairTest3
8 {
9     public static void main(String[] args)
10    {
11        var ceo = new Manager("Gus Greedy", 800000, 2003, 12, 15);
12        var cfo = new Manager("Sid Sneaky", 600000, 2003, 12, 15);
13        var buddies = new Pair<Manager>(ceo, cfo);
14        printBuddies(buddies);
15
16        ceo.setBonus(1000000);
17        cfo.setBonus(500000);
18        Manager[] managers = { ceo, cfo };
19
20        var result = new Pair<Employee>();
21        minmaxBonus(managers, result);
22        System.out.println("first: " + result.getFirst().getName()
23                           + ", second: " + result.getSecond().getName());
24        maxminBonus(managers, result);
25        System.out.println("first: " + result.getFirst().getName()
26                           + ", second: " + result.getSecond().getName());
27    }
28
29    public static void printBuddies(Pair<? extends Employee> p)
30    {
31        Employee first = p.getFirst();
32        Employee second = p.getSecond();
```

```

33     System.out.println(first.getName() + " and " + second.getName() + " are buddies.");
34 }
35
36 public static void minmaxBonus(Manager[] a, Pair<? super Manager> result)
37 {
38     if (a.length == 0) return;
39     Manager min = a[0];
40     Manager max = a[0];
41     for (int i = 1; i < a.length; i++)
42     {
43         if (min.getBonus() > a[i].getBonus()) min = a[i];
44         if (max.getBonus() < a[i].getBonus()) max = a[i];
45     }
46     result.setFirst(min);
47     result.setSecond(max);
48 }
49
50 public static void maxminBonus(Manager[] a, Pair<? super Manager> result)
51 {
52     minmaxBonus(a, result);
53     PairAlg.swapHelper(result); // OK--swapHelper captures wildcard type
54 }
55 // can't write public static <T super manager> . . .
56 }
57
58 class PairAlg
59 {
60     public static boolean hasNulls(Pair<?> p)
61     {
62         return p.getFirst() == null || p.getSecond() == null;
63     }
64
65     public static void swap(Pair<?> p) { swapHelper(p); }
66
67     public static <T> void swapHelper(Pair<T> p)
68     {
69         T t = p.getFirst();
70         p.setFirst(p.getSecond());
71         p.setSecond(t);
72     }
73 }

```

## 8.9 反射和泛型

反射允许你在运行时分析任意对象。如果对象是泛型类的实例，关于泛型类型参数，你可能得不到多少信息，因为它们已经被擦除了。在下面的小节中，我们将学习利用反射可以获得泛型类的哪些信息。

### 8.9.1 泛型 Class 类

现在，Class 类是泛型类。例如，String.class 实际上是一个 Class<String> 类的对象（事实

上，也是唯一的对象）。

类型参数十分有用，这是因为它允许 `Class<T>` 的方法有更特定的返回类型。`Class<T>` 的以下方法就利用了类型参数：

