

## 第5章 继 承

类、超类和子类

Object：所有类的超类

泛型数组列表

对象包装器和自动打包

参数数量可变的方法

枚举类

反射

继承设计的技巧

第4章主要阐述了类和对象的概念，本章将学习面向对象程序设计的另外一个基本概念：继承（inheritance）。利用继承，人们可以基于已存在的类构造一个新类。继承已存在的类就是复用（继承）这些类的方法和域。在此基础上，还可以添加一些新的方法和域，以满足新的需求。这是Java程序设计中的一项核心技术。

与前一章相同，对于只使用过C、Visual Basic或COBOL这类面向过程的程序设计语言的读者来说，一定要仔细地阅读本章的内容。对于使用过C++或Smalltalk这类面向对象的程序设计语言的读者来说，对本章介绍的绝大部分内容可能已经比较熟悉。不过，Java与C++或其他面向对象的程序设计语言相比较，在实现继承的手段上存在着较大的差异。


另外，本章还阐述了反射（reflection）的概念。反射是指在程序运行期间发现更多的类及其属性的能力。这是一个功能强大的特性，使用起来也比较复杂。由于主要是开发软件工具的人员，而不是编写应用程序的人员对这项功能感兴趣，因此对于这部分内容，可以先浏览一下，待日后再返回来学习。

### 5.1 类、超类和子类

现在让我们重新回忆一下在前一章中讨论的Employee类。假设你在某个公司工作，这个公司中经理的待遇与普通雇员的待遇存在着一些差异。不过，他们之间也存在着很多相同的地方，例如，他们都领取薪水。只是普通雇员在完成本职工作之后仅领取薪水，而经理在完成了预期的业绩之后还能得到奖金。这种情形就需要使用继承。这是因为需要为经理定义一个新类Manager，以便增加一些新功能。但可以重用Employee类中已经编写的部分代码，并将其中的所有域保留下来。从理论上讲，在Manager与Employee之间存在着明显的“is-a”（是）关系，每个经理都是一名雇员：“is-a”关系是继承的一个明显特征。


下面是由继承Employee类来定义Manager类的格式，关键字extends表示继承。

```
class Manager extends Employee
{
    添加方法和域
}
```

 **C++注释：**Java与C++定义继承类的方式十分相似。Java用关键字`extends`代替了C++中的冒号（`:`）。在Java中，所有的继承都是公有继承，而没有C++中的私有继承和保护继承。

关键字`extends`表明正在构造的新类派生于一个已存在的类。已存在的类被称为超类（`superclass`）、基类（`base class`）或父类（`parent class`）；新类被称为子类（`subclass`）、派生类（`derived class`）或孩子类（`child class`）。超类和子类是Java程序员最常用的两个术语，而其他语言种类的程序员可能更加偏爱使用父类和孩子类，这些都是继承时使用的术语。

尽管`Employee`类是一个超类，但并不是因为它位于子类之上或者拥有比子类更多的功能。恰恰相反，子类比超类拥有的功能更加丰富。例如，读过`Manager`类的源代码之后就会发现，`Manager`类比超类`Employee`封装了更多的数据，拥有更多的功能。

 **注释：**前缀“超”和“子”来源于计算机科学和数学理论中的集合语言的术语。所有雇员组成的集合包含所有经理组成的集合。可以这样说，雇员集合是经理集合的超集，也可以说，经理集合是雇员集合的子集。

在`Manager`类中，增加了一个用于存储奖金信息的域，以及一个用于设置这个域的方法：

```
class Manager extends Employee
{
    . . .

    public void setBonus(double b)
    {
        bonus = b;
    }

    private double bonus;
}
```

这里定义的方法和域并没有什么特别之处。如果有一个`Manager`对象，就可以使用`setBonus`方法。

```
Manager boss = . . .;
boss.setBonus(5000);
```

当然，由于`setBonus`方法不是在`Employee`类中定义的，所以属于`Employee`类的对象不能使用它。

然而，尽管在`Manager`类中没有显式地定义`getName`和`getHireDay`等方法，但属于`Manager`类的对象却可以使用它们，这是因为`Manager`类自动地继承了超类`Employee`中的这些方法。

同样，从超类中还继承了`name`、`salary`和`hireDay`这3个域。这样一来，每个`Manager`类对象就包含了4个域：`name`、`salary`、`hireDay`和`bonus`。

在通过扩展超类定义子类的时候，仅需要指出子类与超类的不同之处。因此在设计类的时候，应该将通用的方法放在超类中，而将具有特殊用途的方法放在子类中，这种将通用的功能放到超类的做法，在面向对象程序设计中十分普遍。

然而，超类中的有些方法对子类`Manager`并不一定适用。例如，在`Manager`类中的`getSalary`方法应该返回薪水和奖金的总和。为此，需要提供一个新的方法来覆盖（`override`）超类中的这个方法：

```
class Manager extends Employee
{
```

```
...
public double getSalary()
{
    ...
}
...
```

应该如何实现这个方法呢？乍看起来似乎很简单，只要返回salary和bonus域的总和就可以了：

```
public double getSalary()
{
    return salary + bonus; // won't work
}
```

然而，这个方法并不能运行。这是因为Manager类的getSalary方法不能够直接地访问超类的私有域。也就是说，尽管每个Manager对象都拥有一个名为salary的域，但在Manager类的getSalary方法中并不能够直接地访问salary域。只有Employee类的方法才能够访问私有部分。如果Manager类的方法一定要访问私有域，就必须借助于公有的接口，Employee类中的公有方法getSalary正是这样一个接口。

现在，再试一下。将对salary域的访问替换成调用getSalary方法。

```
public double getSalary()
{
    double baseSalary = getSalary(); // still won't work
    return baseSalary + bonus;
}
```

上面这段代码仍然不能运行。问题出现在调用getSalary的语句上，这是因为Manager类也有一个getSalary方法（就是正在实现的这个方法），所以这条语句将会导致无限次地调用自己，直到整个程序崩溃为止。

这里需要指出：我们希望调用超类Employee中的getSalary方法，而不是当前类的这个方法。为此，可以使用特定的关键字super解决这个问题：

```
super.getSalary()
```


上述语句调用的是Employee类中的getSalary方法。下面是Manager类中getSalary方法的正确书写格式：

```
public double getSalary()
{
    double baseSalary = super.getSalary();
    return baseSalary + bonus;
}
```



注释：有些人认为super与this引用是类似的概念，实际上，这样比较并不太恰当。这是因为super不是一个对象的引用，不能将super赋给另一个对象变量，它只是一个指示编译器调用超类方法的特有关键字。

正像前面所看到的那样，在子类中可以增加域、增加方法或覆盖超类的方法，然而绝对不能删除继承的任何域和方法。

 **C++注释：**在Java中使用关键字super调用超类的方法，而在C++中则采用超类名加上 :: 操作符的形式。例如，在Manager类的getSalary方法中，应该将super.getSalary替换为Employee::getSalary。

最后，看一下super在构造器中的应用。

```
public Manager(String n, double s, int year, int month, int day)
{
    super(n, s, year, month, day);
    bonus = 0;
}
```


这里的关键字super具有不同的含义。语句


```
super(n, s, year, month, day);
```

是“调用超类Employee中含有n、s、year、month和day参数的构造器”的简写形式。

由于Manager类的构造器不能访问Employee类的私有域，所以必须利用Employee类的构造器对这部分私有域进行初始化，我们可以通过super实现对超类构造器的调用。使用super调用构造器的语句必须是子类构造器的第一条语句。

如果子类的构造器没有显式地调用超类的构造器，则将自动地调用超类默认（没有参数）的构造器。如果超类没有不带参数的构造器，并且在子类的构造器中又没有显式地调用超类的其他构造器，则Java编译器将报告错误。

 **注释：**回忆一下，关键字this有两个用途：一是引用隐式参数，二是调用该类其他的构造器。同样，super关键字也有两个用途：一是调用超类的方法，二是调用超类的构造器。在调用构造器的时候，这两个关键字的使用方式很相似。调用构造器的语句只能作为另一个构造器的第一条语句出现。构造参数既可以传递给本类（this）的其他构造器，也可以传递给超类（super）的构造器。

 **C++注释：**在C++的构造函数中，使用初始化列表语法调用超类的构造函数，而不调用super。在C++中，Manager的构造函数如下所示：

```
Manager::Manager(String n, double s, int year, int month, int day) // C++
: Employee(n, s, year, month, day)
{
    bonus = 0;
}
```

重新定义Manager对象的getSalary方法之后，奖金就会自动地添加到经理的薪水中。

下面给出一个例子，其功能为创建一个新经理，并设置他的奖金：

```
Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
boss.setBonus(5000);
```

下面定义一个包含3个雇员的数组：

```
Employee[] staff = new Employee[3];
```

将经理和雇员都放到数组中：

```
staff[0] = boss;
staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
```

```
staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
```

输出每个人的薪水：

```
for (Employee e : staff)
    System.out.println(e.getName() + " " + e.getSalary());
```

运行这条循环语句将会输出下列结果：

```
Carl Cracker 85000.0
Harry Hacker 50000.0
Tommy Tester 40000.0
```

这里的staff[1]和staff[2]仅输出了基本薪水，这是因为它们对应的是Employee对象，而staff[0]对应的是Manager对象，它的getSalary方法将奖金与基本薪水加在了一起。

需要提到的是，e.getSalary()能够确定应该执行哪个getSalary方法。请注意，尽管这里将e声明为Employee类型，但实际上e既可以引用Employee类型的对象，也可以引用Manager类型的对象。

当e引用Employee对象时，e.getSalary()调用的是Employee类中的getSalary方法；当e引用Manager对象时，e.getSalary()调用的是Manager类中的getSalary方法。虚拟机知道e实际引用的对象类型，因此能够正确地调用相应的类方法。

一个对象变量（例如，变量e）可以引用多种实际类型的现象被称为多态（polymorphism）。在运行时能够自动地选择调用哪个方法的现象称为动态绑定（dynamic binding）。在本章中将详细地讨论这两个概念。



**C++注释：**在Java中，不需要将方法声明为虚拟方法。动态绑定是默认的处理方式。如果不希望让一个方法具有虚拟特征，可以将它标记为final（稍后将介绍关键字final）。

例5-1的程序展示了Employee对象与Manager对象在薪水计算上的区别。

#### 例5-1 ManagerTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates inheritance.
5.  * @version 1.21 2004-02-21
6.  * @author Cay Horstmann
7.  */
8. public class ManagerTest
9. {
10.     public static void main(String[] args)
11.     {
12.         // construct a Manager object
13.         Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
14.         boss.setBonus(5000);
15.
16.         Employee[] staff = new Employee[3];
17.
18.         // fill the staff array with Manager and Employee objects
19.
20.         staff[0] = boss;
21.         staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
```

```
22.     staff[2] = new Employee("Tommy Tester", 40000, 1990, 3, 15);
23.
24.     // print out information about all Employee objects
25.     for (Employee e : staff)
26.         System.out.println("name=" + e.getName() + ",salary=" + e.getSalary());
27. }
28. }
29.
30. class Employee
31. {
32.     public Employee(String n, double s, int year, int month, int day)
33.     {
34.         name = n;
35.         salary = s;
36.         GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
37.         hireDay = calendar.getTime();
38.     }
39.
40.     public String getName()
41.     {
42.         return name;
43.     }
44.
45.     public double getSalary()
46.     {
47.         return salary;
48.     }
49.
50.     public Date getHireDay()
51.     {
52.         return hireDay;
53.     }
54.
55.     public void raiseSalary(double byPercent)
56.     {
57.         double raise = salary * byPercent / 100;
58.         salary += raise;
59.     }
60.
61.     private String name;
62.     private double salary;
63.     private Date hireDay;
64. }
65.
66. class Manager extends Employee
67. {
68.     /**
69.      * @param n the employee's name
70.      * @param s the salary
71.      * @param year the hire year
72.      * @param month the hire month
73.      * @param day the hire day
74.      */
75.     public Manager(String n, double s, int year, int month, int day)
76.     {
77.         super(n, s, year, month, day);
78.         bonus = 0;
```

