第6章　接口与内部类

- ▲ 接口
- ▲ 对象克隆
- ▲ 接口与回调
- ▲ 内部类
- ▲ 代理

到目前为止，读者已经学习了Java面向对象程序设计的全部基本知识。本章将开始介绍几种常用的高级技术。这些内容可能不太容易理解，但一定要掌握它们，以便完善自己的Java工具箱。

首先，介绍一下接口（interface）技术，这种技术主要用来描述类具有什么功能，而并不给出每个功能的具体实现。一个类可以实现（implement）一个或多个接口，并在需要接口的地方，随时使用实现了相应接口的对象。了解接口以后，再继续看一下克隆对象（有时又称为深拷贝）。对象的克隆是指创建一个新对象，且新对象的状态与原始对象的状态相同。当对克隆的新对象进行修改时，不会影响原始对象的状态。

接下来，看一下内部类（inner class）机制。内部类定义在另外一个类的内部，其中的方法可以访问包含它们的外部类的域，这是一项比较复杂的技术。内部类技术主要用于设计具有相互协作关系的类集合。特别是在编写处理GUI事件的代码时，使用它将可以让代码看起来更加简练专业。

在本章的最后还将介绍代理（proxy），这是一种实现任意接口的对象。代理是一种非常专业的构造工具，它可以用来构建系统级的工具。如果是第一次学习这本书，可以先跳过这个部分。

6.1　接口

在Java程序设计语言中，接口不是类，而是对类的一组需求描述，这些类要遵从接口描述的统一格式进行定义。

我们经常听到服务提供商这样说：“ 如果类遵从某个特定接口，那么就履行这项服务 ”。下面给出一个具体的示例。Arrays类中的sort方法承诺可以对对象数组进行排序，但要求满足下列前提：对象所属的类必须实现了Comparable接口。

下面是Comparable接口的代码：

```
public interface Comparable
{
   int compareTo(Object other);
}
```

这就是说，任何实现Comparable接口的类都需要包含compareTo方法，并且这个方法的参数必须是一个Object对象，返回一个整型数值。

注释：在Java SE 5.0中，Comparable接口已经改进为泛型类型。

```
public interface Comparable<T>
{
   int compareTo(T other); // parameter has type T
}
```

例如，在实现Comparable<Employee>接口的类中，必须提供下列方法

```
int compareTo(Employee other)
```

也可以使用没有类型参数的“原始”Comparable类型，但必须手工地将compareTo方法的参数转换成所希望的类型。

接口中的所有方法自动地属于public。因此，在接口中声明方法时，不必提供关键字public。

当然，接口中还有一个没有明确说明的附加要求：在调用x.compareTo(y)的时候，这个compareTo方法必须确实比较两个对象的内容，并返回比较的结果。当x小于y时，返回一个负数；当x等于y时，返回0；否则返回一个正数。

上面这个接口只有一个方法，而有些接口可能包含多个方法。稍后可以看到，在接口中还可以定义常量。然而，更为重要的是要知道接口不能提供哪些功能。接口绝不能含有实例域，也不能在接口中实现方法。提供实例域和方法实现的任务应该由实现接口的那个类来完成。因此，可以将接口看成是没有实例域的抽象类。但是这两个概念还是有一定区别的，稍后将给出详细的解释。

现在，假设希望使用Arrays类的sort方法对Employee对象数组进行排序，Employee类就必须实现Comparable接口。

为了让类实现一个接口，通常需要下面两个步骤：

1）将类声明为实现给定的接口。

2）对接口中的所有方法进行定义。

要将类声明为实现某个接口，需要使用关键字implements：

```
class Employee implements Comparable
```

当然，这里的Employee类需要提供compareTo方法。假设希望根据雇员的薪水进行比较。如果第一个雇员的薪水低于第二个雇员的薪水就返回－1；如果相等就返回0；否则返回1。

```
public int compareTo(Object otherObject)
{
   Employee other = (Employee) otherObject;
   if (salary < other.salary) return -1;
   if (salary > other.salary) return 1;
   return 0;
}
```

警告：在接口声明中，没有将compareTo方法声明为public，这是因为在接口中的所有方法都自动地是public。不过，在实现接口时，必须把方法声明为public；否则，编译器将认为这个方法的访问属性是包可见性，即类的默认访问属性，之后编译器就会给出试图提供更弱的访问权限的警告信息。

在Java SE 5.0中，可以做得更好一些。可以将上面的实现替换为对Comparable<Employee>接口的实现。

```
class Employee implements Comparable<Employee>
{
   public int compareTo(Employee other)
   {
      if (salary < other.salary) return -1;
      if (salary > other.salary) return 1;
      return 0;
   }
    . . .
}
```

请注意，将参数Object进行类型转换总是让人感觉不太顺眼，但现在已经不见了。

提示：Comparable接口中的compareTo方法将返回一个整型数值。如果两个对象不相等，则返回一个正值或者一个负值。在对两个整数域进行比较时，这点非常有用。例如，假设每个雇员都有一个惟一整数id，并希望根据ID对雇员进行重新排序，那么就可以返回id-other.id。如果第一个ID小于另一个ID，则返回一个负值；如果两个ID相等，则返回0；否则，返回一个正值。但有一点需要注意：整数的范围不能过大，以避免造成减法运算的溢出。如果能够确信ID为非负整数，或者它们的绝对值不会超过(Integer.MAX_VALUE-1)/2，就不会出现问题。

当然，这里的相减技巧不适用于浮点值。因为在salary和other.salary很接近但又不相等的时候，它们的差经过四舍五入后有可能变成0。

现在，我们已经看到，要让一个类使用排序服务必须让它实现compareTo方法。这是理所当然的，因为要向sort方法提供对象的比较方式。但是为什么不能在Employee类直接提供一个compareTo方法，而必须实现Comparable接口呢？

主要原因在于Java程序设计语言是一种强类型（strongly typed）语言。在调用方法的时候，编译器将会检查这个方法是否存在。在sort方法中可能存在下面这样的语句：

```
if (a[i].compareTo(a[j]) > 0)
{
   // rearrange a[i] and a[j]
   . . .
}
```

为此，编译器必须确认a[i]一定有compareTo方法。如果a是一个Comparable对象的数组，就可以确保拥有compareTo方法，因为每个实现Comparable接口的类都必须提供这个方法的定义。

注释：有人认为，将Arrays类中的sort方法定义为接收一个Comparable[ ]数组就可以在使用元素类型没有实现Comparable接口的数组作为参数调用sort方法时，由编译器给出错误报告。但事实并非如此。在这种情况下，sort方法可以接收一个Object[ ]数组，并对其进行笨拙的类型转换：

```
// from the standard library--not recommended
if (((Comparable) a[i]).compareTo(a[j]) > 0)
```

```
{
   // rearrange a[i] and a[j]
   . . .
}
```

如果a[i]不属于实现了Comparable接口的类，那么虚拟机就会抛出一个异常。

例6-1给出了实现雇员数组排序的全部代码。

例6-1 EmployeeSortTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates the use of the Comparable interface.
5.  * @version 1.30 2004-02-27
6.  * @author Cay Horstmann
7.  */
8. public class EmployeeSortTest
9. {
10.    public static void main(String[] args)
11.    {
12.       Employee[] staff = new Employee[3];
13.
14.       staff[0] = new Employee("Harry Hacker", 35000);
15.       staff[1] = new Employee("Carl Cracker", 75000);
16.       staff[2] = new Employee("Tony Tester", 38000);
17.
18.       Arrays.sort(staff);
19.
20.       // print out information about all Employee objects
21.       for (Employee e : staff)
22.          System.out.println("name=" + e.getName() + ",salary=" + e.getSalary());
23.    }
24. }
25.
26. class Employee implements Comparable<Employee>
27. {
28.    public Employee(String n, double s)
29.    {
30.       name = n;
31.       salary = s;
32.    }
33.
34.    public String getName()
35.    {
36.       return name;
37.    }
38.
39.    public double getSalary()
40.    {
41.       return salary;
42.    }
43.
44.    public void raiseSalary(double byPercent)
45.    {
46.       double raise = salary * byPercent / 100;
```

```
47.       salary += raise;
48.    }
49.
50.    /**
51.     * Compares employees by salary
52.     * @param other another Employee object
53.     * @return a negative value if this employee has a lower salary than
54.     * otherObject, 0 if the salaries are the same, a positive value otherwise
55.     */
56.    public int compareTo(Employee other)
57.    {
58.       if (salary < other.salary) return -1;
59.       if (salary > other.salary) return 1;
60.       return 0;
61.    }
62.
63.    private String name;
64.    private double salary;
65. }
```

API java.lang.Comparable<T> 1.0

- int compareTo(T other)
  用这个对象与other进行比较。如果这个对象小于other则返回负值；如果相等则返回0；否则返回正值。

API java.util.Arrays 1.2

- static void sort(Object[] a)
  使用mergesort算法对数组a中的元素进行排序。要求数组中的元素必须属于实现了Comparable接口的类，并且元素之间必须是可比较的。

注释：语言标准规定：对于任意的x和y，实现必须能够保证sgn(x.compareTo(y)) = -sgn(y.compareTo(x))。（也就是说，如果y.compareTo(x) 抛出一个异常，那么x.compareTo (y) 也应该抛出一个异常。）这里的“ sgn ”是一个数值的符号：如果n是负值，sgn(n)等于－1；如果n是0，sgn(n)等于0；如果n是正值，sgn(n)等于1。简单地讲，如果调换compareTo的参数，结果的符号也应该调换（而不是实际值）。

与equals方法一样，在继承过程中有可能会出现问题。

这是因为Manager扩展了Employee，而Employee实现的是Comparable<Employee>，而不是Comparable<Manager>。如果Manager覆盖了compareTo，就必须要有经理与雇员进行比较的思想准备，绝不能仅仅将雇员转换成经理。