```
T newInstance()
T cast(Object obj)
T[] getEnumConstants()
Class<? super T> getSuperclass()
Constructor<T> getConstructor(Class... parameterTypes)
Constructor<T> getDeclaredConstructor(Class... parameterTypes)
```

`newInstance` 方法返回这个类的一个实例，由无参数构造器获得。它的返回类型现在声明为 `T`，其类型与 `Class<T>` 描述的类相同，这样就免除了强制类型转换。

`cast` 方法返回给定的对象，如果给定对象的类型实际上是 `T` 的一个子类型，现在会声明为类型 `T`，否则，会抛出一个 `BadCastException` 异常。

如果这个类不是一个 `enum` 类或 `T` 类型枚举值的一个数组，`getEnumConstants` 方法将返回 `null`。

最后，`getConstructor` 与 `getDeclaredConstructor` 方法返回一个 `Constructor<T>` 对象。`Constructor` 类也已经变成泛型，使得它的 `newInstance` 方法有一个正确的返回类型。

#### API `java.lang.Class<T>` 1.0

- `T newInstance()`  
返回无参数构造器构造的一个新实例。
- `T cast(Object obj)`  
如果 `obj` 为 `null` 或者可以转换成类型 `T`，则返回 `obj`；否则抛出一个 `BadCastException` 异常。
- `T[] getEnumConstants() 5`  
如果 `T` 是枚举类型，则返回所有值组成的一个数组，否则返回 `null`。
- `Class<? super T> getSuperclass()`  
返回这个类的超类。如果 `T` 不是一个类或者如果 `T` 是 `Object` 类，则返回 `null`。
- `Constructor<T> getConstructor(Class... parameterTypes) 1.1`
- `Constructor<T> getDeclaredConstructor(Class... parameterTypes) 1.1`  
获得公共构造器，或者有给定参数类型的构造器。

#### API `java.lang.reflect.Constructor<T>` 1.1

- `T newInstance(Object... parameters)`  
返回用指定参数构造的新实例。

### 8.9.2 使用 `Class<T>` 参数进行类型匹配

匹配泛型方法中 `Class<T>` 参数的类型变量有时会很有用。下面是一个标准的示例：

```
public static <T> Pair<T> makePair(Class<T> c) throws InstantiationException,
    IllegalAccessException
{
```

```

    return new Pair<T>(c.newInstance(), c.newInstance());
}

```

如果调用

```
makePair(Employee.class)
```

`Employee.class` 是一个 `Class<Employee>` 类型的对象。`makePair` 方法的类型参数 `T` 与 `Employee` 匹配，编译器可以推断出这个方法将返回一个 `Pair<Employee>`。

### 8.9.3 虚拟机中的泛型类型信息

Java 泛型的突出特性之一是在虚拟机中擦除泛型类型。令人奇怪的是，擦除的类仍然保留原先泛型的一些微弱记忆。例如，原始 `Pair` 类知道它源于泛型类 `Pair<T>`，尽管一个 `Pair` 类型的对象无法区分它构造为 `Pair<String>` 还是 `Pair<Employee>`。

类似地，考虑以下方法：

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

这是擦除以下泛型方法得到的：

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

可以使用反射 API 确定：

- 这个泛型方法有一个名为 `T` 的类型参数。
- 这个类型参数有一个子类型限定，其自身又是一个泛型类型。
- 这个限定类型有一个通配符参数。
- 这个通配符参数有一个超类型限定。
- 这个泛型方法有一个泛型数组参数。

换句话说，你可以重新构造实现者声明的泛型类和方法的所有有关内容。但是，你不会知道对于特定的对象或方法调用会如何解析类型参数。

为了描述泛型类型声明，可以使用 `java.lang.reflect` 包中的接口 `Type`。这个接口有以下子类型：

- `Class` 类，描述具体类型。
- `TypeVariable` 接口，描述类型变量（如 `T extends Comparable<? super T>`）。
- `WildcardType` 接口，描述通配符（如 `? super T`）。
- `ParameterizedType` 接口，描述泛型类或接口类型（如 `Comparable<? super T>`）。
- `GenericArrayType` 接口，描述泛型数组（如 `T[]`）。

图 8-5 给出了继承层次结构。注意，最后 4 个子类型是接口，虚拟机会实例化实现这些接口的适当的类。

程序清单 8-4 使用泛型反射 API 来打印它发现的一个给定类的有关信息。如果对 `Pair` 类运行这个程序，将会得到以下报告：

```

class Pair<T> extends java.lang.Object
public T getFirst()

```

```
public T getSecond()
public void setFirst(T)
public void setSecond(T)
```

如果对 PairTest2 目录下的 ArrayAlg 运行这个程序，报告会显示以下方法：