```
79.  }
80.
81.  public double getSalary()
82.  {
83.      double baseSalary = super.getSalary();
84.      return baseSalary + bonus;
85.  }
86.
87.  public void setBonus(double b)
88.  {
89.      bonus = b;
90.  }
91.
92.  private double bonus;
93. }
```

### 5.1.1 继承层次

继承并不限于一个层次。例如，可以由Manager类派生Executive类。由一个公共超类派生出来的所有类的集合被称为继承层次（inheritance hierarchy），如图5-1所示。在继承层次中，从某个特定的类到其祖先的路径被称为该类的继承链（inheritance chain）。

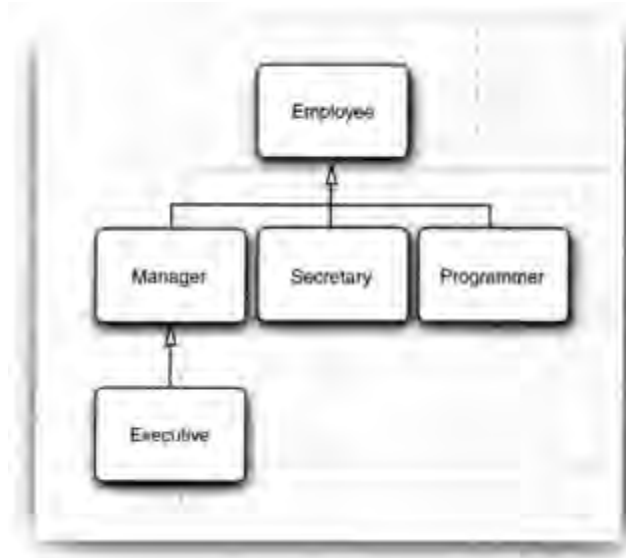


图5-1 Employee继承层次

通常，一个祖先类可以拥有多个子孙继承链。例如，可以由Employee类派生出子类Programmer或Secretary，它们与Manager类没有任何关系（有可能它们彼此之间也没有任何关系）。必要的话，可以将这个过程一直延续下去。



C++注释：Java不支持多继承。有关Java中多继承功能的实现方式，请参看下一章有关接口的讨论。

### 5.1.2 多态

有一个用来判断是否应该设计为继承关系的简单规则，这就是“is-a”规则，它表明子类

的每个对象也是超类的对象。例如，每个经理都是雇员，因此，将Manager类设计为Employee类的子类是显而易见的，反之不然，并不是每一名雇员都是经理。

“is-a”规则的另一种表述法是置换法则。它表明程序中出现超类对象的任何地方都可以用子类对象置换。例如，可以将一个子类的对象赋给超类变量。

```
Employee e;  
e = new Employee(. . .); // Employee object expected  
e = new Manager(. . .); // OK, Manager can be used as well
```

在Java程序设计语言中，对象变量是多态的。一个Employee变量既可以引用一个Employee类对象，也可以引用一个Employee类的任何一个子类的对象（例如，Manager、Executive等等）。

从例5-1中，已经看到了置换法则的优点：

```
Manager boss = new Manager(. . .);  
Employee[] staff = new Employee[3];  
staff[0] = boss;
```

在这个例子中，变量staff[0]与boss引用同一个对象。但编译器将staff[0]看成Employee对象。

这意味着，可以这样调用

```
boss.setBonus(5000); // OK
```

但不能这样调用

```
staff[0].setBonus(5000); // ERROR
```

这是因为staff[0]声明的类型是Employee，而setBonus不是Employee类的方法。

然而，不能将一个超类的引用赋给子类变量。例如，下面的赋值是非法的

```
Manager m = staff[i]; // ERROR
```

原因很清楚：不是所有的雇员都是经理。如果赋值成功，m有可能引用了一个不是经理的Employee对象，当在后面调用m.setBonus(...)时就有可能发生运行时错误。



**警告：**在Java中，子类数组的引用可以转换成超类数组的引用，而不需要采用强制类型转换。例如，下面是一个经理数组

```
Manager[] managers = new Manager[10];
```

将它转换成Employee[]数组完全是合法的：

```
Employee[] staff = managers; // OK
```

这样做肯定不会有问题，请思考一下其中的缘由。毕竟，如果manager[i]是一个Manager，也一定是一个Employee。然而，实际上，将会发生一些令人惊讶的事情。要切记managers和staff引用的是同一个数组。现在看一下这条语句：

```
staff[0] = new Employee("Harry Hacker", ...);
```

编译器竟然接纳了这个赋值操作。但在这里，staff[0]与manager[0]引用的是同一个对象，似乎我们把一个普通雇员擅自归入经理行列中了。这是一种很忌讳发生的情形，当调用managers[0].setBonus(1000)的时候，将会导致调用一个不存在的实例域，进而搅乱相邻存储空间的内容。

为了确保不发生这类错误，所有数组都要牢记创建它们的元素类型，并负责监督仅将类型兼容的引用存储到数组中。例如，使用new managers[10] 创建的数组是一个经理数组。



如果试图存储一个Employee类型的引用就会引发ArrayStoreException异常。

### 5.1.3 动态绑定

弄清调用对象方法的执行过程十分重要。下面是调用过程的详细描述：

1) 编译器查看对象的声明类型和方法名。假设调用x.f(param)，且隐式参数x声明为C类的对象。需要注意的是：有可能存在多个名字为f，但参数类型不一样的方法。例如，可能存在方法f(int)和方法f(String)。编译器将会一一列举所有C类中名为f的方法和其超类中访问属性为public且名为f的方法。

至此，编译器已获得所有可能被调用的候选方法。

2) 接下来，编译器将查看调用方法时提供的参数类型。如果在所有名为f的方法中存在一个与提供的参数类型完全匹配，就选择这个方法。这个过程被称为重载解析 (overloading resolution)。例如，对于调用x.f(“Hello”)来说，编译器将会挑选f(String)，而不是f(int)。由于允许类型转换 (int可以转换成double，Manager可以转换成Employee，等等)，所以这个过程可能很复杂。如果编译器没有找到与参数类型匹配的方法，或者发现经过类型转换后有多个方法与之匹配，就会报告一个错误。

至此，编译器已获得需要调用的方法名字和参数类型。



注释：前面曾经说过，方法的名字和参数列表称为方法的签名。例如，f(int)和f(String)是两个具有相同名字，不同签名的方法。如果在子类中定义了一个与超类签名相同的方法，那么子类中的这个方法就覆盖了超类中的这个相同签名的方法。

不过，返回类型不是签名的一部分，因此，在覆盖方法时，一定要保证返回类型的兼容性。在Java SE 5.0以前的版本中，要求返回类型必须是一样的。现在允许子类将覆盖方法的返回类型定义为原返回类型的子类型。例如，假设Employee类有

```
public Employee getBuddy() { ... }
```

在后面的子类Manager中，可以按照如下所示的方式覆盖这个方法

```
public Manager getBuddy() { ... } // OK in Java SE 5.0
```

我们说，这两个getBuddy方法具有可协变的返回类型。

3) 如果是private方法、static方法、final方法 (有关final修饰符的含义将在下一节讲述) 或者构造器，那么编译器将可以准确地知道应该调用哪个方法，我们将这种调用方式称为静态绑定 (static binding)。与此对应的是，调用的方法依赖于隐式参数的实际类型，并且在运行时实现动态绑定。在我们列举的示例中，编译器采用动态绑定的方式生成一条调用f(String)的指令。

4) 当程序运行，并且采用动态绑定调用方法时，虚拟机一定调用与x所引用对象的实际类型最合适的那个类的方法。假设x的实际类型是D，它是C类的子类。如果D类定义了方法f(String)，就直接调用它；否则，将在D类的超类中寻找f(String)，以此类推。

每次调用方法都要进行搜索，时间开销相当大。因此，虚拟机预先为每个类创建了一个方法表 (method table)，其中列出了所有方法的签名和实际调用的方法。这样一来，在真正调用方法的时候，虚拟机仅查找这个表就行了。在前面的例子中，虚拟机搜索D类的方法表，以便寻找与调用f(String)相匹配的方法。这个方法既有可能是D.f(String)，也有可能是X.f(String)，

这里的X是D的超类。这里需要提醒一点，如果调用`super.f(param)`，编译器将对隐式参数超类的方法表进行搜索。

现在，查看一下例5-1中调用`e.getSalary()`的详细过程。`e`声明为`Employee`类型。`Employee`类只有一个名叫`getSalary`的方法，这个方法没有参数。因此，在这里不必担心重载解析的问题。

由于`getSalary`不是`private`方法、`static`方法或`final`方法，所以将采用动态绑定。虚拟机为`Employee`和`Manager`两个类生成方法表。在`Employee`的方法表中，列出了这个类定义的所有方法：

```
Employee:
  getName() -> Employee.getName()
  getSalary() -> Employee.getSalary()
  getHireDay() -> Employee.getHireDay()
  raiseSalary(double) -> Employee.raiseSalary(double)
```

实际上，上面列出的方法并不完整，稍后会看到`Employee`类有一个超类`Object`，`Employee`类从这个超类中还继承了许多方法，在此，我们略去了这些方法。

`Manager`方法表稍微有些不同。其中有三个方法是继承而来的，一个方法是重新定义的，还有一个方法是新增加的。

```
Manager:
  getName() -> Employee.getName()
  getSalary() -> Manager.getSalary()
  getHireDay() -> Employee.getHireDay()
  raiseSalary(double) -> Employee.raiseSalary(double)
  setBonus(double) -> Manager.setBonus(double)
```

在运行的时候，调用`e.getSalary()`的解析过程为：

1) 首先，虚拟机提取`e`的实际类型的方法表。既可能是`Employee`、`Manager`的方法表，也可能是`Employee`类的其他子类的方法表。

2) 接下来，虚拟机搜索定义`getSalary`签名的类。此时，虚拟机已经知道应该调用哪个方法。

3) 最后，虚拟机调用方法。

动态绑定有一个非常重要的特性：无需对现存的代码进行修改，就可以对程序进行扩展。假设增加一个新类`Executive`，并且变量`e`有可能引用这个类的对象，我们不需要对包含调用`e.getSalary()`的代码进行重新编译。如果`e`恰好引用一个`Executive`类的对象，就会自动地调用`Executive.getSalary()`方法。



**警告：**在覆盖一个方法的时候，子类方法不能低于超类方法的可见性。特别是，如果超类方法是`public`，子类方法一定要声明为`public`。经常会发生这类错误：在声明子类方法的时候，遗漏了`public`修饰符。此时，编译器将会把它解释为试图降低访问权限。

#### 5.1.4 阻止继承：`final`类和方法

有时候，可能希望阻止人们利用某个类定义子类。不允许扩展的类被称为`final`类。如果在定义类的时候使用了`final`修饰符就表明这个类是`final`类。例如，假设希望阻止人们定义`Executive`类的子类，就可以在定义这个类的时候，使用`final`修饰符声明。声明格式如下所示：

```
final class Executive extends Manager
{
```

```
    ...  
}
```

类中的方法也可以被声明为final。如果这样做，子类就不能覆盖这个方法（final类中的所有方法自动地成为final方法）。例如