```
class Manager extends Employee
{
   public int compareTo(Employee other)
   {
      Manager otherManager = (Manager) other; // NO
      ...
   }
```

```
   ...
}
```

这不符合“反对称”的规则。如果x是一个Employee对象，y是一个Manager对象，调用x.compareTo(y)不会抛出异常，它只是将x和y都作为雇员进行比较。但是反过来，y.compareTo(x)将会抛出一个ClassCastException。

这种情况与第5章中讨论的equals方法一样，修改的方式也一样。有两种不同的情况。

如果子类之间的比较含义不一样，那就属于不同类对象的非法比较。每个compareTo方法都应该在开始时进行下列检测：

```
if (getClass() != other.getClass()) throw new ClassCastException();
```

如果存在这样一种通用算法，它能够对两个不同的子类对象进行比较，则应该在超类中提供一个compareTo方法，并将这个方法声明为final。

例如，假设不管薪水的多少都想让经理大于雇员，像Executive和Secretary这样的子类又该怎么办呢？如果一定要按照职务排列的话，那就应该在Employee类中提供一个rank方法。每个子类覆盖rank，并实现一个比较rank值的compareTo方法。

6.1.1 接口的特性

接口不是类，尤其不能使用new运算符实例化一个接口：

```
x = new Comparable(. . .); // ERROR
```

然而，尽管不能构造接口的对象，却能声明接口的变量：

```
Comparable x; // OK
```

接口变量必须引用实现了接口的类对象：

```
x = new Employee(. . .); // OK provided Employee implements Comparable
```

接下来，如同使用instanceof检查一个对象是否属于某个特定类一样，也可以使用instance检查一个对象是否实现了某个特定的接口：

```
if (anObject instanceof Comparable) { . . . }
```

与可以建立类的继承关系一样，接口也可以被扩展。这里允许存在多条从具有较高通用性的接口到较高专用性的接口的链。例如，假设有一个称为Moveable的接口：

```
public interface Moveable
{
   void move(double x, double y);
}
```

然后，可以以它为基础扩展一个叫做Powered的接口：

```
public interface Powered extends Moveable
{
   double milesPerGallon();
}
```

虽然在接口中不能包含实例域或静态方法，但却可以包含常量。例如：

```
public interface Powered extends Moveable
{
   double milesPerGallon();
```

```
    double SPEED_LIMIT = 95; // a public static final constant
}
```

与接口中的方法都自动地被设置为public一样，接口中的域将被自动设为public static final。

注释：可以将接口方法标记为public，将域标记为public static final。有些程序员出于习惯或提高清晰度的考虑，愿意这样做。但Java语言规范却建议不要书写这些多余的关键字，本书也采纳了这个建议。

有些接口只定义了常量，而没有定义方法。例如，在标准库中有一个SwingConstants就是这样一个接口，其中只包含NORTH、SOUTH和HORIZONTAL等常量。任何实现SwingConstants接口的类都自动地继承了这些常量，并可以在方法中直接地引用NORTH，而不必采用SwingConstants.NORTH这样的繁琐书写形式。然而，这样应用接口似乎有点偏离了接口概念的初衷，最好不要这样使用它。

尽管每个类只能够拥有一个超类，但却可以实现多个接口。这就为定义类的行为提供了极大的灵活性。例如，Java程序设计语言有一个非常重要的内置接口，称为Cloneable（将在下一节中给予详细的讨论）。如果某个类实现了这个Cloneable接口，Object类中的clone方法就可以创建类对象的一个拷贝。如果希望自己设计的类拥有克隆和比较的能力，只要实现这两个接口就可以了。

```
class Employee implements Cloneable, Comparable
```

使用逗号将实现的各个接口分隔开。

6.1.2 接口与抽象类

如果阅读了第5章中有关抽象类的内容，那就可能会产生这样一个疑问：为什么Java程序设计语言还要不辞辛苦地引入接口概念？为什么不将Comparable直接设计成如下所示的抽象类。

```
abstract class Comparable // why not?
{
   public abstract int compareTo(Object other);
}
```

然后，Employee类再直接扩展这个抽象类，并提供compareTo方法的实现：

```
class Employee extends Comparable // why not?
{
   public int compareTo(Object other) { . . . }
}
```

非常遗憾，使用抽象类表示通用属性存在这样一个问题：每个类只能扩展于一个类。假设Employee类已经扩展于一个类，例如Person，它就不能再像下面这样扩展第二个类了：

```
class Employee extends Person, Comparable // ERROR
```

但每个类可以像下面这样实现多个接口：

```
class Employee extends Person implements Comparable // OK
```

有些程序设计语言允许一个类有多个超类，例如C++。我们将此特性称为多继承（multiple inheritance）。而Java的设计者选择了不支持多继承，其主要原因是多继承会让语言本身变得非

常复杂（如同C++），效率也会降低（如同Eiffel）。

为了避免这类问题的出现，Java语言利用接口机制来实现多继承的大部分功能。

C++注释：C++具有多继承，随之带来了一些诸如虚基类、控制规则和横向指针类型转换等复杂特性。很少有C++程序员使用多继承，甚至有些人说：就不应该使用多继承。也有些程序员建议只对“混合”风格的继承使用多继承。在“混合”风格中，一个主要的基类描述父对象，其他的基类（因此称为混合）扮演辅助的角色。这种风格类似于Java类中从一个基类派生，然后实现若干个辅助接口。然而，在C++中，“混合”类可以添加默认的行为，而Java的接口则不行。

6.2 对象克隆

当拷贝一个变量时，原始变量与拷贝变量引用同一个对象，如图6-1所示。这就是说，改变一个变量所引用的对象将会对另一个变量产生影响。

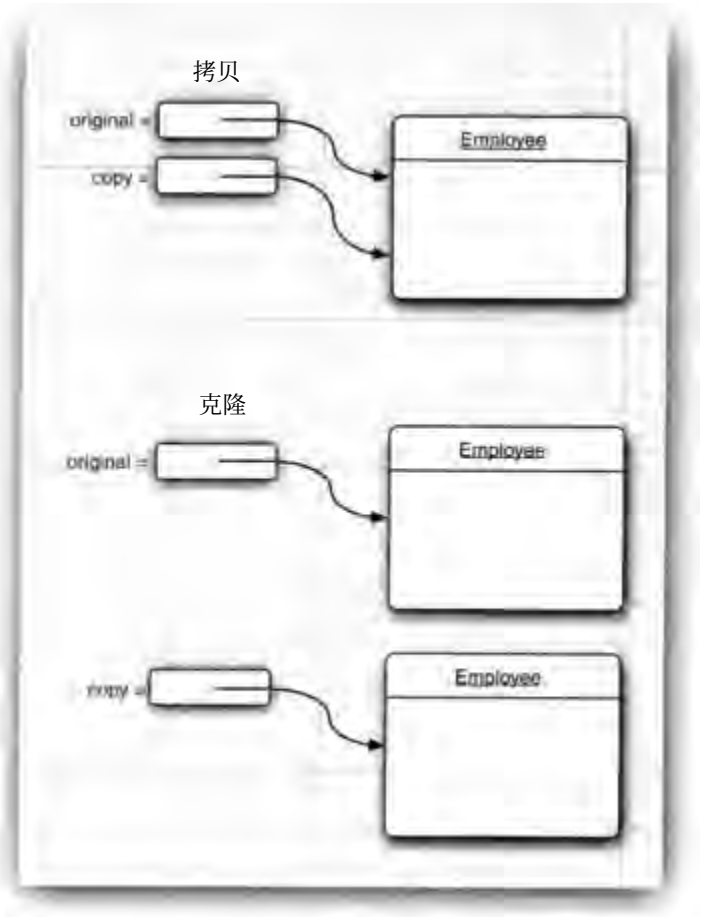

图6-1 拷贝与克隆

```
Employee original = new Employee("John Public", 50000);
Employee copy = original;
copy.raiseSalary(10); // oops--also changed original
```

如果创建一个对象的新的copy，它的最初状态与original一样，但以后将可以各自改变各自的状态，那就需要使用clone方法。