```
public static <T extends java.lang.Comparable> Pair<T> minmax(T[])
```

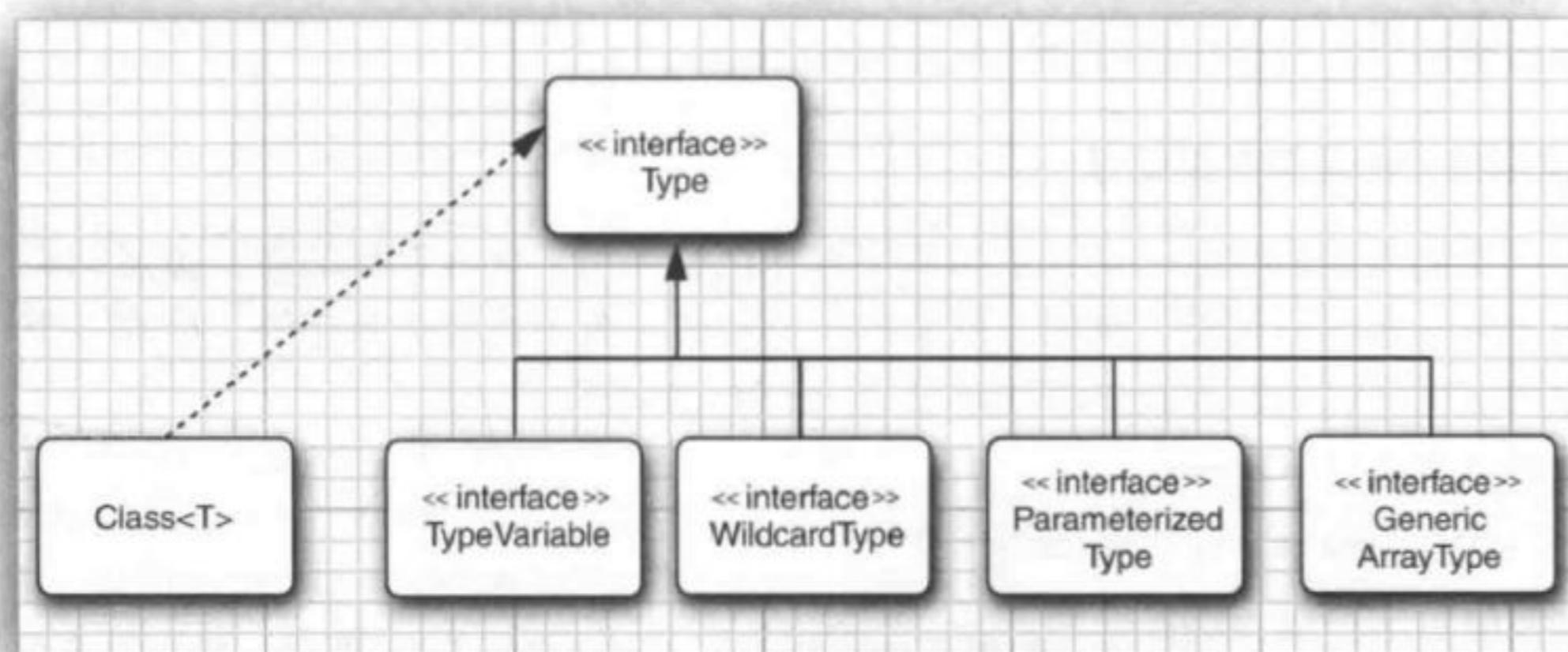


图 8-5 Type 接口及其子类型

#### 程序清单 8-4 genericReflection/GenericReflectionTest.java

```

1 package genericReflection;
2
3 import java.lang.reflect.*;
4 import java.util.*;
5
6 /**
7 * @version 1.12 2021-05-30
8 * @author Cay Horstmann
9 */
10 public class GenericReflectionTest
11 {
12     public static void main(String[] args)
13     {
14         // read class name from command line args or user input
15         String name;
16         if (args.length > 0) name = args[0];
17         else
18         {
19             try (var in = new Scanner(System.in))
20             {
21                 System.out.println("Enter class name (e.g., java.util.Collections): ");
22                 name = in.next();
23             }
24         }
25
26         try
27         {
28             // print generic info for class and public methods
29             Class<?> cl = Class.forName(name);
30         }
31     }
32 }
```