```
class Employee  
{  
    ...  
    public final String getName()  
    {  
        return name;  
    }  
    ...  
}
```



注释：前面曾经说过，域也可以被声明为final。对于final域来说，构造对象之后就不允许改变它们的值了。不过，如果将一个类声明为final，只有其中的方法自动地成为final，而不包括域。

将方法或类声明为final主要鉴于以下原因：确保它们不会在子类中改变语义。例如，Calendar类中的getTime和setTime方法都声明为final。这表明Calendar类的设计者负责实现Date类与日历状态之间的转换，而不允许子类处理这些问题。同样地，String类也是final类，这意味着不允许任何人定义String的子类。换言之，如果有一个String的引用，它引用的一定是一个String对象，而不可能是其他类的对象。

有些程序员认为：除非有足够的理由使用多态性，应该将所有的方法都声明为final。事实上，在C++和C#中，如果没有特别地说明，所有的方法都不具有多态性。这两种做法可能都有些偏激。我们提倡在设计类层次时，仔细地思考应该将哪些方法和类声明为final。

在早期的Java中，有些程序员为了避免动态绑定带来的系统开销而使用final关键字。如果一个方法没有被覆盖并且很短，编译器就能够对它进行优化处理，这个过程称为内联（inlining）。例如，内联调用e.getName()将被替换为访问e.name域。这是一项很有意义的改进，这是由于CPU在处理调用方法的指令时，使用的分支转移会扰乱预取指令的策略，所以，这被视为不受欢迎的。然而，如果getName在另外一个类中被覆盖，那么编译器就无法知道覆盖的代码将会做什么操作，因此也就不能对它进行内联处理了。

幸运的是，虚拟机中的即时编译器比传统编译器的处理能力强得多。这种编译器可以准确地知道类之间的继承关系，并能够检测出类中是否真正地存在覆盖给定的方法。如果方法很短、被频繁调用且没有真正地被覆盖，那么即时编译器就会将这个方法进行内联处理。如果虚拟机加载了另外一个子类，而在这个子类中包含了对内联方法的覆盖，那么将会发生什么情况呢？优化器将取消对覆盖方法的内联。这个过程很慢，但却很少发生。



C++注释：C++中的方法在默认情况下不是动态绑定的，而是利用inline标记将方法调用替换成方法的源代码。不过，在C++中，没有提供任何机制阻止一个子类覆盖超类的方法。在C++中，也能够定义一个不允许被派生的类，但这需要使用一定的技巧，并且这样做也没有什么意义（这个问题将作为练习留给读者。提示：使用虚基类）。

### 5.1.5 强制类型转换

第3章曾经讲过，将一个类型强制转换成另外一个类型的过程被称为类型转换。Java程序设计语言提供了一种专门用于进行类型转换的表示法。例如：

```
double x = 3.405;
int nx = (int) x;
```

将表达式x的值转换成整数类型，舍弃了小数部分。

正像有时候需要将浮点型数值转换成整型数值一样，有时候也可能需要将某个类的对象引用转换成另外一个类的对象引用。对象引用的转换语法与数值表达式的类型转换类似，仅需要用一对圆括号将目标类名括起来，并放置在需要转换的对象引用之前就可以了。例如：

```
Manager boss = (Manager) staff[0];
```

进行类型转换的惟一原因是：在暂时忽视对象的实际类型之后，使用对象的全部功能。例如，在managerTest类中，由于某些项是普通雇员，所以staff数组必须是Employee对象的数组。我们需要将数组中引用经理的元素复原成Manager类，以便能够访问新增加的所有变量（需要注意，在前面的示例代码中，为了避免类型转换，我们做了一些特别的处理，即将boss变量存入数组之前，先用Manager对象对它进行初始化。而为了设置经理的奖金，必须使用正确的类型）。

大家知道，在Java中，每个对象变量都属于一个类型。类型描述了这个变量所引用的以及能够引用的对象类型。例如，staff[i]引用一个Employee对象（因此它还可以引用Manager对象）。

将一个值存入变量时，编译器将检查是否允许该操作。将一个子类的引用赋给一个超类变量，编译器是允许的。但将一个超类的引用赋给一个子类变量，必须进行类型转换，这样才能通过运行时的检查。

如果试图在继承链上进行向下的类型转换，并且“谎报”有关对象包含的内容，会发生什么情况呢？

```
Manager boss = (Manager) staff[1]; // ERROR
```

运行这个程序时，Java运行时系统将报告这个错误，并产生一个ClassCastException异常。如果没有捕获这个异常，那么程序就会终止。因此，应该养成这样一个良好的程序设计习惯：在进行类型转换之前，先查看一下是否能够成功地转换。这个过程简单地使用instanceof运算符就可以实现。例如：

```
if (staff[1] instanceof Manager)
{
    boss = (Manager) staff[1];
    ...
}
```

最后，如果这个类型转换不可能成功，编译器就不会进行这个转换。例如，下面这个类型转换：

```
Date c = (Date) staff[1];
```

将会产生编译性错误，这是因为Date不是Employee的子类。

综上所述：

- 只能在继承层次内进行类型转换。

- 在将超类转换成子类之前，应该使用instanceof进行检查。



注释：如果x为null，进行下列测试

```
x instanceof C
```

不会产生异常，只是返回false。之所以这样处理是因为null没有引用任何对象，当然也不会引用C类型的对象。

实际上，通过类型转换调整对象的类型并不是一种好的做法。在我们列举的示例中，大多数情况并不需要将Employee对象转换成Manager对象，两个类的对象都能够正确地调用getSalary方法，这是因为实现多态性的动态绑定机制能够自动地找到相应的方法。

只有在使用Manager中特有的方法时才需要进行类型转换，例如，setBonus方法。如果鉴于某种原因，发现需要通过Employee对象调用setBonus方法，那么就应该检查一下超类的设计是否合理。重新设计一下超类，并添加setBonus方法才是正确的选择。请记住，只要没有捕获ClassCastException异常，程序就会终止执行。在一般情况下，应该尽量少用类型转换和instanceof运算符。



C++注释：Java使用的类型转换语法来源于C语言“糟糕的旧时期”，但处理过程却有些像C++的dynamic\_cast操作。例如，

```
Manager boss = (Manager) staff[1]; // Java
```

等价于

```
Manager* boss = dynamic_cast<Manager*>(staff[1]); // C++
```

它们之间只有一点重要的区别：当类型转换失败时，Java不会生成一个null对象，而是抛出一个异常。从这个意义上讲，有点像C++中的引用（reference）转换。真是令人生厌。在C++中，可以在一个操作中完成类型测试和类型转换。

```
Manager* boss = dynamic_cast<Manager*>(staff[1]); // C++  
if (boss != NULL) . . .
```

而在Java中，需要将instanceof运算符和类型转换组合起来使用：

```
if (staff[1] instanceof Manager)  
{  
    Manager boss = (Manager) staff[1];  
    . . .  
}
```

### 5.1.6 抽象类

如果自下而上仰视类的继承层次结构，位于上层的类更具有通用性，甚至可能更加抽象。从某种角度看，祖先类更加通用，人们只将它作为派生其他类的基类，而不作为想使用的特定的实例类。例如，考虑一下对Employee类层次的扩展。一名雇员是一个人，一名学生也是一个人。下面将类Person和类Student添加到类的层次结构中。图5-2是这三个类之间的关系层次图。

为什么要花费精力进行这样高层次的抽象呢？每个人都有一些诸如姓名这样的属性。学生与雇员都有姓名属性，因此可以将getName方法放置在位于继承关系较高层次的通用超类中。

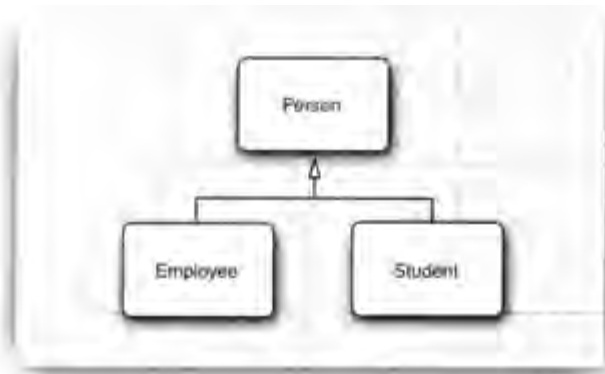


图5-2 Person与子类的关系层次示意图

现在，再增加一个getDescription方法，它可以返回对一个人的简短描述。例如：

an employee with a salary of \$50,000.00  
a student majoring in computer science

在Employee类和Student类中实现这个方法很容易。但是在Person类中应该提供什么内容呢？除了姓名之外，Person类一无所知。当然，可以让Person.getDescription()返回一个空字符串。然而，还有一个更好的方法，就是使用abstract关键字，这样就完全不需要实现这个方法了。

```
public abstract String getDescription();  
// no implementation required
```

为了提高程序的清晰度，包含一个或多个抽象方法的类本身必须被声明为抽象的。

```
abstract class Person  
{  
    ...  
    public abstract String getDescription();  
}
```

除了抽象方法之外，抽象类还可以包含具体数据和具体方法。例如，Person类还保存着姓名和一个返回姓名的具体方法。

```
abstract class Person  
{  
    public Person(String n)  
    {  
        name = n;  
    }  
  
    public abstract String getDescription();  
  
    public String getName()  
    {  
        return name;  
    }  
  
    private String name;  
}
```



提示：许多程序员认为，在抽象类中不能包含具体方法。建议尽量将通用的域和方法（不管是否是抽象的）放在超类（不管是否是抽象的）中。

抽象方法充当着占位的角色，它们的具体实现在子类中。扩展抽象类可以有两种选择。一种是在子类中定义部分抽象方法或抽象方法也不定义，这样就必须将子类也标记为抽象类；另一种是定义全部的抽象方法，这样一来，子类就不是抽象的了。

例如，通过扩展抽象Person类，并实现getDescription方法来定义Student类。由于在Student类中不再含有抽象方法，所以不必将这个类声明为抽象的。

类即使不含抽象方法，也可以将类声明为抽象类。

抽象类不能被实例化。也就是说，如果将一个类声明为abstract，就不能创建这个类的对象。例如，表达式

```
new Person("Vince Vu")
```

是错误的，但可以创建一个具体子类的对象。

需要注意，可以定义一个抽象类的对象变量，但是它只能引用非抽象子类的对象。例如，  
`Person p = new Student("Vince Vu", "Economics");`

这里的p是一个抽象类person的变量，它引用了一个非抽象子类Student的实例。



C++注释：在C++中，有一种在尾部用=0标记的抽象方法，被称为纯虚函数，例如：

```
class Person // C++
{
public:
    virtual string getDescription() = 0;
    ...
};
```

只要有一个纯虚函数，这个类就是抽象类。在C++中，没有提供用于表示抽象类的特殊关键字。

下面通过抽象类Person扩展一个具体子类Student：

```
class Student extends Person
{
    public Student(String n, String m)
    {
        super(n);
        major = m;
    }

    public String getDescription()
    {
        return "a student majoring in " + major;
    }

    private String major;
}
```

在Student类中定义了getDescription方法。因此，在Student类中的全部方法都是非抽象的，这个类不再是抽象类。

在例5-2的程序中定义了抽象超类Person和两个具体子类Employee和Student。下面将雇员和学生对象赋给Person引用的数组。

```
Person[] people = new Person[2];
people[0] = new Employee(. . .);
people[1] = new Student(. . .);
```

然后，输出这些对象的姓名和信息描述：

```
for (Person p : people)
    System.out.println(p.getName() + ", " + p.getDescription());
```

有些人可能对下面这个调用感到困惑：