```
Employee copy = original.clone();
copy.raiseSalary(10); // OK--original unchanged
```

不过，事情并没有这么简单。clone方法是Object类的一个proteced方法，也就是说，在用户编写的代码中不能直接调用它。只有Employee类才能够克隆Employee对象。这种限制有一定的道理。这里查看一下Object类实现的clone方法。由于这个类对具体的类对象一无所知，所以只能将各个域进行对应的拷贝。如果对象中的所有数据域都属于数值或基本类型，这样拷贝域没有任何问题。但是，如果在对象中包含了子对象的引用，拷贝的结果会使得两个域引用同一个子对象，因此原始对象与克隆对象共享这部分信息。

为了能够说明这种现象，请再看一下第4章中介绍的Employee类。图6-2显示了使用Object类的clone方法克隆Employee对象的结果。可以看到，默认的克隆操作是浅拷贝，它并没有克隆包含在对象中的内部对象。

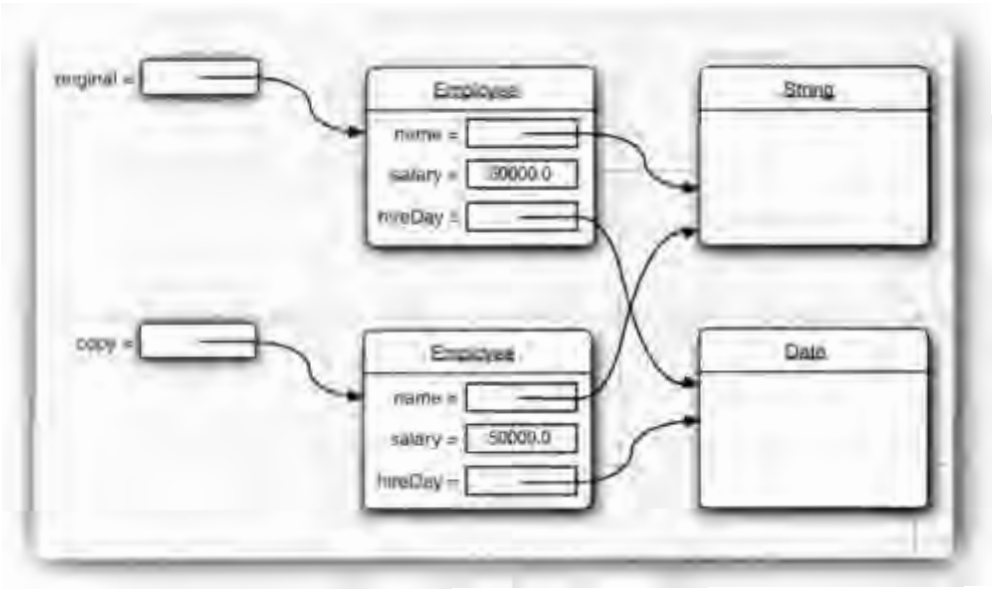

图6-2 浅拷贝

如果进行浅拷贝会发生什么呢？这要根据具体情况而定。如果原始对象与浅克隆对象共享的子对象是不可变的，将不会产生任何问题。也确实存在这种情形。例如，子对象属于像String类这样的不允许改变的类；也有可能子对象在其生命周期内不会发生变化，既没有更改它们的方法，也没有创建对它引用的方法。

然而，更常见的情况是子对象可变，因此必须重新定义clone方法，以便实现克隆子对象的深拷贝。在列举的示例中，hireDay域属于Date类，这就是一个可变的子对象。

对于每一个类，都需要做出下列判断：

1）默认的clone方法是否满足要求。

2）默认的clone方法是否能够通过调用可变子对象的clone得到修补。

3）是否不应该使用clone。

实际上，选项3是默认的。如果要选择1或2，类必须：

1）实现Cloneable接口。

2）使用public访问修饰符重新定义clone方法。

注释：在Object类中，clone方法被声明为protected，因此无法直接调用anObject.clone( )。但是，不是所有子类都可以访问受保护的方法吗？不是每个类都是Object的子类吗？值得庆幸的是，受保护访问的规则极为微妙（参阅第5章）。子类只能调用受保护的clone方法克隆它自己。为此，必须重新定义clone方法，并将它声明为public，这样才能够让所有的方法克隆对象。

在这里，Cloneable接口的出现与接口的正常使用没有任何关系。尤其是，它并没有指定clone方法，这个方法是从Object类继承而来的。接口在这里只是作为一个标记，表明类设计者知道要进行克隆处理。如果一个对象需要克隆，而没有实现Cloneable接口，就会产生一个已检验异常（checked exception）。

注释：Cloneable接口是Java提供的几个标记接口（tagging interface）之一（有些程序员将它们称为标记接口（marker interface））。我们知道，通常使用接口的目的是为了确保类实现某个特定的方法或一组特定的方法，Comparable接口就是这样一个示例。而标记接口没有方法，使用它的惟一目的是可以用instanceof进行类型检查：

```
if (obj instanceof Cloneable) . . .
```

建议在自己编写程序时，不要使用这种技术。

即使clone的默认实现（浅拷贝）能够满足需求，也应该实现Cloneable接口，将clone重定义为public，并调用super.clone( )。下面是一个示例：

```
class Employee implements Cloneable
{
   // raise visibility level to public, change return type
   public Employee clone() throws CloneNotSupportedException
   {
      return (Employee) super.clone();
   }
   . . .
}
```

注释：在Java SE 5.0以前的版本中，clone方法总是返回Object类型，而在Java SE 5.0中，允许克隆方法指定返回类型。

刚才看到的clone方法并没有在Object.clone提供的浅拷贝基础上增加任何新功能，而只是将这个方法声明为public。为了实现深拷贝，必须克隆所有可变的实例域。

下面是一个建立深拷贝clone方法的一个示例：

```
class Employee implements Cloneable
{
```

```
   . . .
   public Employee clone() throws CloneNotSupportedException
   {
      // call Object.clone()
      Employee cloned = (Employee) super.clone();

      // clone mutable fields
      cloned.hireDay = (Date) hireDay.clone()

      return cloned;
   }
}
```

只要在clone中含有没有实现Cloneable接口的对象，Object类的clone方法就会抛出一个CloneNot-SupportException异常。当然，Employee和Date类都实现了Cloneable接口，因此不会抛出异常。但是编译器并不知道这些情况，因此需要声明异常：

```
public Employee clone() throws CloneNotSupportedException
```

如果将上面这种形式替换成捕获异常呢？

```
public Employee clone()
{
   try
   {
      return super.clone();
   }
   catch (CloneNotSupportedException e) { return null; }
   // this won't happen, since we are Cloneable
}
```

这种写法比较适用于final类，否则最好还是在这个地方保留throws说明符。如果不支持克隆，子类具有抛出CloneNotSupportException异常的选择权。

必须谨慎地实现子类的克隆。例如，一旦为Employee类定义了clone方法，任何人都可以利用它克隆Manager对象。Employee的克隆方法能够完成这项重任吗？这将取决于Manager类中包含哪些域。在前面列举的示例中，由于bonus域属于基本类型，所以不会出现任何问题。但是，在Manager类中有可能存在一些需要深拷贝的域，或者包含一些没有实现Cloneable接口的域。没有人能够保证子类实现的clone一定正确。鉴于这个原因，应该将Object类中的clone方法声明为protected。但是，如果想让用户调用clone方法，就不能这样做。

在自定义的类中应该实现clone方法吗？如果客户需要深拷贝就应该实现它。有些人认为应该完全避免使用clone，并通过实现其他的方法达到此目的。我们同意这种观点，clone的确显得有点笨拙，但改用其他方法实现这项操作也会遇到同样的问题。至少，克隆的应用并不像人们想像的那样普遍。在标准类库中，只有不到5%的类实现了clone。

在例6-2的程序中，克隆了一个Employee对象，然后，调用了两个改变域值的方法。raiseSalary方法改变了salary域值，setHireDay方法改变了hireDay域值。由于clone实现的是深拷贝，所以对这两个域值的改变并没有影响原始对象。

注释：所有的数组类型均包含一个clone方法，这个方法被设为public，而不是protected。可以利用这个方法创建一个包含所有数据元素拷贝的一个新数组。例如：