```
30     printClass(cl);
31     for (Method m : cl.getDeclaredMethods())
32         printMethod(m);
33 }
34 catch (ClassNotFoundException e)
35 {
36     e.printStackTrace();
37 }
38 }
39
40 public static void printClass(Class<?> cl)
41 {
42     System.out.print(cl);
43     printTypes(cl.getTypeParameters(), "<", ", ", ">", true);
44     Type sc = cl.getGenericSuperclass();
45     if (sc != null)
46     {
47         System.out.print(" extends ");
48         printType(sc, false);
49     }
50     printTypes(cl.getGenericInterfaces(), " implements ", ", ", "", false);
51     System.out.println();
52 }
53
54 public static void printMethod(Method m)
55 {
56     String name = m.getName();
57     System.out.print(Modifier.toString(m.getModifiers()));
58     System.out.print(" ");
59     printTypes(m.getTypeParameters(), "<", ", ", ">", true);
60
61     printType(m.getGenericReturnType(), false);
62     System.out.print(" ");
63     System.out.print(name);
64     System.out.print("(");
65     printTypes(m.getGenericParameterTypes(), "", ", ", "", false);
66     System.out.println(")");
67 }
68
69 public static void printTypes(Type[] types, String pre, String sep, String suf,
70     boolean isDefinition)
71 {
72     if (pre.equals(" extends ") && Arrays.equals(types, new Type[] { Object.class }))
73         return;
74     if (types.length > 0) System.out.print(pre);
75     for (int i = 0; i < types.length; i++)
76     {
77         if (i > 0) System.out.print(sep);
78         printType(types[i], isDefinition);
79     }
80     if (types.length > 0) System.out.print(suf);
81 }
82
83 public static void printType(Type type, boolean isDefinition)
```

```

84  {
85      if (type instanceof Class t)
86      {
87          System.out.print(t.getName());
88      }
89      else if (type instanceof TypeVariable t)
90      {
91          System.out.print(t.getName());
92          if (isDefinition)
93              printTypes(t.getBounds(), " extends ", " & ", "", false);
94      }
95      else if (type instanceof WildcardType t)
96      {
97          System.out.print("?");
98          printTypes(t.getUpperBounds(), " extends ", " & ", "", false);
99          printTypes(t.getLowerBounds(), " super ", " & ", "", false);
100     }
101     else if (type instanceof ParameterizedType t)
102     {
103         Type owner = t.getOwnerType();
104         if (owner != null)
105         {
106             printType(owner, false);
107             System.out.print(".");
108         }
109         printType(t.getRawType(), false);
110         printTypes(t.getActualTypeArguments(), "<", ", ", ", ", ">", false);
111     }
112     else if (type instanceof GenericArrayType t)
113     {
114         System.out.print("");
115         printType(t.getGenericComponentType(), isDefinition);
116         System.out.print("[]");
117     }
118 }
119 }
```

#### 8.9.4 类型字面量

有时，你会希望由值的类型决定程序的行为。例如，在一种持久存储机制中，你可能希望用户指定一种方法来保存某个特定类的对象。通常的实现方法是将 `Class` 对象与一个动作关联。

不过，如果有泛型类，擦除会带来问题。比如说，既然 `ArrayList<Integer>` 和 `ArrayList<String>` 都擦除为同一个原始类型 `ArrayList`，如何让它们有不同的动作呢？

这里有一个技巧，在某些情况下可以解决这个问题。可以捕获 `Type` 接口（上一节介绍过）的一个实例。然后构造一个匿名子类，如下所示：

```
var type = new TypeLiteral<ArrayList<Integer>>(){} // note the {}
```

`TypeLiteral` 构造器会捕获泛型超类型：