```
p.getDescription()
```

这不是调用了一个没有定义的方法吗？请牢记，由于不能构造抽象类Person的对象，所以变量p永远不会引用Person对象，而是引用诸如Employee或Student这样的具体子类对象，而在这些对象中都定义了getDescription方法。

是否可以省略Person超类中的抽象方法，而仅在Employee和Student子类中定义getDescription方法呢？如果这样的话，就不能通过变量p调用getDescription方法了。编译器只允许调用在类中声明的方法。

在Java程序设计语言中，抽象方法是一个重要的概念。在接口（interface）中将会看到更多的抽象方法。有关接口的详细介绍请参看第6章。

#### 例5-2 PersonTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates abstract classes.
5.  * @version 1.01 2004-02-21
6.  * @author Cay Horstmann
7.  */
8. public class PersonTest
9. {
10.     public static void main(String[] args)
11.     {
12.         Person[] people = new Person[2];
13.
14.         // fill the people array with Student and Employee objects
15.         people[0] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
16.         people[1] = new Student("Maria Morris", "computer science");
17.
18.         // print out names and descriptions of all Person objects
19.         for (Person p : people)
20.             System.out.println(p.getName() + ", " + p.getDescription());
21.     }
22. }
23.
24. abstract class Person
25. {
26.     public Person(String n)
27.     {
28.         name = n;
29.     }
30.
31.     public abstract String getDescription();
```



```
32.
33. public String getName()
34. {
35.     return name;
36. }
37.
38. private String name;
39. }
40.
41. class Employee extends Person
42. {
43.     public Employee(String n, double s, int year, int month, int day)
44.     {
45.         super(n);
46.         salary = s;
47.         GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
48.         hireDay = calendar.getTime();
49.     }
50.
51.     public double getSalary()
52.     {
53.         return salary;
54.     }
55.
56.     public Date getHireDay()
57.     {
58.         return hireDay;
59.     }
60.
61.     public String getDescription()
62.     {
63.         return String.format("an employee with a salary of $%.2f", salary);
64.     }
65.
66.     public void raiseSalary(double byPercent)
67.     {
68.         double raise = salary * byPercent / 100;
69.         salary += raise;
70.     }
71.
72.     private double salary;
73.     private Date hireDay;
74. }
75.
76. class Student extends Person
77. {
78.     /**
79.      * @param n the student's name
80.      * @param m the student's major
81.      */
82.     public Student(String n, String m)
83.     {
84.         // pass n to superclass constructor
85.         super(n);
86.         major = m;
87.     }
88.
```

```
89. public String getDescription()  
90. {  
91.     return "a student majoring in " + major;  
92. }  
93.  
94. private String major;  
95. }
```

### 5.1.7 受保护访问

大家都知道，最好将类中的域标记为private，而方法标记为public。任何声明为private的内容对其他类都是不可见的。前面已经看到，这对于子类来说也完全适用，即子类也不能访问超类的私有域。

然而，在有些时候，人们希望超类中的某些方法允许被子类访问，或允许子类的方法访问超类的某个域。为此，需要将这些方法或域声明为protected。例如，如果将超类Employee中的hireDay声明为protected，而不是私有的，Manager中的方法就可以直接地访问它。

不过，Manager类中的方法只能够访问Manager对象中的hireDay域，而不能访问其他Employee对象中的这个域。这种限制有助于避免滥用受保护机制，使得子类只能获得访问受保护域的权利。

在实际应用中，要谨慎使用protected属性。假设需要将设计的类提供给其他程序员使用，而在这个类中设置了一些受保护域，由于其他程序员可以由这个类再派生出新类，并访问其中的受保护域。在这种情况下，如果需要对这个类的实现进行修改，就必须通知所有使用这个类的程序员。这违背了OOP提倡的数据封装原则。

受保护的方法更具有实际意义。如果需要限制某个方法的使用，就可以将它声明为protected。这表明子类（可能很熟悉祖先类）得到信任，可以正确地使用这个方法，而其他类则不行。

这种方法的一个最好的示例就是Object类中的clone方法，有关它的详细内容请参看第6章。



C++注释：事实上，Java中的受保护部分对所有子类及同一个包中的所有其他类都可见。这与C++中的保护机制稍有不同，Java中的protected概念要比C++中的安全性差。

下面归纳一下Java用于控制可见性的4个访问修饰符：

- 1) 仅对本类可见——private。
- 2) 对所有类可见——public。
- 3) 对本包和所有子类可见——protected。

4) 对本包可见——默认，所谓默认是指没有标明任何修饰符的情况，这是一种不太受欢迎的形式。

## 5.2 Object：所有类的超类

Object类是Java中所有类的最终祖先，在Java中每个类都是由它扩展而来的。但是并不需要这样写：

```
class Employee extends Object
```

如果没有明确地指出超类，Object就被认为是这个类的超类。由于在Java中，每个类都是由Object类扩展而来的，所以，熟悉这个类提供的所有服务十分重要。本章将介绍一些基本的内容，没有提到的部分请参看后面的章节或在线文档（在Object中有几个只在处理线程时才会被调用的方法，有关线程内容请参看卷 Ⅱ）。

可以使用Object类型的变量引用任何类型的对象：

```
Object obj = new Employee("Harry Hacker", 35000);
```

当然，Object类型的变量只能用于作为各种值的通用持有者。要想对其中的内容进行具体的操作，还需要清楚对象的原始类型，并进行相应的类型转换：

```
Employee e = (Employee) obj;
```

在Java中，只有基本类型（primitive types）不是对象，例如，数值、字符和布尔类型的值都不是对象。所有的数组类型，不管是对象数组还是基本类型的数组都扩展于Object类。

```
Employee[] staff = new Employee[10];  
obj = staff; // OK  
obj = new int[10]; // OK
```



**C++注释：**在C++中没有根类，不过，每个指针都可以转换成void\*。

### 5.2.1 Equals方法

Object类中的equals方法用于检测一个对象是否等于另外一个对象。在Object类中，这个方法将判断两个对象是否具有相同的引用。如果两个对象具有相同的引用，它们一定是相等的。从这点上看，将其作为默认操作也是合乎情理的。然而，对于多数类来说，这种判断并没有什么意义。例如，采用这种方式比较两个PrintStream对象是否相等就完全没有意义。然而，经常需要检测两个对象状态的相等性，如果两个对象的状态相等，就认为这两个对象是相等的。

例如，如果两个雇员对象的姓名、薪水和雇佣日期都一样，就认为它们是相等的（在实际的雇员数据库中，比较ID更有意义。利用下面这个示例演示equals方法的实现技巧）。

```
class Employee  
{  
    ...  
    public boolean equals(Object otherObject)  
    {  
        // a quick test to see if the objects are identical  
        if (this == otherObject) return true;  
  
        // must return false if the explicit parameter is null  
        if (otherObject == null) return false;  
  
        // if the classes don't match, they can't be equal  
        if (getClass() != otherObject.getClass())  
            return false;  
  
        // now we know otherObject is a non-null Employee  
        Employee other = (Employee) otherObject;
```

```

        // test whether the fields have identical values
        return name.equals(other.name)
            && salary == other.salary
            && hireDay.equals(other.hireDay);
    }
}

```

getClass方法将返回一个对象所属的类，有关这个方法的详细内容稍后进行介绍。在检测中，只有在两个对象属于同一个类时，才有可能相等。

在子类中定义equals方法时，首先调用超类的equals。如果检测失败，对象就不可能相等。如果超类中的域都相等，就需要比较子类中的实例域。

```

class Manager extends Employee
{
    . . .
    public boolean equals(Object otherObject)
    {
        if (!super.equals(otherObject)) return false;
        // super.equals checked that this and otherObject belong to the same class
        Manager other = (Manager) otherObject;
        return bonus == other.bonus;
    }
}

```

### 5.2.2 相等测试与继承

如果隐式和显式的参数不属于同一个类，equals方法将如何处理呢？这是一个很有争议的问题。在前面的例子中，如果发现类不匹配，equals方法就返回false。但是，许多程序员却喜欢使用instanceof进行检测：

```
if (!(otherObject instanceof Employee)) return false;
```

这样做不但没有解决otherObject是子类的情况，并且还有可能会招致一些麻烦。这就是建议不要使用这种处理方式的原因所在。Java语言规范要求equals方法具有下面的特性：

- 1) 自反性：对于任何非空引用x，x.equals(x) 应该返回true。
- 2) 对称性：对于任何引用x和y，当且仅当y.equals(x)返回true，x.equals(y)也应该返回true。
- 3) 传递性：对于任何引用x、y和z，如果x.equals(y)返回true，y.equals(z)返回true，x.equals(z) 也应该返回true。
- 4) 一致性：如果x和y引用的对象没有发生变化，反复调用x.equals(y) 应该返回同样的结果。
- 5) 对于任意非空引用x，x.equals(null) 应该返回false。

这些规则十分合乎情理，从而避免了类库实现者在数据结构中定位一个元素时还要考虑调用x.equals(y)，还是调用y.equals(x)的问题。

然而，就对称性来说，当参数不属于同一个类的时候需要仔细地思考一下。请看下面这个调用：

```
e.equals(m)
```

这里的e是一个Employee对象，m是一个Manager对象，并且两个对象具有相同的姓名、薪水和雇佣日期。如果在Employee.equals中用instanceof进行检测，则返回true。然而这意味着反过来

调用：

```
m.equals(e)
```

也需要返回true。对称性不允许这个方法调用返回false，或者抛出异常。

这就使得Manager类受到了束缚。这个类的equals方法必须能够用自己与任何一个Employee对象进行比较，而不考虑经理拥有的那部分特有信息！猛然间会让人感觉instanceof测试并不是完美无暇。

某些书的作者认为不应该利用getClass检测，因为这样不符合置换原则。有一个应用AbstractSet类的equals方法的典型例子，它将检测两个集合是否有相同的元素。AbstractSet类有两个具体子类：TreeSet和HashSet，它们分别使用不同的算法实现查找集合元素的操作。无论集合采用何种方式实现，都需要拥有对任意两个集合进行比较的功能。

然而，集合是相当特殊的一个例子，应该将AbstractSet.equals声明为final，这是因为没有任何一个子类需要重定义集合是否相等的语义（事实上，这个方法并没有被声明为final。这样做，可以让子类选择更加有效的算法对集合进行是否相等的检测）。

下面可以从两个截然不同的情况看一下这个问题：

- 如果子类能够拥有自己的相等概念，则对称性需求将强制采用getClass进行检测。
- 如果由超类决定相等的概念，那么就可以使用instanceof进行检测，这样可以在不同子类的对象之间进行相等的比较。

在雇员和经理的例子中，只要对应的域相等，就认为两个对象相等。如果两个Manager对象所对应的姓名、薪水和雇佣日期均相等，而奖金不相等，就认为它们是不相同的，因此，可以使用getClass检测。

但是，假设使用雇员的ID作为相等的检测标准，并且这个相等的概念适用于所有的子类，就可以使用instanceof进行检测，并应该将Employee.equals声明为final。



注释：在标准Java库中包含150多个equals方法的实现，包括使用instanceof检测、调用getClass检测、捕获ClassCastException或者什么也不做。

下面给出编写一个完美的equals方法的建议：

- 1) 显式参数命名为otherObject，稍后需要将它转换成另一个叫做other的变量。
- 2) 检测this与otherObject是否引用同一个对象：

```
if (this == otherObject) return true;
```

这条语句只是一个优化。实际上，这是一种经常采用的形式。因为计算这个等式要比一个一个地比较类中的域所付出的代价小得多。

- 3) 检测otherObject是否为null，如果为null，返回false。这项检测是很必要的。

```
if (otherObject == null) return false;
```

4) 比较this与otherObject是否属于同一个类。如果equals的语义在每个子类中有所改变，就使用getClass检测：

```
if (getClass() != otherObject.getClass()) return false;
```

如果所有的子类都拥有统一的语义，就使用instanceof检测：

```
if (!(otherObject instanceof ClassName)) return false;
```

5) 将otherObject转换为相应的类类型变量：

```
ClassName other = (ClassName) otherObject
```

6) 现在开始对所有需要比较的域进行比较了。使用 == 比较基本类型域，使用equals比较对象域。如果所有的域都匹配，就返回true；否则返回false。

```
return field1 == other.field1
    && field2.equals(other.field2)
    && . . .;
```

如果在子类中重新定义equals，就要在其中包含调用super.equals(other)。



提示：对于数组类型的域，可以使用静态的Arrays.equals方法检测相应的数组元素是否相等。



警告：下面是实现equals方法的一种常见的错误。可以找到其中的问题吗？

```
public class Employee
{
    public boolean equals(Employee other)
    {
        return name.equals(other.name)
            && salary == other.salary
            && hireDay.equals(other.hireDay);
    }
    ...
}
```