```
int[] luckyNumbers = { 2, 3, 5, 7, 11, 13 };
int[] cloned = (int[]) luckyNumbers.clone();
cloned[5] = 12; // doesn't change luckyNumbers[5]
```

注释：将在卷 II第1章中介绍另一种克隆对象的机制，其中使用了Java的序列化功能。这种机制很容易实现并且也很安全，但效率较低。

例6-2 CloneTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates cloning.
5.  * @version 1.10 2002-07-01
6.  * @author Cay Horstmann
7.  */
8. public class CloneTest
9. {
10.    public static void main(String[] args)
11.    {
12.       try
13.       {
14.          Employee original = new Employee("John Q. Public", 50000);
15.          original.setHireDay(2000, 1, 1);
16.          Employee copy = original.clone();
17.          copy.raiseSalary(10);
18.          copy.setHireDay(2002, 12, 31);
19.          System.out.println("original=" + original);
20.          System.out.println("copy=" + copy);
21.       }
22.       catch (CloneNotSupportedException e)
23.       {
24.          e.printStackTrace();
25.       }
26.    }
27. }
28.
29. class Employee implements Cloneable
30. {
31.    public Employee(String n, double s)
32.    {
33.       name = n;
34.       salary = s;
35.       hireDay = new Date();
36.    }
37.
38.    public Employee clone() throws CloneNotSupportedException
39.    {
40.       // call Object.clone()
41.       Employee cloned = (Employee) super.clone();
42.
43.       // clone mutable fields
44.       cloned.hireDay = (Date) hireDay.clone();
45.
46.       return cloned;
47.    }
```

```
48.
49.    /**
50.     * Set the hire day to a given date.
51.     * @param year the year of the hire day
52.     * @param month the month of the hire day
53.     * @param day the day of the hire day
54.     */
55.    public void setHireDay(int year, int month, int day)
56.    {
57.       Date newHireDay = new GregorianCalendar(year, month - 1, day).getTime();
58.
59.       // Example of instance field mutation
60.       hireDay.setTime(newHireDay.getTime());
61.    }
62.
63.    public void raiseSalary(double byPercent)
64.    {
65.       double raise = salary * byPercent / 100;
66.       salary += raise;
67.    }
68.
69.    public String toString()
70.    {
71.       return "Employee[name=" + name + ",salary=" + salary + ",hireDay=" + hireDay + "]";
72.    }
73.
74.    private String name;
75.    private double salary;
76.    private Date hireDay;
77. }
```

6.3 接口与回调

回调（callback）是一种常见的程序设计模式。在这种模式中，可以指出某个特定事件发生时应该采取的动作。例如，可以指出在按下鼠标或选择某个菜单项时应该采取什么行动。然而，由于至此还没有介绍如何实现用户接口，所以只能讨论一些与上述操作类似，但比较简单的示例。

在java.swing包中有一个Timer类，可以使用它在到达给定的时间间隔时发出通告。例如，假如程序中有一个时钟，就可以请求每秒钟获得一个通告，以便更新时钟的画面。

在构造定时器时，需要设置一个时间间隔，并告之定时器，当到达时间间隔时需要做些什么操作。

如何告之定时器做什么呢？在很多程序设计语言中，可以提供一个函数名，定时器周期性地调用它。但是，在Java标准类库中的类采用的是面向对象方法。它将某个类的对象传递给定时器，然后，定时器调用这个对象的方法。由于对象可以携带一些附加的信息，所以传递一个对象比传递一个函数要灵活的多。

当然，定时器需要知道调用哪一个方法，并要求传递的对象所属的类实现了java.awt.event包的ActionListener接口。下面是这个接口：

```
public interface ActionListener
```

```
{
   void actionPerformed(ActionEvent event);
}
```

当到达指定的时间间隔时，定时器就调用actionPerformed方法。

C++注释：第5章已经讲过，Java有函数指针的对应物—Method对象。然而，使用起来却比较困难，速度也稍慢一些，并且在编译时不能提供类型的安全性检查。因此，在任何使用C++函数指针的地方，都应该考虑使用Java中的接口。

假设希望每隔10秒种打印一条信息“ At the tone, the time is . . . ”，然后响一声，就应该定义一个实现ActionListener接口的类，然后将需要执行的语句放在actionPerformed方法中。

```
class TimePrinter implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      Date now = new Date();
      System.out.println("At the tone, the time is " + now);
      Toolkit.getDefaultToolkit().beep();
   }
}
```

需要注意actionPerformed方法的ActionEvent参数。这个参数提供了事件的相关信息，例如，产生这个事件的源对象。有关这方面的详细内容请参看第8章。在这个程序中，事件的详细信息并不重要，因此，可以放心地忽略这个参数。

接下来，构造这个类的一个对象，并将它传递给Timer构造器。

```
ActionListener listener = new TimePrinter();
Timer t = new Timer(10000, listener);
```

Timer构造器的第一个参数是发出通告的时间间隔，它的单位是毫秒。这里希望每隔10秒种通告一次。第二个参数是监听器对象。

最后，启动定时器：

```
t.start();
```

每个10秒钟，下列信息显示一次，然后响一声铃。

```
At the tone, the time is Thu Apr 13 23:29:08 PDT 2000
```

例6-3给出了定时器和监听器的操作行为。在定时器启动以后，程序将弹出一个消息对话框，并等待用户点击Ok按钮来终止程序的执行。在程序等待用户操作的同时，每隔10秒显示一次当前的时间。

运行这个程序时要有一些耐心。程序启动后，将会立即显示一个包含“ Quit program? ”字样的对话框，10秒钟之后，第1条定时器消息才会显示出来。

需要注意，这个程序除了导入javax.swing.*和java.util.*外，还通过类名导入了javax.swing.Timer。这就消除了javax.swing.Timer与java.util.Timer之间产生的二义性。这里的java.util.Timer是一个与本例无关的类，它主要用于调度后台任务。

例6-3 TimerTest.java

```
1. /**
2.    @version 1.00 2000-04-13
3.    @author Cay Horstmann
4. */
5.
6. import java.awt.*;
7. import java.awt.event.*;
8. import java.util.*;
9. import javax.swing.*;
10. import javax.swing.Timer;
11. // to resolve conflict with java.util.Timer
12.
13. public class TimerTest
14. {
15.    public static void main(String[] args)
16.    {
17.       ActionListener listener = new TimePrinter();
18.
19.       // construct a timer that calls the listener
20.       // once every 10 seconds
21.       Timer t = new Timer(10000, listener);
22.       t.start();
23.
24.       JOptionPane.showMessageDialog(null, "Quit program?");
25.       System.exit(0);
26.    }
27. }
28.
29. class TimePrinter implements ActionListener
30. {
31.    public void actionPerformed(ActionEvent event)
32.    {
33.       Date now = new Date();
34.       System.out.println("At the tone, the time is " + now);
35.       Toolkit.getDefaultToolkit().beep();
36.    }
37. }
```

API javax.swing.JOptionPane 1.2

- `static void showMessageDialog(Component parent, Object message)`
  显示一个包含一条消息和OK按钮的对话框。这个对话框将位于其parent组件的中央。如果parent为null，对话框将显示在屏幕的中央。

API javax.swing.Timer 1.2

- `Timer(int interval, AvtionListener listener)`
  构造一个定时器，每隔interval毫秒钟通告listener一次。
- `void start()`
  启动定时器。一旦启动成功，定时器将调用监听器的actionPerformed。

- `void stop()`
  停止定时器。一旦停止成功，定时器将不再调用监听器的actionPerformed。

API `javax.awt.Toolkit 1.0`

- `static Toolkit getDefaultToolkit()`
  获得默认的工具箱。工具箱包含有关GUI环境的信息。
- `void beep()`
  发出一声铃响。

6.4 内部类

内部类（inner class）是定义在另一个类中的类。为什么需要使用内部类呢？其主要原因有以下三点：

- 内部类方法可以访问该类定义所在的作用域中的数据，包括私有的数据。
- 内部类可以对同一个包中的其他类隐藏起来。
- 当想要定义一个回调函数且不想编写大量代码时，使用匿名（anonymous）内部类比较便捷。

我们将这个比较复杂的内容分几部分介绍。

- 在6.4.1节中，给出一个简单的内部类，它将访问外围类的实例域。
- 在6.4.2节中，给出内部类的特殊语法规则。
- 在6.4.3节中，领略一下内部类的内部，探讨一下如何将其转换成常规类。读者可以跳过这一节。
- 在6.4.4节中，讨论局部内部类，它可以访问作用域中的局部变量。
- 在6.4.5节中，介绍匿名内部类，说明用于实现回调的基本方法。
- 最后在6.4.6节中，介绍如何将静态内部类嵌套在辅助类中。

C++ C++注释：C++有嵌套类。一个被嵌套的类包含在外围类的作用域内。下面是一个典型的例子，一个链表类定义了一个存储结点的类和一个定义迭代器位置的类。

```
class LinkedList
{
public:
   class Iterator // a nested class
   {
   public:
      void insert(int x);
      int erase();
      . . .
   };
   . . .
private:
   class Link // a nested class
   {
   public:
      Link* next;
```

```
      int data;
    };
    . . .
  };
```

嵌套是一种类之间的关系，而不是对象之间的关系。一个LinkedList对象并不包含Iterator类型或Link类型的子对象。

嵌套类有两个好处：命名控制和访问控制。由于名字Iterator嵌套在LinkedList类的内部，所以在外部被命名为LinkedList::Iterator，这样就不会与其他名为Iterator的类发生冲突。在Java中这个并不重要，因为Java包已经提供了相同的命名控制。需要注意的是，Link类位于LinkedList类的私有部分，因此，Link对其他的代码均不可见。鉴于此情况，可以将Link的数据域设计为公有的，它仍然是安全的。这些数据域只能被LinkedList类（具有访问这些数据域的合理需要）中的方法访问，而不会暴露给其他的代码。在Java中，只有内部类能够实现这样的控制。