```

class TypeLiteral
{
    public TypeLiteral()
    {
        Type parentType = getClass().getGenericSuperclass();
        if (parentType instanceof ParameterizedType paramType)
            type = paramType.getActualTypeArguments()[0];
        else
            throw new UnsupportedOperationException(
                "Construct as new TypeLiteral<..>(){}");
    }
    ...
}

```

如果运行时有一个泛型类型，可以将它与 TypeLiteral 匹配。我们无法从一个对象得到泛型类型（已经被擦除）。不过，正如上一节看到的，字段和方法参数的泛型类型还留存在虚拟机中。

CDI 和 Guice 等注入框架（Injection framework）就使用类型字面量来控制泛型类型的注入。程序清单 8-5 给出了一个更简单的例子。给定一个对象，我们可以罗列它的字段，哪些有泛型类型，并查找相关联的格式化动作。

#### 程序清单 8-5 genericReflection/TypeLiterals.java

```

1 package genericReflection;
2
3 /**
4  * @version 1.02 2021-05-30
5  * @author Cay Horstmann
6 */
7
8 import java.lang.reflect.*;
9 import java.util.*;
10 import java.util.function.*;
11
12 /**
13  * A type literal describes a type that can be generic, such as
14  * ArrayList<String>.
15 */
16 class TypeLiteral<T>
17 {
18     private Type type;
19
20     /**
21      * This constructor must be invoked from an anonymous subclass
22      * as new TypeLiteral<..>(){}.
23      */
24     public TypeLiteral()
25     {
26         Type parentType = getClass().getGenericSuperclass();
27         if (parentType instanceof ParameterizedType paramType)
28             type = paramType.getActualTypeArguments()[0];

```

```
29     else
30         throw new UnsupportedOperationException(
31             "Construct as new TypeLiteral<. .>(){}");
32     }
33
34     private TypeLiteral(Type type)
35     {
36         this.type = type;
37     }
38
39     /**
40      * Yields a type literal that describes the given type.
41      */
42     public static TypeLiteral<?> of(Type type)
43     {
44         return new TypeLiteral<Object>(type);
45     }
46
47     public String toString()
48     {
49         if (type instanceof Class clazz) return clazz.getName();
50         else return type.toString();
51     }
52
53     public boolean equals(Object otherObject)
54     {
55         return otherObject instanceof TypeLiteral otherLiteral
56             && type.equals(otherLiteral.type);
57     }
58
59     public int hashCode()
60     {
61         return type.hashCode();
62     }
63 }
64
65 /**
66  * Formats objects, using rules that associate types with formatting functions.
67  */
68 class Formatter
69 {
70     private Map<TypeLiteral<?>, Function<?, String>> rules = new HashMap<>();
71
72     /**
73      * Add a formatting rule to this formatter.
74      * @param type the type to which this rule applies
75      * @param formatterForType the function that formats objects of this type
76      */
77     public <T> void forType(TypeLiteral<T> type, Function<T, String> formatterForType)
78     {
79         rules.put(type, formatterForType);
80     }
81
82     /**
```

```
83 * Formats all fields of an object using the rules of this formatter.
84 * @param obj an object
85 * @return a string with all field names and formatted values
86 */
87 public String formatFields(Object obj)
88     throws IllegalArgumentException, IllegalAccessException
89 {
90     var result = new StringBuilder();
91     for (Field f : obj.getClass().getDeclaredFields())
92     {
93         result.append(f.getName());
94         result.append("=");
95         f.setAccessible(true);
96         Function<?, String> formatterForType = rules.get(TypeLiteral.of(f.getGenericType()));
97         if (formatterForType != null)
98         {
99             // formatterForType has parameter type ?. Nothing can be passed to its apply
100            // method. Cast makes the parameter type to Object so we can invoke it.
101            @SuppressWarnings("unchecked")
102            Function<Object, String> objectFormatter
103                = (Function<Object, String>) formatterForType;
104            result.append(objectFormatter.apply(f.get(obj)));
105        }
106        else
107            result.append(f.get(obj).toString());
108            result.append("\n");
109    }
110    return result.toString();
111 }
112 }
113
114 public class TypeLiterals
115 {
116     public static class Sample
117     {
118         ArrayList<Integer> nums;
119         ArrayList<Character> chars;
120         ArrayList<String> strings;
121         public Sample()
122         {
123             nums = new ArrayList<>();
124             nums.add(42); nums.add(1729);
125             chars = new ArrayList<>();
126             chars.add('H'); chars.add('i');
127             strings = new ArrayList<>();
128             strings.add("Hello"); strings.add("World");
129         }
130     }
131
132     private static <T> String join(String separator, ArrayList<T> elements)
133     {
134         var result = new StringBuilder();
135         for (T e : elements)
136         {
```