这个方法声明的显式参数类型是Employee。其结果并没有覆盖Object类的equals方法，而是定义了一个完全无关的方法。

从Java SE 5.0开始，为了避免发生类型错误，可以使用@Override对覆盖超类的方法进行标记：

```
@Override public boolean equals(Object other)
```

如果出现了错误，并且正在定义一个新方法，编译器就会给出错误报告。例如，假设将下面的声明添加到Employee类中：

```
@Override public boolean equals(Employee other)
```

就会看到一个错误报告，这是因为这个方法并没有覆盖超类Object中的任何方法。



java.util.Arrays 1.2

- static Boolean equals(type[] a, type[] b) 5.0

如果两个数组长度相同，并且在对应的位置上数据元素也均相同，将返回true。数组的元素类型可以是Object,int,long,short,char,byte,boolean,float或double。

### 5.2.3 hashCode方法

散列码 (hash code) 是由对象导出的一个整型值。散列码是没有规律的。如果x和y是两个不同的对象，x.hashCode()与y.hashCode()基本上不会相同。在表5-1中列出了几个通过调用String类的hashCode方法得到的散列码。

String类使用下列算法计算散列码：

```
int hash = 0;
for (int i = 0; i < length(); i++)
    hash = 31 * hash + charAt(i);
```

由于hashCode方法定义在Object类中，因此每个对象都有一个默认的散列码，其值为对象的存储地址。看一下下面这个例子。

```
String s = "Ok";
StringBuilder sb = new StringBuilder(s);
System.out.println(s.hashCode() + " " + sb.hashCode());
String t = new String("Ok");
StringBuilder tb = new StringBuilder(t);
System.out.println(t.hashCode() + " " + tb.hashCode());
```

表5-2列出了结果。

表5-1 调用hashCode函数得到的散列码

String	Hash Code
Hello	69609650
Harry	69496448
Hacker	- 2141031506

表5-2 String和String Buffers的散列码

Object	Hash Code
s	2556
sb	20526976
t	2556
tb	20527144

请注意，字符串s与t拥有相同的散列码，这是因为字符串的散列码是由内容导出的。而字符串缓冲sb与tb却有着不同的散列码，这是因为在StringBuffer类中没有定义hashCode方法，它的散列码是由Object类的默认hashCode方法导出的对象存储地址。

如果重新定义equals方法，就必须重新定义hashCode方法，以使用户可以将对象插入到散列表中（有关散列表的内容将在卷II的第2章中讨论）。

HashCode方法应该返回一个整型数值（也可以是负数），并合理地组合实例域的散列码，以便能够让各个不同的对象产生的散列码更加均匀。

例如，下面是Employee类的hashCode方法。

```
class Employee
{
    public int hashCode()
    {
        return 7 * name.hashCode()
            + 11 * new Double(salary).hashCode()
            + 13 * hireDay.hashCode();
    }
    ...
}
```

Equals与hashCode的定义必须一致：如果x.equals(y)返回true，那么x.hashCode()就必须与y.hashCode()具有相同的值。例如，如果用定义的Employee.equals比较雇员的ID，那么hashCode方法就需要散列ID，而不是雇员的姓名或存储地址。



提示：如果存在数组类型的域，那么可以使用静态的Arrays.hashCode方法计算一个散列码，这个散列码由数组元素的散列码组成。

**API** java.lang.Object 1.0

## • int hashCode()

返回对象的散列码。散列码可以是任意的整数，包括正数或负数。两个相等的对象要求返回相等的散列码。

**API** java.util.Arrays 1.2

## • static int hashCode(type[] a) 5.0

计算数组a的散列码。组成这个数组的元素类型可以是object, int, long, short, char, byte, boolean, float或double。

### 5.2.4 ToString方法

在Object中还有一个重要的方法，就是toString方法，它用于返回表示对象值的字符串。下面是一个典型的例子。Point类的toString方法将返回下面这样的字符串：

```
java.awt.Point[x=10,y=20]
```

绝大多数（但不是全部）的toString方法都遵循这样的格式：类的名字，随后是一对方括号括起来的域值。下面是Employee类中的toString方法的实现：

```
public String toString()
{
    return "Employee[name=" + name
        + ",salary=" + salary
        + ",hireDay=" + hireDay
        + "];"
}
```

实际上，还可以设计得更好一些。最好通过调用getClass().getName() 获得类名的字符串，而不要将类名硬加到toString方法中：

```
public String toString()
{
    return getClass().getName()
        + "[name=" + name
        + ",salary=" + salary
        + ",hireDay=" + hireDay
        + "];"
}
```

toString方法也可以供子类调用。

当然，设计子类的程序员也应该定义自己的toString方法，并将子类域的描述添加进去。如果超类使用了getClass().getName()，那么子类只要调用super.toString()就可以了。例如，下面是Manager类中的toString方法：

```
class Manager extends Employee
{
    . . .
    public String toString()
    {
        return super.toString()
    }
}
```



```
        + "[bonus=" + bonus
        + "]\n";
    }
}
```

现在，Manager对象将打印输出如下所示的内容：

```
Manager[name=...,salary=...,hireDay=...][bonus=...]
```

随处可见toString方法的主要原因是：只要对象与一个字符串通过操作符“+”连接起来，Java编译就会自动地调用toString方法，以便获得这个对象的字符串描述。例如，

```
Point p = new Point(10, 20);
String message = "The current position is " + p;
// automatically invokes p.toString()
```



**提示：**在调用x.toString()的地方可以用""+x替代。这条语句将一个空串与x的字符串表示相连接。这里的x就是x.toString()。与toString不同的是，如果x是基本类型，这条语句照样能够执行。

如果x是任意一个对象，并调用

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

println方法就会直接地调用x.toString()，并打印输出得到的字符串。

Object类定义了toString方法，用来打印输出对象所属的类名和散列码。例如，调用

```
System.out.println(System.out)
```

将输出下列内容：

```
java.io.PrintStream@2f6684
```

之所以得到这样的结果是因为PrintStream类的设计者没有覆盖toString方法。



**警告：**令人烦恼的是，数组继承了object类的toString方法，数组类型将按照旧的格式打印。例如：

```
int[] luckyNumbers = { 2, 3, 5, 7, 11, 13 };
String s = "" + luckyNumbers;
```

生成字符串 “[I@1a46e30 ”（前缀[I表明是一个整型数组）。修正的方式是调用静态方法Arrays.toString。代码：

```
String s = Arrays.toString(luckyNumbers);
```

将生成字符串 “[2,3,5,7,11,13] ”。

要想打印多维数组（即，数组的数组）则需要调用Arrays.deepToString方法。

toString方法是一种非常有用的调试工具。在标准类库中，许多类都定义了toString方法，以使用户能够获得一些有关对象状态的必要信息。像下面这样显示调试信息非常有益：

```
System.out.println("Current position = " + position);
```

读者在第11章中将可以看到，更好的解决方法是：

```
Logger.global.info("Current position = " + position);
```



**提示：**强烈建议为自定义的每一个类增加toString方法。这样做不仅自己受益，而且所有使用这个类的程序员也会受益匪浅。

例5-3的程序实现了Employee类和Manager类的equals、hashCode和toString方法。

例5-3 EqualsTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates the equals method.
5.  * @version 1.11 2004-02-21
6.  * @author Cay Horstmann
7.  */
8. public class EqualsTest
9. {
10.     public static void main(String[] args)
11.     {
12.         Employee alice1 = new Employee("Alice Adams", 75000, 1987, 12, 15);
13.         Employee alice2 = alice1;
14.         Employee alice3 = new Employee("Alice Adams", 75000, 1987, 12, 15);
15.         Employee bob = new Employee("Bob Brandson", 50000, 1989, 10, 1);
16.
17.         System.out.println("alice1 == alice2: " + (alice1 == alice2));
18.
19.         System.out.println("alice1 == alice3: " + (alice1 == alice3));
20.
21.         System.out.println("alice1.equals(alice3): " + alice1.equals(alice3));
22.
23.         System.out.println("alice1.equals(bob): " + alice1.equals(bob));
24.
25.         System.out.println("bob.toString(): " + bob);
26.
27.         Manager carl = new Manager("Carl Cracker", 80000, 1987, 12, 15);
28.         Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
29.         boss.setBonus(5000);
30.         System.out.println("boss.toString(): " + boss);
31.         System.out.println("carl.equals(boss): " + carl.equals(boss));
32.         System.out.println("alice1.hashCode(): " + alice1.hashCode());
33.         System.out.println("alice3.hashCode(): " + alice3.hashCode());
34.         System.out.println("bob.hashCode(): " + bob.hashCode());
35.         System.out.println("carl.hashCode(): " + carl.hashCode());
36.     }
37. }
38.
39. class Employee
40. {
41.     public Employee(String n, double s, int year, int month, int day)
42.     {
43.         name = n;
44.         salary = s;
45.         GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
46.         hireDay = calendar.getTime();
47.     }
48.
49.     public String getName()
50.     {
51.         return name;
52.     }
53. }
```

```
54. public double getSalary()
55. {
56.     return salary;
57. }
58.
59. public Date getHireDay()
60. {
61.     return hireDay;
62. }
63.
64. public void raiseSalary(double byPercent)
65. {
66.     double raise = salary * byPercent / 100;
67.     salary += raise;
68. }
69.
70. public boolean equals(Object otherObject)
71. {
72.     // a quick test to see if the objects are identical
73.     if (this == otherObject) return true;
74.
75.     // must return false if the explicit parameter is null
76.     if (otherObject == null) return false;
77.
78.     // if the classes don't match, they can't be equal
79.     if (getClass() != otherObject.getClass()) return false;
80.
81.     // now we know otherObject is a non-null Employee
82.     Employee other = (Employee) otherObject;
83.
84.     // test whether the fields have identical values
85.     return name.equals(other.name) && salary == other.salary && hireDay.equals(other.hireDay);
86. }
87.
88. public int hashCode()
89. {
90.     return 7 * name.hashCode() + 11 * new Double(salary).hashCode() + 13 * hireDay.hashCode();
91. }
92.
93. public String toString()
94. {
95.     return getClass().getName() + "[name=" + name + ",salary=" + salary + ",hireDay=" + hireDay
96.         + "]\n";
97. }
98.
99. private String name;
100. private double salary;
101. private Date hireDay;
102. }
103.
104. class Manager extends Employee
105. {
106.     public Manager(String n, double s, int year, int month, int day)
107.     {
108.         super(n, s, year, month, day);
109.         bonus = 0;
110.     }
```

```
111.
112. public double getSalary()
113. {
114.     double baseSalary = super.getSalary();
115.     return baseSalary + bonus;
116. }
117.
118.
119. public void setBonus(double b)
120. {
121.     bonus = b;
122. }
123.
124. public boolean equals(Object otherObject)
125. {
126.     if (!super.equals(otherObject)) return false;
127.     Manager other = (Manager) otherObject;
128.     // super.equals checked that this and other belong to the same class
129.     return bonus == other.bonus;
130. }
131.
132. public int hashCode()
133. {
134.     return super.hashCode() + 17 * new Double(bonus).hashCode();
135. }
136.
137. public String toString()
138. {
139.     return super.toString() + "[bonus=" + bonus + "]";
140. }
141.
142. private double bonus;
143. }
```

#### java.lang.Object 1.0

- `Class getClass()`  
返回包含对象信息的类对象。稍后会看到Java提供了类运行时的描述，它的内容被封装在Class类中。
- `boolean equals(Object otherObject)`  
比较两个对象是否相等，如果两个对象指向同一块存储区域，方法返回true；否则方法返回false。在自定义的类中，应该覆盖这个方法。
- `String toString()`  
返回描述该对象值的字符串。在自定义的类中，应该覆盖这个方法。
- `Object clone()`  
创建一个对象的副本。Java运行时系统将为新实例分配存储空间，并将当前的对象复制到这块存储区域中。



注释：复制对象是非常重要的，然而，却常常让人陷入一种相当棘手的境地，一不小心就落入陷阱。有关clone方法的详细内容将在第6章中阐述。

**API** java.lang.Class 1.0

- String getName()  
返回这个类的名字。
- Class getSuperclass()  
以Class对象的形式返回这个类的超类信息。

### 5.3 泛型数组列表

在许多程序设计语言中，特别是在C语言中，必须在编译时就确定整个数组的大小。程序员对此十分反感，因为这样做将迫使程序员做出一些不情愿的折中。例如，在一个部门中有多少雇员？肯定不会超过100人。一旦出现一个拥有150名雇员的大型部门呢？愿意为那些仅有10名雇员的部门浪费90名雇员占据的存储空间吗？

在Java中，情况就好多了。它允许在运行时确定数组的大小。