然而，Java内部类还有另外一个功能，这使得它比C++的嵌套类更加丰富，用途更加广泛。内部类的对象有一个隐式引用，它引用了实例化该内部对象的外围类对象。通过这个指针，可以访问外围类对象的全部状态。在本章后续内容中，我们将会看到有关这个Java机制的详细介绍。

在Java中，static内部类没有这种附加指针，这样的内部类与C++中的嵌套类很相似。

6.4.1 使用内部类访问对象状态

内部类的语法比较复杂。鉴于此情况，我们选择一个简单但不太实用的例子说明内部类的使用方式。下面将进一步分析TimerTest示例，并抽象出一个TalkingClock类。构造一个语音时钟时需要提供两个参数：发布通告的间隔和开关铃声的标志。

```
public class TalkingClock
{
   public TalkingClock(int interval, boolean beep) { . . . }
   public void start() { . . . }

   private int interval;
   private boolean beep;

   public class TimePrinter implements ActionListener
      // an inner class
   {
      . . .
   }
}
```

需要注意，这里的TimePrinter类位于TalkingClock类内部。这并不意味着每个TalkingClock都有一个TimePrinter实例域。如前所示，TimePrinter对象是由TalkingClock类的方法构造。

下面是TimePrinter类的详细内容。需要注意一点，actionPerformed方法在发出铃声之前检查了beep标志。

```
private class TimePrinter implements ActionListener
```

```
{
   public void actionPerformed(ActionEvent event)
   {
      Date now = new Date();
      System.out.println("At the tone, the time is " + now);
      if (beep) Toolkit.getDefaultToolkit().beep();
   }
}
```

令人惊讶的事情发生了。TimePrinter类没有实例域或者名为beep的变量，取而代之的是beep引用了创建TimePrinter的TalkingClock对象的域。这是一种创新的想法。从传统意义上讲，一个方法可以引用调用这个方法的对象数据域。内部类既可以访问自身的数据域，也可以访问创建它的外围类对象的数据域。

为了能够运行这个程序，内部类的对象总有一个隐式引用，它指向了创建它的外部类对象。如图6-3所示。

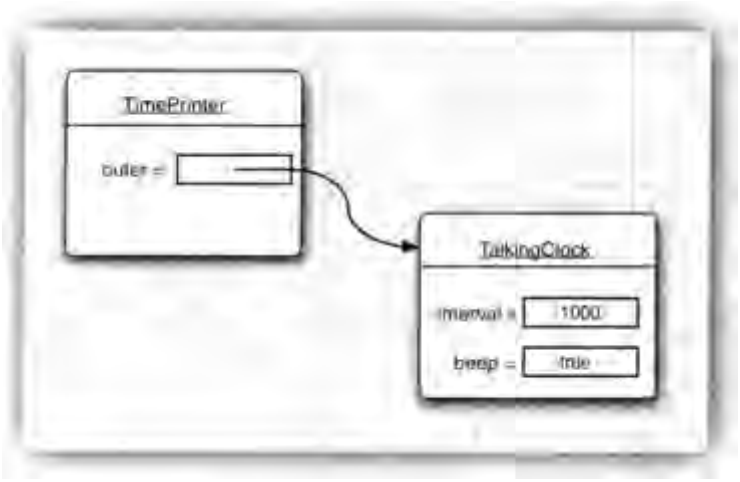

图6-3 内部类对象拥有一个对外围类对象的引用

这个引用在内部类的定义中是不可见的。然而，为了说明这个概念，我们将外围类对象的引用称为outer。于是actionPerformed方法将等价于下列形式：

```
public void actionPerformed(ActionEvent event)
{
   Date now = new Date();
   System.out.println("At the tone, the time is " + now);
   if (outer.beep) Toolkit.getDefaultToolkit().beep();
}
```

外围类的引用在构造器中设置。编译器修改了所有的内部类的构造器，添加一个外围类引用的参数。因为TimePrinter类没有定义构造器，所以编译器为这个类生成了一个默认的构造器，其代码如下所示：

```
public TimePrinter(TalkingClock clock) // automatically generated code
{
   outer = clock;
}
```

请再注意一下，outer不是Java的关键字。我们只是用它说明内部类中的机制。

当在start方法中创建了TimePrinter对象后，编译器就会将this引用传递给当前的语音时钟的构造器：

```
ActionListener listener = new TimePrinter(this); // parameter automatically added
```

例6-4给出了一个测试内部类的完整程序。下面我们再看一下访问控制。如果有一个TimePrinter类是一个常规类，它就需要通过TalkingClock类的公有方法访问beep标志，而使用内部类可以给予改进，即不必提供仅用于访问其他类的访问器。

注释：TimePrinter类被声明为私有的。这样一来，只有TalkingClock的方法才能够生成TimePrinter对象。只有内部类可以是私有类，而常规类只可以具有包可见性，或公有可见性。

例6-4 InnerClassTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.util.*;
4. import javax.swing.*;
5. import javax.swing.Timer;
6.
7. /**
8.  * This program demonstrates the use of inner classes.
9.  * @version 1.10 2004-02-27
10.  * @author Cay Horstmann
11.  */
12. public class InnerClassTest
13. {
14.    public static void main(String[] args)
15.    {
16.       TalkingClock clock = new TalkingClock(1000, true);
17.       clock.start();
18.
19.       // keep program running until user selects "Ok"
20.       JOptionPane.showMessageDialog(null, "Quit program?");
21.       System.exit(0);
22.    }
23. }
24.
25. /**
26.  * A clock that prints the time in regular intervals.
27.  */
28. class TalkingClock
29. {
30.    /**
31.     * Constructs a talking clock
32.     * @param interval the interval between messages (in milliseconds)
33.     * @param beep true if the clock should beep
34.     */
35.    public TalkingClock(int interval, boolean beep)
36.    {
37.       this.interval = interval;
38.       this.beep = beep;
```

```
39.    }
40.
41.    /**
42.     * Starts the clock.
43.     */
44.    public void start()
45.    {
46.       ActionListener listener = new TimePrinter();
47.       Timer t = new Timer(interval, listener);
48.       t.start();
49.    }
50.
51.    private int interval;
52.    private boolean beep;
53.
54.    public class TimePrinter implements ActionListener
55.    {
56.       public void actionPerformed(ActionEvent event)
57.       {
58.          Date now = new Date();
59.          System.out.println("At the tone, the time is " + now);
60.          if (beep) Toolkit.getDefaultToolkit().beep();
61.       }
62.    }
63. }
```

6.4.2 内部类的特殊语法规则

在上一节中，已经讲述了内部类有一个外围类的引用outer。事实上，使用外围类引用的正规语法还要复杂一些。表达式

OuterClass.this

表示外围类引用。例如，可以像下面这样编写TimePrinter内部类的actionPerformed方法：

```
public void actionPerformed(ActionEvent event)
{
   . . .
   if (TalkingClock.this.beep) Toolkit.getDefaultToolkit().beep();
}
```

反过来，可以采用下列语法格式更加明确地编写内部对象的构造器：

outerObject.new *InnerClass*(*construction parameters*)

例如，

```
ActionListener listener = this.new TimePrinter();
```

在这里，最新构造的TimePrinter对象的外围类引用被设置为创建内部类对象的方法中的this引用。这是一种很常见的情况。通常，this限定词是多余的。不过，可以通过显式地命名将外围类引用设置为其他的对象。例如，如果TimePrinter是一个公有内部类，对于任意的语音时钟都可以构造一个TimePrinter：

```
TalkingClock jabberer = new TalkingClock(1000, true);
TalkingClock.TimePrinter listener = jabberer.new TimePrinter();
```

需要注意，在外围类的作用域之外，可以这样引用内部类：

OuterClass.InnerClass

6.4.3 内部类是否有用、必要和安全

当在Java 1.1的Java语言中增加内部类时，很多程序员都认为这是一项很主要的新特性，但这却违背了Java要比C++更加简单的设计理念。内部类的语法很复杂（可以看到，稍后介绍的匿名内部类更加复杂）。它与访问控制和安全性等其他的语言特性的没有明显的关联。

由于增加了一些看似优美有趣，实属没必要的特性，似乎Java也开始走上了许多语言饱受折磨的毁灭性道路上。

我们并不打算就这个问题给予一个完整的答案。内部类是一种编译器现象，与虚拟机无关。编译器将会把内部类翻译成用$（美元符号）分隔外部类名与内部类名的常规类文件，而虚拟机则对此一无所知。

例如，在TalkingClock类内部的TimePrinter类将被翻译成类文件TalkingClock$TimePrinter.class。为了能够看到执行的效果，可以做一下这个实验：运行第5章中的程序ReflectionTest，并将类TalkingClock$TimePrinter传递给它进行反射。也可以选择简单的使用javap，如下所示：

```
javap -private ClassName
```

注释：如果使用UNIX，并以命令行的方式提供类名，就需要记住将$字符进行转义。也就是说，应该按照下面这种格式或javap程序运行ReflectionTest程序：