```

137         if (result.length() > 0) result.append(separator);
138         result.append(e.toString());
139     }
140     return result.toString();
141 }
142
143 public static void main(String[] args) throws Exception
144 {
145     var formatter = new Formatter();
146     formatter.forType(new TypeLiteral<ArrayList<Integer>>() {},
147         lst -> join(" ", lst));
148     formatter.forType(new TypeLiteral<ArrayList<Character>>() {},
149         lst -> "\"" + join("", lst) + "\"");
150     System.out.println(formatter.formatFields(new Sample()));
151 }
152 }
```

我们将对一个 `ArrayList<Integer>` 进行格式化，各个值之间用空格分隔；另外还会格式化一个 `ArrayList<Character>`，将字符连接成一个字符串。所有其他数组列表都由 `ArrayList.toString` 格式化。

#### **API** `java.lang.Class<T>` 1.0

- `TypeVariable[] getTypeParameters()` 5

如果这个类型声明为泛型类型，则获得泛型类型变量，否则获得一个长度为 0 的数组。

- `Type getGenericSuperclass()` 5

获得这个类型所声明超类的泛型类型；如果这个类型是 `Object` 或者不是类类型（class type），则返回 `null`。

- `Type[] getGenericInterfaces()` 5

获得这个类型所声明接口的泛型类型（按照声明的次序），否则，如果这个类型没有实现接口，则返回长度为 0 的数组。

#### **API** `java.lang.reflect.Method` 1.1

- `TypeVariable[] getTypeParameters()` 5

如果这个方法声明为一个泛型方法，则获得泛型类型变量，否则返回长度为 0 的数组。

- `Type getGenericReturnType()` 5

获得这个方法声明的泛型返回类型。

- `Type[] getGenericParameterTypes()` 5

获得这个方法声明的泛型参数类型。如果这个方法没有参数，返回长度为 0 的数组。

#### **API** `java.lang.reflect.TypeVariable` 5

- `String getName()`

获得这个类型变量的名字。

- `Type[] getBounds()`

获得这个类型变量的子类限定，否则，如果该变量无限定，则返回长度为 0 的数组。

#### API `java.lang.reflect.WildcardType` 5

- `Type[] getUpperBounds()`

获得这个类型变量的子类（`extends`）限定，否则，如果这个变量没有子类限定，则返回长度为 0 的数组。

- `Type[] getLowerBounds()`

获得这个类型变量的超类（`super`）限定，否则，如果这个变量没有超类限定，则返回长度为 0 的数组。

#### API `java.lang.reflect.ParameterizedType` 5

- `Type getRawType()`

获得这个参数化类型的原始类型。

- `Type[] getActualTypeArguments()`

获得这个参数化类型声明的类型参数。

- `Type getOwnerType()`

如果是内部类型，则返回其外部类类型；如果这是一个顶级类型，则返回 `null`。

#### API `java.lang.reflect.GenericArrayType` 5

- `Type getGenericComponentType()`

获得这个数组类型声明的泛型元素类型。

现在我们已经了解了如何使用泛型类，以及在必要时如何编写自己的泛型类和泛型方法。同样重要的是，你知道了如何理解 API 文档和错误消息中可能遇到的泛型类型声明。要想全面地了解有关 Java 泛型的详尽信息，可以查看 Angelika Langer 提供的一个很不错的常见问题（也有一些不太常见）列表（<http://angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html>）。

在下一章中，我们将学习 Java 集合框架如何使用泛型。