```
int actualSize = . . . ;  
Employee[] staff = new Employee[actualSize];
```


当然，这段代码并没有完全解决运行时动态更改数组的问题。一旦确定了数组的大小，改变它就不太容易了。在Java中，解决这个问题最简单的方法是使用Java中另外一个被称为ArrayList的类。它使用起来有点像数组，但在添加或删除元素时，具有自动调节数组容量的功能，而不需要为此编写任何代码。

在Java SE 5.0中，ArrayList是一个采用类型参数( type parameter )的泛型类( generic class )。为了指定数组列表保存的元素对象类型，需要用一对尖括号将类名括起来加在后面，例如，ArrayList <Employee>。在第13章中可以看到如何自定义一个泛型类，这里并不需要了解任何技术细节就可以使用ArrayList类型。

下面声明和构造一个保存Employee对象的数组列表：

```
ArrayList<Employee> staff = new ArrayList<Employee>();
```

 注释：Java SE 5.0以前的版本没有提供泛型类，而是有一个ArrayList类，其中保存类型为Object的元素，它是“自适应大小”的集合。如果一定要使用老版本的Java，则需要将所有的后缀<...>删掉。在Java SE 5.0以后的版本中，没有后缀<...>仍然可以使用ArrayList，它将被认为是一个删去了类型参数的“原始”类型。

 注释：在Java程序设计语言的老版本中，程序员使用Vector类实现动态数组。不过，ArrayList类更加有效，没有任何理由一定要使用Vector类。

使用add方法可以将元素添加到数组列表中。例如，下面展示了如何将雇员对象添加到数组列表中的方法：

```
staff.add(new Employee("Harry Hacker", . . . ));  
staff.add(new Employee("Tony Tester", . . . ));
```

数组列表管理着对象引用的一个内部数组。最终，数组的全部空间有可能被用尽。这就显现出数组列表的操作魅力：如果调用add且内部数组已经满了，数组列表就将自动地创建一个

更大的数组，并将所有的对象从较小的数组中拷贝到较大的数组中。

如果已经清楚或能够估计出数组可能存储的元素数量，就可以在填充数组之前调用 `ensureCapacity` 方法：

```
staff.ensureCapacity(100);
```

这个方法调用将分配一个包含100个对象的内部数组。然后调用100次 `add`，而不用重新分配空间。

另外，还可以把初始容量传递给 `ArrayList` 构造器：

```
ArrayList<Employee> staff = new ArrayList<Employee>(100);
```

**X** 警告：分配数组列表，如下所示：

```
new ArrayList<Employee>(100) // capacity is 100
```

它与为新数组分配空间有所不同：

```
new Employee[100] // size is 100
```

数组列表的容量与数组的大小有一个非常重要的区别。如果为数组分配100个元素的存储空间，数组就有100个空位置可以使用。而容量为100个元素的数组列表只是拥有保存100个元素的潜力（实际上，重新分配空间的话，将会超过100），但是在最初，甚至完成初始化构造之后，数组列表根本就不含有任何元素。

`size` 方法将返回数组列表中包含的实际元素数目。例如，

```
staff.size()
```

将返回 `staff` 数组列表的当前元素数量，它等价于数组 `a` 的 `a.length`。

一旦能够确认数组列表的大小不再发生变化，就可以调用 `trimToSize` 方法。这个方法将存储区域的大小调整为当前元素数量所需要的存储空间数目。垃圾回收器将回收多余的存储空间。

一旦整理了数组列表的大小，添加新元素就需要花时间再次移动存储块，所以应该在确认不会添加任何元素时，再调用 `trimToSize`。

**C++** C++ 注释： `ArrayList` 类似于 C++ 的 `vector` 模板。 `ArrayList` 与 `vector` 都是泛型类型。但是 C++ 的 `vector` 模板为了便于访问元素重载了 `[]` 运算符。由于 Java 没有运算符重载，所以必须调用显式的方法。此外，C++ 向量是值拷贝。如果 `a` 和 `b` 是两个向量，赋值操作 `a = b` 将会构造一个与 `b` 长度相同的新向量 `a`，并将所有的元素由 `b` 拷贝到 `a`，而在 Java 中，这条赋值语句的操作结果是让 `a` 和 `b` 引用同一个数组列表。

#### **API** java.util.ArrayList<T> 1.2

- `ArrayList<T>()`  
构造一个空数组列表。
- `ArrayList<T>(int initialCapacity)`  
用指定容量构造一个空数组列表。  
参数： `initialCapacity`      数组列表的最初容量
- `boolean add(T obj)`  
在数组列表的尾端添加一个元素。永远返回 `true`。

参数：obj                      添加的元素

- `int size()`

返回存储在数组列表中的当前元素数量。（这个值将小于或等于数组列表的容量。）

- `void ensureCapacity(int capacity)`

确保数组列表在不重新分配存储空间的情况下就能够保存给定数量的元素。

参数：capacity                  需要的存储容量

- `void trimToSize()`

将数组列表的存储容量削减到当前尺寸。

### 5.3.1 访问数组列表元素

很遗憾，天下没有免费的午餐。数组列表自动扩展容量的便利增加了访问元素语法的复杂程度。其原因是`ArrayList`类并不是Java程序设计语言的一部分；它只是一个由某些人编写且被放在标准库中的一个实用类。

使用`get`和`set`方法实现访问或改变数组元素的操作，而不使用人们喜爱的`[]`语法格式。

例如，要设置第*i*个元素，可以使用：

```
staff.set(i, harry);
```

它等价于对数组*a*的元素赋值（数组的下标从0开始）：

```
a[i] = harry;
```



**警告：**只有*i*小于或等于数组列表的大小时，才能够调用`list.set(i,x)`。例如，下面这段代码是错误的：

```
ArrayList<Employee> list = new ArrayList<Employee>(100); // capacity 100, size 0
list.set(0, x); // no element 0 yet
```

使用`add`方法为数组添加新元素，而不要使用 `set`方法，它只能替换数组中已经存在的元素内容。

使用下列格式获得数组列表的元素：

```
Employee e = staff.get(i);
```

等价于：

```
Employee e = a[i];
```



**注释：**Java SE 5.0以前的版本没有泛型类，并且原始的`ArrayList`类提供的`get`方法毫无选择地返回`Object`，因此，`get`方法的调用者必须对返回值进行类型转换：

```
Employee e = (Employee) staff.get(i);
```

原始的`ArrayList`存在一定的危险性。它的`add`和`set`方法允许接受任意类型的对象。对于下面这个调用

```
staff.set(i, new Date());
```

编译不会给出任何警告，只有在检索对象并试图对它进行类型转换时，才会发现有问题。如果使用`ArrayList<Employee>`，编译器就会检测到这个错误。

下面这个技巧可以一举两得，既可以灵活地扩展数组，又可以方便地访问数组元素。首先，创建一个数组，并添加所有的元素。

```
ArrayList<X> list = new ArrayList<X>();
while (. . .)
{
    X = . . . ;
    list.add(x);
}
```

执行完上述操作后，使用toArray方法将数组元素拷贝到一个数组中。

```
X[] a = new X[list.size()];
list.toArray(a);
```

除了在数组列表的尾部追加元素之外，还可以在数组列表的中间插入元素，使用带索引参数的add方法。

```
int n = staff.size() / 2;
staff.add(n, e);
```

为了插入一个新元素，位于n之后的所有元素都要向后移动一个位置。如果插入新元素后，数组列表的大小超过了容量，数组列表就会被重新分配存储空间。

同样地，可以从数组列表中删除一个元素。

```
Employee e = staff.remove(n);
```

位于这个位置之后的所有元素都向前移动一个位置，并且数组的大小减1。

对数组实施插入和删除元素的操作其效率比较低。对于小型数组来说，这一点不必担心。但如果数组存储的元素数比较多，又经常需要在中间位置插入、删除元素，就应该考虑使用链表了。有关链表操作的实现方式将在第13章中讲述。

在Java SE 5.0中，可以使用“for each”循环对数组列表遍历：

```
for (Employee e : staff)
    do something with e
```

这个循环和下列代码具有相同的效果

```
for (int i = 0; i < staff.size(); i++)
{
    Employee e = staff.get(i);
    do something with e
}
```

例5-4是对第4章中EmployeeTest做出修改后的程序。在这里，将Employee[]数组替换成了ArrayList<Employee>。请注意下面的变化：

- 不必指出数组的大小。
- 使用add将任意多的元素添加到数组中。
- 使用size()替代length计算元素的数目。
- 使用a.get(i)替代a[i]访问元素。

#### 例5-4 ArrayListTest.java

```
1. import java.util.*;
```



```
2.
3. /**
4.  * This program demonstrates the ArrayList class.
5.  * @version 1.1 2004-02-21
6.  * @author Cay Horstmann
7.  */
8. public class ArrayListTest
9. {
10.     public static void main(String[] args)
11.     {
12.         // fill the staff array list with three Employee objects
13.         ArrayList<Employee> staff = new ArrayList<Employee>();
14.
15.         staff.add(new Employee("Carl Cracker", 75000, 1987, 12, 15));
16.         staff.add(new Employee("Harry Hacker", 50000, 1989, 10, 1));
17.         staff.add(new Employee("Tony Tester", 40000, 1990, 3, 15));
18.
19.         // raise everyone's salary by 5%
20.         for (Employee e : staff)
21.             e.raiseSalary(5);
22.
23.         // print out information about all Employee objects
24.         for (Employee e : staff)
25.             System.out.println("name=" + e.getName() + ",salary=" + e.getSalary() + ",hireDay="
26.                                 + e.getHireDay());
27.     }
28. }
29.
30. class Employee
31. {
32.     public Employee(String n, double s, int year, int month, int day)
33.     {
34.         name = n;
35.         salary = s;
36.         GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
37.         hireDay = calendar.getTime();
38.     }
39.
40.     public String getName()
41.     {
42.         return name;
43.     }
44.
45.     public double getSalary()
46.     {
47.         return salary;
48.     }
49.
50.     public Date getHireDay()
51.     {
52.         return hireDay;
53.     }
54.
55.     public void raiseSalary(double byPercent)
56.     {
57.         double raise = salary * byPercent / 100;
58.         salary += raise;
```

```
59.     }  
60.  
61.     private String name;  
62.     private double salary;  
63.     private Date hireDay;  
64. }
```

#### java.util.ArrayList<T> 1.2

- void set(int index, T obj)  
设置数组列表指定位置的元素值，这个操作将覆盖这个位置的原有内容。  
参数：index      位置（必须介于0~size() - 1之间）  
         obj      新的值
- T get(int index)  
获得指定位置的元素值。  
参数：index      获得的元素位置（必须介于0~size() - 1之间）
- void add(int index, T obj)  
向后移动元素，以便插入元素。  
参数：index      插入位置（必须介于0~size() - 1之间）  
         obj      新元素
- T remove(int index)  
删除一个元素，并将后面的元素向前移动。被删除的元素由返回值返回。  
参数：index      被删除的元素位置（必须介于0~size() - 1之间）

### 5.3.2 类型化与原始数组列表的兼容性

在采用Java SE 5.0以后的版本编写程序代码时，应该使用类型参数的数组列表，例如，ArrayList <Employee>。然而，有可能会发生与现存程序中的原始ArrayList类型进行交叉操作的情形。

假设有下面这个遗留下来的类：