```
java ReflectionTest TalkingClock\$TimePrinter
```

或

```
javap -private TalkingClock\$TimePrinter
```

这时会看到下面的输出结果：

```
public class TalkingClock$TimePrinter
{
   public TalkingClock$TimePrinter(TalkingClock);

   public void actionPerformed(java.awt.event.ActionEvent);

   final TalkingClock this$0;
}
```

可以清楚地看到，编译器为了引用外围类，生成了一个附加的实例域this$0（名字this$0是由编译器合成的，在自己编写的代码中不能够引用它）。另外，还可以看到构造器的TalkingClock参数。

如果编译器能够自动地进行转换，那么能不能自己编写程序实现这种机制呢？让我们试试看。将TimePrinter定义成一个常规类，并把它置于TalkingClock类的外部。在构造TimePrinter对象的时候，将创建该对象的this指针传递给它。

```
class TalkingClock
{
   . . .
```

```
   public void start()
   {
      ActionListener listener = new TimePrinter(this);
      Timer t = new Timer(interval, listener);
      t.start();
   }
}

class TimePrinter implements ActionListener
{
   public TimePrinter(TalkingClock clock)
   {
       outer = clock;
   }
   . . .
   private TalkingClock outer;
}
```

现在，看一下actionPerformed方法，它需要访问outer.beep。

```
if (outer.beep) . . . // ERROR
```

这就遇到了一个问题。内部类可以访问外围类的私有数据，但这里的TimePrinter类则不行。

可见，由于内部类拥有访问特权，所以与常规类比较起来功能更加强大。

可能有人会好奇，既然内部类可以被翻译成名字很古怪的常规类（而虚拟机对此一点也不了解），内部类如何管理那些额外的访问特权呢？为了揭开这个谜团，让我们再次利用ReflectTest程序查看一下TalkingClock类：

```
class TalkingClock
{
   public TalkingClock(int, boolean);

   static boolean access$0(TalkingClock);
   public void start();

   private int interval;
   private boolean beep;
}
```

请注意编译器在外围类添加静态方法access$0。它将返回作为参数传递给它的对象域beep。

内部类方法将调用那个方法。在TimePrinter类的actionPerformed方法中编写语句：

```
if (beep)
```

将会提高下列调用的效率：

```
if (access$0(outer));
```

这样做不是存在安全风险吗？这种担心是很有道理的。任何人都可以通过调用access$0方法很容易地读取到私有域beep。当然，access$0不是Java的合法方法名。但熟悉类文件结构的黑客可以使用十六进制编辑器轻松地创建一个用虚拟机指令调用那个方法的类文件。由于隐秘地访问方法需要拥有包可见性，所以攻击代码需要与被攻击类放在同一个包中。

总而言之，如果内部类访问了私有数据域，就有可能通过附加在外围类所在包中的其他类访问它们，但做这些事情需要高超的技巧和极大的决心。程序员不可能无意之中就获得对类的

访问权限，而必须刻意地构建或修改类文件才有可能达到这个目的。

注释：合成构造器和方法是复杂令人费解的（如果过于注重细节，可以跳过这个注释）。假设将TimePrinter转换为一个内部类。在虚拟机中不存在私有类，因此编译器将会利用私有构造器生成一个包可见的类：

```
private TalkingClock$TimePrinter(TalkingClock);
```

当然，没有人可以调用这个构造器，因此，存在第二个包可见构造器

```
TalkingClock$TimePrinter(TalkingClock, TalkingClock$1);
```

它将调用第一个构造器。

编译器将TalkingClock类start方法中的构造器调用翻译为：

```
new TalkingClock$TimePrinter(this, null)
```

6.4.4 局部内部类

如果仔细地阅读一下TalkingClock示例的代码就会发现，TimePrinter这个类名字只在start方法中创建这个类型的对象时使用了一次。

当遇到这类情况时，可以在一个方法中定义局部类。

```
public void start()
{
   class TimePrinter implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
          Date now = new Date();
          System.out.println("At the tone, the time is " + now);
          if (beep) Toolkit.getDefaultToolkit().beep();
      }
   }

   ActionListener listener = new TimePrinter();
   Timer t = new Timer(interval, listener);
   t.start();
}
```

局部类不能用public或private访问说明符进行声明。它的作用域被限定在声明这个局部类的块中。

局部类有一个优势，即对外部世界可以完全地隐藏起来。即使TalkingClock类中的其他代码也不能访问它。除start方法之外，没有任何方法知道TimePrinter类的存在。

6.4.5 由外部方法访问final变量

与其他内部类相比较，局部类还有一个优点。它们不仅能够访问包含它们的外部类，还可以访问局部变量。不过，那些局部变量必须被声明为final。下面是一个典型的示例。这里，将TalkjingClock构造器的参数interval和beep移至start方法中。

```
public void start(int interval, final boolean beep)
{
   class TimePrinter implements ActionListener
```

```
        {
            public void actionPerformed(ActionEvent event)
            {
                Date now = new Date();
                System.out.println("At the tone, the time is " + now);
                if (beep) Toolkit.getDefaultToolkit().beep();
            }
        }

        ActionListener listener = new TimePrinter();
        Timer t = new Timer(interval, listener);
        t.start();
    }
```

请注意，TalkingClock类不再需要存储实例变量beep了，它只是引用start方法中的beep参数变量。

这看起来好像没什么值得大惊小怪的。程序行

```
if (beep) . . .
```

毕竟在start方法内部，为什么不能访问beep变量的值呢？

为了能够清楚地看到内部的问题，让我们仔细地考查一下控制流程。

1）调用start方法。

2）调用内部类TimePrinter的构造器，以便初始化对象变量listener。

3）将listener引用传递给Timer构造器，定时器开始计时，start方法结束。此时，start方法的beep参数变量不复存在。

4）然后，actionPerformed方法执行if (beep)...。

为了能够让actionPerformed方法工作，TimePrinter类在beep域释放之前将beep域用start方法的局部变量进行备份。实际上也是这样做的。在我们列举的例子中，编译器为局部内部类构造了名字TalkingClock$TimePrinter。如果再次运行ReflectionTest程序，查看TalkingClock$TimePrinter类，就会看到下列结果：

```
class TalkingClock$1TimePrinter
{
   TalkingClock$1TimePrinter(TalkingClock, boolean);

   public void actionPerformed(java.awt.event.ActionEvent);

   final boolean val$beep;
   final TalkingClock this$0;
}
```

请注意构造器的boolean参数和val$beep实例变量。当创建一个对象的时候，beep就会被传递给构造器，并存储在val$beep域中。编译器必须检测对局部变量的访问，为每一个变量建立相应的数据域，并将局部变量拷贝到构造器中，以便将这些数据域初始化为局部变量的副本。

从程序员的角度看，局部变量的访问非常容易。它减少了需要显式编写的实例域，从而使得内部类更加简单。

前面曾经提到，局部类的方法只可以引用定义为final的局部变量。鉴于此情况，在列举的

示例中，将beep参数声明为final，对它进行初始化后不能够再进行修改。因此，就使得局部变量与在局部类内建立的拷贝保持一致。

注释：前面曾经将final变量作为常量使用，例如：

```
public static final double SPEED_LIMIT = 55;
```

final关键字可以应用于局部变量、实例变量和静态变量。在所有这些情况下，它们的含义都是：在创建这个变量之后，只能够为之赋值一次。此后，再也不能修改它的值了，这就是final。

不过，在定义final变量的时候，不必进行初始化。例如，当调用start方法时，final参数变量beep只能够在创建之后被初始化一次（如果这个方法被调用多次，那么每次调用都有一个新创建的beep参数）。可以看到在Talking$1TimePrinter内部类中的val$beep实例变量仅在内部类的构造器中被设置一次。定义时没有初始化的final变量通常被称为空final（blank final）变量。

有时，final限制显得并不太方便。例如，假设想更新在一个封闭作用域内的计数器。这里想要统计一下在排序过程中调用compareTo方法的次数。

```
int counter = 0;
Date[] dates = new Date[100];
for (int i = 0; i < dates.length; i++)
   dates[i] = new Date()
      {
         public int compareTo(Date other)
         {
            counter++; // ERROR
            return super.compareTo(other);
         }
      };
Arrays.sort(dates);
System.out.println(counter + " comparisons.");
```

由于清楚地知道counter需要更新，所以不能将counter声明为final。由于Integer对象是不可变的，所以也不能用Integer代替它。补救的方法是使用一个长度为1的数组：

```
final int[] counter = new int[1];
for (int i = 0; i < dates.length; i++)
   dates[i] = new Date()
      {
         public int compareTo(Date other)
         {
            counter[0]++;
            return super.compareTo(other);
         }
      };
```

（数组变量仍然被声明为final，但是这仅仅表示不可以让它引用另外一个数组。数组中的数据元素可以自由地更改。）

在内部类被首次提出时，原型编译器对内部类中修改的局部变量自动地进行转换。然而，有些程序员对于编译器在背后生成堆对象感到十分惧怕，而采用final限制代替。在未来的Java

语言版本中有可能会修改这种策略。

6.4.6 匿名内部类

将局部内部类的使用再深入一步。假如只创建这个类的一个对象，就不必命名了。这种类被称为匿名内部类（anonymous inner class）。

```
public void start(int interval, final boolean beep)
{
   ActionListener listener = new ActionListener()
      {
         public void actionPerformed(ActionEvent event)
         {
             Date now = new Date();
             System.out.println("At the tone, the time is " + now);
             if (beep) Toolkit.getDefaultToolkit().beep();
         }
      };
   Timer t = new Timer(interval, listener);
   t.start();
}
```

这种语法确实有些难以理解。它的含义是：创建一个实现ActionListener接口的类的新对象，需要实现的方法actionPerformed定义在括号{}内。

用于构造对象的任何参数都要被放在超类名后面的括号( )内。通常的语法格式为：

```
new SuperType(construction parameters)
   {
      inner class methods and data
   }
```

其中，SuperType可以是ActionListener这样的接口，于是内部类就要实现这个接口。SuperType也可以是一个类，于是内部类就要扩展它。

由于构造器的名字必须与类名相同，而匿名类没有类名，所以，匿名类不能有构造器。取而代之的是，将构造器参数传递给超类（superclass）构造器。尤其是在内部类实现接口的时候，不能有任何构造参数。不仅如此，还要像下面这样提供一组括号：

```
new InterfaceType()
   {
      methods and data
   }
```

请仔细研究一下，看看构造一个类的新对象与构造一个扩展了那个类的匿名内部类的对象之间有什么差别。

```
Person queen = new Person("Mary");
   // a Person object
Person count = new Person("Dracula") { . . .};
   // an object of an inner class extending Person
```

如果构造参数的闭圆括号跟一个开花括号，正在定义的就是匿名内部类。

匿名内部类是一种好想法呢？还是一种让人迷惑不解的想法呢？也许两者兼有。如果内部类的代码比较短，例如，只有几行简单的代码，匿名内部类就可以节省一些录入的时间。但是

节省这点时间却会让人陷入“混乱的Java代码竞赛”。

例6-5包含了用匿名内部类实现语音时钟程序的全部源代码。将这个程序与例6-4相比较就会发现使用匿名内部类的解决方案比较简短、更切实际、更易于理解。

例6-5 AnonymousInnerClassTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.util.*;
4. import javax.swing.*;
5. import javax.swing.Timer;
6.
7. /**
8.  * This program demonstrates anonymous inner classes.
9.  * @version 1.10 2004-02-27
10.  * @author Cay Horstmann
11.  */
12. public class AnonymousInnerClassTest
13. {
14.    public static void main(String[] args)
15.    {
16.       TalkingClock clock = new TalkingClock();
17.       clock.start(1000, true);
18.
19.       // keep program running until user selects "Ok"
20.       JOptionPane.showMessageDialog(null, "Quit program?");
21.       System.exit(0);
22.    }
23. }
24.
25. /**
26.  * A clock that prints the time in regular intervals.
27.  */
28. class TalkingClock
29. {
30.    /**
31.     * Starts the clock.
32.     * @param interval the interval between messages (in milliseconds)
33.     * @param beep true if the clock should beep
34.     */
35.    public void start(int interval, final boolean beep)
36.    {
37.       ActionListener listener = new ActionListener()
38.          {
39.             public void actionPerformed(ActionEvent event)
40.             {
41.                Date now = new Date();
42.                System.out.println("At the tone, the time is " + now);
43.                if (beep) Toolkit.getDefaultToolkit().beep();
44.             }
45.          };
46.       Timer t = new Timer(interval, listener);
47.       t.start();
48.    }
49. }
```

6.4.7 静态内部类

有时候，使用内部类只是为了把一个类隐藏在另外一个类的内部，并不需要内部类引用外围类对象。为此，可以将内部类声明为static，以便取消产生的引用。

下面是一个使用静态内部类的典型例子。考虑一下计算数组中最小值和最大值的问题。当然，可以编写两个方法，一个方法用于计算最小值，另一个方法用于计算最大值。在调用这两个方法的时候，数组被遍历两次。如果只遍历数组一次，并能够同时计算出最小值和最大值，那么就可以大大地提高效率了。

```
double min = Double.MAX_VALUE;
double max = Double.MIN_VALUE;
for (double v : values)
{
   if (min > v) min = v;
   if (max < v) max = v;
}
```

然而，这个方法必须返回两个数值，为此，可以定义一个包含两个值的类pair：

```
class Pair
{
   public Pair(double f, double s)
   {
      first = f;
      second = s;
   }
   public double getFirst() { return first; }
   public double getSecond() {  return second; }

   private double first;
   private double second;
}
```

minmax方法可以返回一个Pair类型的对象。

```
class ArrayAlg
{
   public static Pair minmax(double[] values)
   {
      . . .
      return new Pair(min, max);
   }
}
```

这个方法的调用者可以34使用getFirst和getSecond方法获得答案：

```
Pair p = ArrayAlg.minmax(d);
System.out.println("min = " + p.getFirst());
System.out.println("max = " + p.getSecond());
```

Pair是一个十分大众化的名字。在大型项目中，除了定义包含一对字符串的Pair类之外，其他程序员也很可能使用这个名字。这样就会产生名字冲突。解决这个问题的办法是将Pair定义为ArrayAlg的内部公有类。此后，通过ArrayAlg.Pair访问它：

```
ArrayAlg.Pair p = ArrayAlg.minmax(d);
```

与前面例子中所使用的内部类不同，在Pair对象中不需要引用任何其他的对象，为此，可以将这个内部类声明为static：

```
class ArrayAlg
{
   public static class Pair
   {
      . . .
   }
   . . .
}
```

当然，只有内部类可以声明为static。静态内部类的对象除了没有对生成它的外围类对象的引用特权外，与其他所有内部类完全一样。在我们列举的示例中，必须使用静态内部类，这是由于内部类对象是在静态方法中构造的：

```
public static Pair minmax(double[] d)
{
    . . .
    return new Pair(min, max);
}
```

如果没有将Pair类声明为static，那么编译器将会给出错误报告：没有可用的隐式ArrayAlg类型对象初始化内部类对象。

注释：在内部类不需要访问外围类对象的时候，应该使用静态内部类。有些程序员用嵌套类（nested class）表示静态内部类。

注释：声明在接口中的内部类自动成为static和public。

例6-6包含ArrayAlg类和嵌套的Pair类的全部源代码。

例6-6 StaticInnerClassTest.java

```
1. /**
2.  * This program demonstrates the use of static inner classes.
3.  * @version 1.01 2004-02-27
4.  * @author Cay Horstmann
5.  */
6. public class StaticInnerClassTest
7. {
8.    public static void main(String[] args)
9.    {
10.      double[] d = new double[20];
11.      for (int i = 0; i < d.length; i++)
12.         d[i] = 100 * Math.random();
13.      ArrayAlg.Pair p = ArrayAlg.minmax(d);
14.      System.out.println("min = " + p.getFirst());
15.      System.out.println("max = " + p.getSecond());
16.   }
17. }
18.
19. class ArrayAlg
20. {
```

```
21.    /**
22.     * A pair of floating-point numbers
23.     */
24.    public static class Pair
25.    {
26.       /**
27.        * Constructs a pair from two floating-point numbers
28.        * @param f the first number
29.        * @param s the second number
30.        */
31.       public Pair(double f, double s)
32.       {
33.          first = f;
34.          second = s;
35.       }
36.
37.       /**
38.        * Returns the first number of the pair
39.        * @return the first number
40.        */
41.       public double getFirst()
42.       {
43.          return first;
44.       }
45.
46.       /**
47.        * Returns the second number of the pair
48.        * @return the second number
49.        */
50.       public double getSecond()
51.       {
52.          return second;
53.       }
54.
55.       private double first;
56.       private double second;
57.    }
58.
59.    /**
60.     * Computes both the minimum and the maximum of an array
61.     * @param values an array of floating-point numbers
62.     * @return a pair whose first element is the minimum and whose second element
63.     * is the maximum
64.     */
65.    public static Pair minmax(double[] values)
66.    {
67.       double min = Double.MAX_VALUE;
68.       double max = Double.MIN_VALUE;
69.       for (double v : values)
70.       {
71.          if (min > v) min = v;
72.          if (max < v) max = v;
73.       }
74.       return new Pair(min, max);
75.    }
76. }
```

6.5 代理

在本章的最后，讨论一下代理（proxy），这是Java SE 1.3新增加的特性。利用代理可以在运行时创建一个实现了一组给定接口的新类。这种功能只有在编译时无法确定需要实现哪个接口时才有必要使用。对于应用程序设计人员来说，遇到这种情况的机会很少。如果对这种高级技术不感兴趣，可以跳过本节内容。然而，对于系统程序设计人员来说，代理带来的灵活性却十分重要。

假设有一个表示接口的Class对象（有可能只包含一个接口），它的确切类型在编译时无法知道。这确实有些难度。要想构造一个实现这些接口的类，就需要使用newInstance方法或反射找出这个类的构造器。但是，不能实例化一个接口，需要在程序处于运行状态时定义一个新类。