```
public class EmployeeDB  
{  
    public void update(ArrayList list) { ... }  
    public ArrayList find(String query) { ... }  
}
```

可以将一个类型化的数组列表传递给update方法，而并不需要进行任何类型转换。

```
ArrayList<Employee> staff = ...;  
employeeDB.update(staff);
```

也可以将staff对象传递给update方法。



**警告：**尽管编译器没有给出任何错误信息或警告，但是这样调用并不太安全。在update方法中，添加到数组列表中的元素可能不是Employee类型。在对这些元素进行检索时就会出现异常。听起来似乎很吓人，但思考一下就会发现，这与Java SE 5.0以前的版本是一

样的。虚拟机的完整性绝对没有受到威胁。在这种情形下，既没有降低安全性，也没有受益于编译时的检查。

相反地，将一个原始ArrayList赋给一个类型化ArrayList会得到一个警告。

```
ArrayList<Employee> result = employeeDB.find(query); // yields warning
```



注释：为了能够看到警告性错误的文字信息，要将编译选项置为-Xlint:unchecked。

使用类型转换并不能避免出现警告。

```
ArrayList<Employee> result = (ArrayList<Employee>)  
    employeeDB.find(query); // yields another warning
```

这样，将会得到另外一个警告信息，被告之类型转换有误。

这就是Java中不尽人意的参数化类型的限制所带来的结果。鉴于兼容性的考虑，编译器在对类型转换进行检查之后，如果没有发现违反规则的现象，就将所有的类型化数组列表转换成原始ArrayList对象。在程序运行时，所有的数组列表都是一样的，即没有虚拟机中的类型参数。因此，类型转换（ArrayList）和（ArrayList<Employee>）将执行相同的运行时检查。

在这种情形下，不必做什么。只要在与遗留的代码进行交叉操作时，研究一下编译器的警告性提示，并确保这些警告不会造成太严重的后果就行了。

## 5.4 对象包装器与自动打包

有时，需要将int这样的基本类型转换为对象。所有的基本类型都有一个与之对应的类。例如，Integer类对应基本类型int。通常，这些类称为包装器（wrapper）。这些对象包装器类拥有很鲜明的名字：Integer、Long、Float、Double、Short、Byte、Character、Void和Boolean（前6个类派生于公共的超类Number）。对象包装器类是不可变的，即一旦构造了包装器，就不允许更改包装在其中的值。同时，对象包装器类还是final，因此不能定义它们的子类。

假设定义一个整型数组列表。而尖括号中的类型参数不允许是基本类型，也就是说，不允许写成ArrayList<int>。这里就用到了Integer对象包装器类。我们可以声明一个Integer对象的数组列表。

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



警告：由于每个值分别包装在对象中，所以ArrayList<Integer>的效率远远低于int[]数组。因此，应该用它构造小型集合，其原因是此时程序员操作的方便性要比执行效率更加重要。


Java SE 5.0的另一个改进之处是更加便于添加或获得数组元素。下面这个调用

```
list.add(3);
```

将自动地变换成

```
list.add(new Integer(3));
```

这种变换被称为自动打包（autoboxing）。

 注释：大家可能认为自动打包（autowrapping）更加合适，而“装箱（boxing）”这个词源自于C#。

相反地，当将一个Integer对象赋给一个int值时，将会自动地拆包。也就是说，编译器将下列语句：

```
int n = list.get(i);
```

翻译成

```
int n = list.get(i).intValue();
```

甚至在算术表达式中也能够自动地打包和拆包。例如，可以将自增操作符应用于一个包装器引用：


```
Integer n = 3;  
n++;
```

编译器将自动地插入一条拆开对象包的指令，然后进行自增计算，最后再将结果打入对象包内。

在很多情况下，容易有一种假象，即基本类型与它们的对象包装器是一样的，只是它们的相等性不同。大家知道，==运算符也可以应用于对象包装器对象，只不过检测的是对象是否指向同一个存储区域，因此，下面的比较通常不会成立：

```
Integer a = 1000;  
Integer b = 1000;  
if (a == b) ...
```

然而，Java实现却有可能（may）让它成立。如果将经常出现的值包装到同一个对象中，这种比较就有可能成立。这种不确定的结果并不是我们所希望的。解决这个问题的办法是在两个包装器对象比较时调用equals方法。

 注释：自动打包规范要求boolean、byte、char 127，介于-128~127之间的short和int被包装到固定的对象中。例如，如果在前面的例子中将a和b初始化为100，对它们进行比较的结果一定成立。

最后强调一下，打包和拆包是编译器认可的，而不是虚拟机。编译器在生成类的字节码时，插入必要的方法调用。虚拟机只是执行这些字节码。


使用数值对象包装器还有另外一个好处。Java设计者发现，可以将某些基本方法放置在包装器中，例如，将一个数字字符串转换成数值。

要想将字符串转换成整型，可以使用下面这条语句：

```
int x = Integer.parseInt(s);
```

这里与Integer对象没有任何关系，parseInt是一个静态方法。但Integer类是放置这个方法的一个好地方。

API注释说明了Integer类中包含的一些重要方法。其他数值类也实现了相应的方法。

 警告：有些人认为包装器类可以用来实现修改数值参数的方法，然而这是错误的。在第4章中曾经讲到，由于Java方法都是值传递，所以不可能编写一个下面这样的能够增加整型参数值的Java方法。

```
public static void triple(int x) // won't work
{
    x = 3 * x; // modifies local variable
}
```

将int替换成Integer又会怎样呢？

```
public static void triple(Integer x) // won't work
{
    ...
}
```

问题是Integer对象是不可变的：包含在包装器中的内容不会改变。不能使用这些包装器类创建修改数值参数的方法。

如果想编写一个修改数值参数值的方法，就需要使用在org.omg.CORBA包中定义的持有者（holder）类型，包括IntHolder、BooleanHolder等等。每个持有者类型都包含一个公有（!）域值，通过它可以访问存储在其中的值。

```
public static void triple(IntHolder x)
{
    x.value = 3 * x.value;
}
```

#### API java.lang.Integer 1.0

- int intValue()  
以int的形式返回Integer对象的值（在Number类中覆盖了intValue方法）。
- static String toString(int i)  
以一个新String对象的形式返回给定数值i的十进制表示。
- static String toString(int i, int radix)  
返回数值i的基于给定radix参数进制的表示。
- static int parseInt(String s)
- static int parseInt(String s, int radix)  
返回字符串s表示的整型数值，给定字符串表示的是十进制的整数（第一种方法），或者是radix参数进制的整数（第二种方法）。
- static Integer valueOf(String s)
- static Integer valueOf(String s, int radix)  
返回用s表示的整型数值进行初始化后的一个新Integer对象，给定字符串表示的是十进制的整数（第一种方法），或者是radix参数进制的整数（第二种方法）。

#### API java.text.NumberFormat 1.1

- Number parse(String s)  
返回数字值，假设给定的String表示了一个数值。

## 5.5 参数数量可变的方法

在Java SE 5.0以前的版本中，每个Java方法都有固定数量的参数。然而，现在的版本提供

了可以用可变的参数数量调用的方法（有时称为“可变参”方法）。

前面已经看到过这样的方法：printf。例如，下面的方法调用：

```
System.out.printf("%d", n);
```

和

```
System.out.printf("%d %s", n, "widgets");
```

在上面两条语句中，尽管一个调用包含两个参数，另一个调用包含三个参数，但它们调用的都是同一个方法。printf方法是这样定义的：

```
public class PrintStream
{
    public PrintStream printf(String fmt, Object... args) { return format(fmt, args); }
}
```

这里的省略号...是Java代码的一部分，它表明这个方法可以接收任意数量的对象（除fmt参数之外）。

实际上，printf方法接收两个参数，一个是格式字符串，另一个是Object[]数组，其中保存着所有的参数（如果调用者提供的是整型数组或者其他基本类型的值，自动打包功能将把它们转换成对象）。现在将扫描fmt字符串，并将第i个格式说明符与args[i]的值匹配起来。

换句话说，对于printf的实现者来说，Object...参数类型与Object[]完全一样。

编译器需要对printf的每次调用进行转换，以便将参数绑定到数组上，并在必要的时候进行自动打包：

```
System.out.printf("%d %s", new Object[] { new Integer(n), "widgets" } );
```

用户自己也可以定义可变参数的方法，并将参数指定为任意类型，甚至是基本类型。下面是一个简单的示例：其功能为计算若干个数值的最大值。

```
public static double max(double... values)
{
    double largest = Double.MIN_VALUE;
    for (double v : values) if (v > largest) largest = v;
    return largest;
}
```

可以像下面这样调用这个方法：

```
double m = max(3.1, 40.4, -5);
```

编译器将new double[] {3.1, 40.4, -5}传递给max方法。



注释：允许将一个数组传递给可变参数方法的最后一个参数。例如：

```
System.out.printf("%d %s", new Object[] { new Integer(1), "widgets" } );
```

因此，可以将已经存在且最后一个参数是数组的方法重新定义为可变参数的方法，而不会破坏任何已经存在的代码。例如，MessageFormat.format在Java SE 5.0就采用了这种方式。甚至可以将main方法声明为下列形式：

```
public static void main(String... args)
```

## 5.6 枚举类

读者在第3章已经看到，如何在Java SE 5.0以后的版本中定义枚举类型。下面是一个典型的例子：

```
public enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE };
```

实际上，这个声明定义的类型是一个类，它刚好有4个实例，在此尽量不要构造新对象。

因此，在比较两个枚举类型的值时，永远不需要调用equals，而直接使用“==”就可以了。

如果需要的话，可以在枚举类型中添加一些构造器、方法和域。当然，构造器只是在构造枚举常量的时候被调用。下面是一个示例：

```
enum Size
{
    SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");

    private Size(String abbreviation) { this.abbreviation = abbreviation; }
    public String getAbbreviation() { return abbreviation; }

    private String abbreviation;
}
```

所有的枚举类型都是Enum类的子类。它们继承了这个类的许多方法。其中最有用的一个是toString，这个方法能够返回枚举常量名。例如，Size.SMALL.toString()将返回字符串“SMALL”。

toString的逆方法是静态方法valueOf。例如，语句：

```
Size s = (Size) Enum.valueOf(Size.class, "SMALL");
```

将s设置成Size.SMALL。

每个枚举类型都有一个静态的values方法，它将返回一个包含全部枚举值的数组。例如，如下调用

```
Size[] values = Size.values();
```

返回包含元素Size.SMALL,Size.MEDIUM,Size.LARGE和Size.EXTRA\_LARGE的数组。

ordinal方法返回enum声明中枚举常量的位置，位置从0开始计数。例如：Size.MEDIUM.ordinal()返回1。

例5-5程序演示了枚举类型的工作方式。



注释：如同Class类一样，鉴于简化的考虑，Enum类省略了一个类型参数。例如，实际上，应该将枚举类型Size扩展为Enum<Size>。类型参数在compareTo方法中使用（类型参数方法在第6章中介绍，类型参数在第12章中介绍）。

### 例5-5 EnumTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates enumerated types.
```

```
5.  * @version 1.0 2004-05-24
6.  * @author Cay Horstmann
7.  */
8.  public class EnumTest
9.  {
10.     public static void main(String[] args)
11.     {
12.         Scanner in = new Scanner(System.in);
13.         System.out.print("Enter a size: (SMALL, MEDIUM, LARGE, EXTRA_LARGE) ");
14.         String input = in.next().toUpperCase();
15.         Size size = Enum.valueOf(Size.class, input);
16.         System.out.println("size=" + size);
17.         System.out.println("abbreviation=" + size.getAbbreviation());
18.         if (size == Size.EXTRA_LARGE)
19.             System.out.println("Good job--you paid attention to the _.");
20.     }
21. }
22.
23. enum Size
24. {
25.     SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");
26.
27.     private Size(String abbreviation) { this.abbreviation = abbreviation; }
28.     public String getAbbreviation() { return abbreviation; }
29.
30.     private String abbreviation;
31. }
```