为了解决这个问题，有些程序将会生成代码；将这些代码放置在一个文件中；调用编译器；然后再加载结果类文件。很自然，这样做的速度会比较慢，并且需要将编译器与程序放在一起。而代理机制则是一种更好的解决方案。代理类可以在运行时创建全新的类。这样的代理类能够实现指定的接口。尤其是，它具有下列方法：

- 指定接口所需要的全部方法。
- Object类中的全部方法，例如，toString、equals等。

然而，不能在运行时定义这些方法的新代码。而是要提供一个调用处理器（invocation handler）。调用处理器是实现了InvocationHandler接口的类对象。在这个接口中只有一个方法：

```
Object invoke(Object proxy, Method method, Object[] args)
```

无论何时调用代理对象的方法，调用处理器的invoke方法都会被调用，并向其传递Method对象和原始的调用参数。调用处理器必须给出处理调用的方式。

要想创建一个代理对象，需要使用Proxy类的newProxyInstance方法。这个方法有三个参数：

- 一个类加载器（class loader）。作为Java安全模型的一部分，对于系统类和从因特网上下载下来的类，可以使用不同的类加载器。有关类加载器的详细内容将在卷II第9章中讨论。目前，用null表示使用默认的类加载器。
- 一个Class对象数组，每个元素都是需要实现的接口。
- 一个调用处理器。

还有两个需要解决的问题。如何定义一个处理器？能够用结果代理对象做些什么？当然，这两个问题的答案取决于打算使用代理机制解决什么问题。使用代理可能出于很多原因，例如：

- 路由对远程服务器的方法调用。
- 在程序运行期间，将用户接口事件与动作关联起来。
- 为调试，跟踪方法调用。

在列举的示例中，使用代理和调用处理器跟踪方法调用，并且定义了一个TraceHander包装器类存储包装的对象。其中的invoke方法打印出被调用方法的名字和参数，随后用包装好的对象作为隐式参数调用这个方法。

```
class TraceHandler implements InvocationHandler
{
   public TraceHandler(Object t)
```

```
   {
      target = t;
   }

   public Object invoke(Object proxy, Method m, Object[] args)
      throws Throwable
   {
      // print method name and parameters
      . . .
      // invoke actual method
      return m.invoke(target, args);
   }

   private Object target;
}
```

下面说明一下如何构造用于跟踪方法调用的代理对象。

```
Object value = . . .;
// construct wrapper
InvocationHandler handler = new TraceHandler(value);
// construct proxy for one or more interfaces
Class[] interfaces = new Class[] { Comparable.class };
Object proxy = Proxy.newProxyInstance(null, interfaces, handler);
```

现在，无论何时用proxy调用某个方法，这个方法的名字和参数就会打印出来，之后再用value调用它。

在例6-7给出的程序中，使用代理对象对二分查找进行跟踪。这里，首先将用1～1000整数的代理填充数组，然后调用Arrays类中的binarySearch方法在数组中查找一个随机整数。最后，打印出与之匹配的元素。

```
Object[] elements = new Object[1000];
// fill elements with proxies for the integers 1  . . . 1000
for (int i = 0; i < elements.length; i++)
{
   Integer value = i + 1;
   elements[i] = Proxy.newInstance(. . .); // proxy for value;
}

// construct a random integer
Integer key = new Random().nextInt(elements.length) + 1;

// search for the key
int result = Arrays.binarySearch(elements, key);

// print match if found
if (result >= 0) System.out.println(elements[result]);
```

在上述代码中，Integer类实现了Comparable接口。代理对象属于在运行时定义的类（它有一个名字，如$Proxy0）。这个类也实现了Comparable接口。然而，它的compareTo方法调用了代理对象处理器的invoke方法。

注释：前面已经讲过，在Java SE 5.0中，Integer类实际上实现了Comparable<Integer>。然而，在运行时，所有的泛型类都被取消，代理将它们构造为原Comparable类的类对象。

binarySearch方法按下面这种方式调用：

```
if (elements[i].compareTo(key) < 0) . . .
```

由于数组中填充了代理对象，所以compareTo调用了TraceHander类中的invoke方法。这个方法打印出了方法名和参数，之后用包装好的Integer对象调用compareTo。

最后，在示例程序的结尾调用：

```
System.out.println(elements[result]);
```

println方法调用代理对象的toString，这个调用也会被重定向到调用处理器上。下面是程序运行的全部跟踪结果：

```
500.compareTo(288)
250.compareTo(288)
375.compareTo(288)
312.compareTo(288)
281.compareTo(288)
296.compareTo(288)
288.compareTo(288)
288.toString()
```

可以看出，二分查找算法查找关键字的过程，即每一步都将查找区间缩减一半。注意，即使不属于Comparable接口，toString方法也被代理。在下一节中会看到，有相当一部分的Object方法都被代理。

例6-7 ProxyTest.java

```
1. import java.lang.reflect.*;
2. import java.util.*;
3.
4. /**
5.  * This program demonstrates the use of proxies.
6.  * @version 1.00 2000-04-13
7.  * @author Cay Horstmann
8.  */
9. public class ProxyTest
10. {
11.    public static void main(String[] args)
12.    {
13.       Object[] elements = new Object[1000];
14.
15.       // fill elements with proxies for the integers 1 ... 1000
16.       for (int i = 0; i < elements.length; i++)
17.       {
18.          Integer value = i + 1;
19.          InvocationHandler handler = new TraceHandler(value);
20.          Object proxy = Proxy.newProxyInstance(null, new Class[] { Comparable.class } , handler);
21.          elements[i] = proxy;
22.       }
23.
24.       // construct a random integer
25.       Integer key = new Random().nextInt(elements.length) + 1;
26.
27.       // search for the key
28.       int result = Arrays.binarySearch(elements, key);
```

```
29.
30.       // print match if found
31.       if (result >= 0) System.out.println(elements[result]);
32.    }
33. }
34.
35. /**
36.  * An invocation handler that prints out the method name and parameters, then
37.  * invokes the original method
38.  */
39. class TraceHandler implements InvocationHandler
40. {
41.    /**
42.     * Constructs a TraceHandler
43.     * @param t the implicit parameter of the method call
44.     */
45.    public TraceHandler(Object t)
46.    {
47.       target = t;
48.    }
49.
50.    public Object invoke(Object proxy, Method m, Object[] args) throws Throwable
51.    {
52.       // print implicit argument
53.       System.out.print(target);
54.       // print method name
55.       System.out.print("." + m.getName() + "(");
56.       // print explicit arguments
57.       if (args != null)
58.       {
59.          for (int i = 0; i < args.length; i++)
60.          {
61.             System.out.print(args[i]);
62.             if (i < args.length - 1) System.out.print(", ");
63.          }
64.       }
65.       System.out.println(")");
66.
67.       // invoke actual method
68.       return m.invoke(target, args);
69.    }
70.
71.    private Object target;
72. }
```

代理类的特性

现在，我们已经看到了代理类的应用，接下来了解它们的一些特性。需要记住，代理类是在程序运行过程中创建的。然而，一旦被创建，就变成了常规类，与虚拟机中的任何其他类没有什么区别。

所有的代理类都扩展于Proxy类。一个代理类只有一个实例域——调用处理器，它定义在Proxy的超类中。为了履行代理对象的职责，所需要的任何附加数据都必须存储在调用处理器中。例如，在例6-7给出的程序中，代理Comparable对象时，TraceHandler包装了实际的对象。

所有的代理类都覆盖了Object类中的方法toString、equals和hashCode。如同所有的代理方法一样，这些方法仅仅调用了调用处理器的invoke。Object类中的其他方法（如clone和getClass）没有被重新定义。

没有定义代理类的名字，Sun虚拟机中的Proxy类将生成一个以字符串$Proxy开头的类名。

对于特定的类加载器和预设的一组接口来说，只能有一个代理类。也就是说，如果使用同一个类加载器和接口数组调用两次newProxyInstance方法的话，那么只能够得到同一个类的两个对象，也可以利用getProxyClass方法获得这个类：

```
Class proxyClass = Proxy.getProxyClass(null, interfaces);
```

代理类一定是public和final。如果代理类实现的所有接口都是public，代理类就不属于某个特定的包；否则，所有非公有的接口都必须属于同一个包，同时，代理类也属于这个包。

可以通过调用Proxy类中的isProxyClass方法检测一个特定的Class对象是否代表一个代理类。

API `java.lang.reflect.InvocationHandler` 1.3

- `Object invoke(Object proxy,Method method,Object[] args)`

  定义了代理对象调用方法时希望执行的动作。

API `java.lang.reflect.Proxy` 1.3

- `static Class getProxyClass(ClassLoader loader,Class[] interfaces)`

  返回实现指定接口的代理类。

- `static Object newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler handler)`

  构造一个实现指定接口的代理类的实例。所有方法都将调用给定处理器对象的invoke方法。

- `static boolean isProxyClass(Class c)`

  如果c是一个代理类返回true。

到此为止，Java程序设计语言的基础概念介绍完毕了。接口和内部类是两个经常使用的概念。然而，前面已经提到过，代理是一项工具构造者感兴趣的高级技术，对应用程序员来说，并不十分重要。下面准备继续学习由第7章开始介绍的图形和用户接口。