#### java.lang.Enum <E> 5.0

- static Enum `valueOf(Class enumClass, String name)`  
返回指定名字、给定类的枚举常量。
- String `toString()`  
返回枚举常量名。
- int `ordinal()`  
返回枚举常量在enum声明中的位置，位置从0开始计数。
- int `compareTo(E other)`  
如果枚举常量出现在other之前，则返回一个负值；如果this==other，则返回0；否则，返回正值。枚举常量的出现次序在enum声明中给出。

## 5.7 反射

反射库（reflection library）提供了一个非常丰富且精心设计的工具集，以便编写能够动态操纵Java代码的程序。这项功能被大量地应用于JavaBeans中，它是Java组件的体系结构（有关JavaBeans的详细内容在卷II中阐述）。使用反射，Java可以支持Visual Basic用户习惯使用的工具。特别是在设计或运行中添加新类时，能够快速地将应用开发工具动态地查询新添加类的能力。

能够分析类能力的程序被称为反射（reflective）。反射机制的功能极其强大。在下面的章节中可以看到，可以用反射机制：



- 在运行中分析类的能力。
- 在运行中查看对象，例如，编写一个toString方法供所有类使用。
- 实现数组的操作代码。
- 利用Method对象，这个对象很像C++中的函数指针。

反射是一种功能强大且复杂的机制。使用它的主要对象是工具构造者，而不是应用程序员。如果仅对设计应用程序感兴趣，而对构造工具不感兴趣，可以跳过本章的剩余部分，稍后再返回来学习。

### 5.7.1 Class类

在程序运行期间，Java运行时系统始终为所有的对象维护一个被称为运行时的类型标识。这个信息保存着每个对象所属的类足迹。虚拟机利用运行时信息选择相应的方法执行。

然而，可以通过专门的Java类访问这些信息。保存这些信息的类被称为Class，这个名字很容易让人混淆。Object类中的getClass()方法将会返回一个Class类型的实例。

```
Employee e;  
...  
Class cl = e.getClass();
```

如同用一个Employee对象表示一个特定的雇员属性一样，一个Class对象将表示一个特定类的属性。最常用的Class方法是getName。这个方法将返回类的名字。例如，下面这条语句：

```
System.out.println(e.getClass().getName() + " " + e.getName());
```

如果e是一个雇员，则会打印输出：

```
Employee Harry Hacker
```

如果e是经理，则会打印输出：

```
Manager Harry Hacker
```

如果类在一个包里，包的名字也作为类名的一部分：

```
Date d = new Date();  
Class cl = d.getClass();  
String name = cl.getName(); // name is set to "java.util.Date"
```

还可以调用静态方法forName获得类名对应的Class对象。

```
String className = "java.util.Date";  
Class cl = Class.forName(className);
```

如果类名保存在字符串中，并可在运行中改变，就可以使用这个方法。当然，这个方法只有在className是类名或接口名时才能够执行。否则，forName方法将抛出一个checked exception（已检查异常）。无论何时使用这个方法，都应该提供一个异常处理器（exception handler）。如何提供一个异常处理器，请参看本节后面的“捕获异常”部分。




提示：在启动时，包含main方法的类被加载。它会加载所有需要的类。这些被加载的类又要加载它们需要的类，以此类推。对于一个大型的应用程序来说，这将会消耗很多时间，用户会因此感到不耐烦。可以使用下面这个技巧给用户一种启动速度比较快的幻觉。不过，要确保包含main方法的类没有显式地引用其他的类。首先，显示一个启动画面；


然后，通过调用Class.forName手工地加载其他的类。

获得Class类对象的第三种方法非常简单。如果T是任意的Java类型，T.class将代表匹配该类对象。例如：

```
Class c1 = Date.class; // if you import java.util.*;
Class c2 = int.class;
Class c3 = Double[].class;
```

请注意，一个Class对象实际上表示的是一个类型，而这个类型未必一定是一种类。例如，int不是类，但int.class是一个Class类型的对象。

 注释：从Java SE 5.0开始，Class类被参数化。例如，Class<Employee>的类型是Employee.class。没有说明这个问题的原因是：它将已经抽象的概念更加复杂化了。在大多数实际问题中，可以忽略类型参数，而使用原始的Class类。有关这个问题更详细的论述请参看第13章。

 警告：鉴于历史原因，getName方法在应用于数组类型的时候会返回一个很奇怪的名字：

- Double[ ].class.getName( ) 返回 “[Ljava.lang.Double;”。
- int[ ].class.getName( ) 返回 “[I”。

虚拟机为每个类型管理一个Class对象。因此，可以利用 == 运算符实现两个类对象比较的操作。例如，

```
if (e.getClass() == Employee.class) . . .
```


还有一个很有用的方法newInstance()，可以用来快速地创建一个类的实例。例如，


```
e.getClass().newInstance();
```

创建了一个与e具有相同类类型的实例。newInstance方法调用默认的构造器（没有参数的构造器）初始化新创建的对象。如果这个类没有默认的构造器，就会抛出一个异常。

将forName与newInstance配合起来使用，可以根据存储在字符串中的类名创建一个对象。

```
String s = "java.util.Date";
Object m = Class.forName(s).newInstance();
```

 注释：如果需要以这种方式向希望按名称创建的类的构造器提供参数，就不要使用上面那条语句，而必须使用Constructor类中的newInstance方法。

 C++注释：newInstance方法对应C++中虚拟构造器的习惯用法。然而，C++中的虚拟构造器不是一种语言特性，需要由专门的库支持。Class类与C++中的type\_info类相似，getClass方法与C++中的typeid运算符等价。但Java中的Class比C++中的type\_info的功能强。C++中的type\_info只能以字符串的形式显示一个类型的名字，而不能创建那个类型的对象。

### 5.7.2 捕获异常

我们将在第11章中全面地讲述异常处理机制，但现在时常遇到一些方法需要抛出异常。

当程序运行过程中发生错误时，就会“抛出异常”。抛出异常比终止程序要灵活得多，这是因为可以提供一个“捕获”异常的处理器（handler）对异常情况进行处理。

如果没有提供处理器，程序就会终止，并在控制台上打印出一条信息，其中给出了异常的类型。可能在前面已经看到过一些异常报告，例如，偶然使用了null引用或者数组越界等。

异常有两种类型：未检查异常和已检查异常。对于已检查异常，编译器将会检查是否提供了处理器。然而，有很多常见的异常，例如，访问null引用，都属于未检查异常。编译器不会查看是否为这些错误提供了处理器。毕竟，应该精心地编写代码来避免这些错误的发生，而不要将精力花在编写异常处理器上。

并不是所有的错误都是可以避免的。如果竭尽全力还是发生了异常，编译器就要求提供一个处理器。Class.forName方法就是一个抛出已检查异常的例子。在第11章中，将会看到几种异常处理的策略。现在，只介绍一下如何实现最简单的处理器。

将可能抛出已检查异常的一个或多个方法调用代码放在try块中，然后在catch子句中提供处理器代码。

```
try
{
    statements that might throw exceptions
}
catch(Exception e)
{
    handler action
}
```

下面是一个示例：

```
try
{
    String name = . . . ; // get class name
    Class c1 = Class.forName(name); // might throw exception
    . . . // do something with c1
}
catch(Exception e)
{
    e.printStackTrace();
}
```

如果类名不存在，则将跳过try块中的剩余代码，程序直接进入catch子句（这里，利用Throwable类的printStackTrace方法打印出栈的轨迹。Throwable是Exception类的超类）。如果try块中没有抛出任何异常，那么会跳过catch子句的处理器代码。

对于已检查异常，只需要提供一个异常处理器。可以很容易地发现会抛出已检查异常的方法。如果调用了抛出已检查异常的方法，而又没有提供处理器，编译器就会给出错误报告。

#### java.lang.Class 1.0

- static Class forName(String className)  
返回描述类名为className的Class对象。
- Object newInstance()  
返回这个类的一个新实例。

**API** java.lang.reflect.Constructor 1.1

- Object newInstance(Object[] args)

构造一个这个构造器所属类的新实例。

参数：args           这是提供给构造器的参数。有关如何提供参数的详细情况请参看反射的论述。

**API** java.lang.Throwable 1.0

- void printStackTrace()

将Throwable对象和栈的轨迹输出到标准错误流。

### 5.7.3 利用反射分析类的能力

下面简要地介绍一下反射机制最重要的内容——检查类的结构。

在java.lang.reflect包中有三个类Field、Method和Constructor分别用于描述类的域、方法和构造器。这三个类都有一个叫做getName的方法，用来返回项目的名称。Field类有一个getType方法，用来返回描述域所属类型的Class对象。Method和Constructor类有能够报告参数类型的方法，Method类还有一个可以报告返回类型的方法。这三个类还有一个叫做getModifiers的方法，它将返回一个整型数值，用不同的位开关描述public和static这样的修饰符使用状况。另外，还可以利用java.lang.reflect包中的Modifier类的静态方法分析getModifiers返回的整型数值。例如，可以使用Modifier类中的isPublic、isPrivate或isFinal判断方法或构造器是否是public、private或final。我们需要做的全部工作就是调用Modifier类的相应方法，并对返回的整型数值进行分析，另外，还可以利用Modifier.toString方法将修饰法打印出来。

Class类中的getFields、getMethods和getConstructors方法将分别返回类提供的public域、方法和构造器数组，其中包括超类的公有成员。Class类的getDeclaredFields、getDeclaredMethods和getDeclaredConstructors方法将分别返回类中声明的全部域、方法和构造器，其中包括私有和保护成员，但不包括超类的成员。

例5-6显示了如何打印一个类的全部信息的方法。这个程序将提醒用户输入类名，然后输出类中所有的方法和构造器的签名，以及全部域名。假如用户输入

```
java.lang.Double
```

程序将会输出：

```
public class java.lang.Double extends java.lang.Number
{
    public java.lang.Double(java.lang.String);
    public java.lang.Double(double);

    public int hashCode();
    public int compareTo(java.lang.Object);
    public int compareTo(java.lang.Double);
    public boolean equals(java.lang.Object);
    public java.lang.String toString();
    public static java.lang.String toString(double);
    public static java.lang.Double valueOf(java.lang.String);
    public static boolean isNaN(double);
    public boolean isNaN();
}
```

```
public static boolean isInfinite(double);
public boolean isInfinite();
public byte byteValue();
public short shortValue();
public int intValue();
public long longValue();
public float floatValue();
public double doubleValue();
public static double parseDouble(java.lang.String);
public static native long doubleToLongBits(double);
public static native long doubleToRawLongBits(double);
public static native double longBitsToDouble(long);

public static final double POSITIVE_INFINITY;
public static final double NEGATIVE_INFINITY;
public static final double NaN;
public static final double MAX_VALUE;
public static final double MIN_VALUE;
public static final java.lang.Class TYPE;
private double value;
private static final long serialVersionUID;
}
```

值得注意的是：这个程序可以分析Java解释器能够加载的任何类，而不仅仅是编译程序时可以使用的类。在下一章中，还将使用这个程序查看Java编译器自动生成的内部类。

#### 例5-6 ReflectionTest.java

```
1. import java.util.*;
2. import java.lang.reflect.*;
3.
4. /**
5.  * This program uses reflection to print all features of a class.
6.  * @version 1.1 2004-02-21
7.  * @author Cay Horstmann
8.  */
9. public class ReflectionTest
10. {
11.     public static void main(String[] args)
12.     {
13.         // read class name from command line args or user input
14.         String name;
15.         if (args.length > 0) name = args[0];
16.         else
17.         {
18.             Scanner in = new Scanner(System.in);
19.             System.out.println("Enter class name (e.g. java.util.Date): ");
20.             name = in.next();
21.         }
22.
23.         try
24.         {
25.             // print class name and superclass name (if != Object)
26.             Class c1 = Class.forName(name);
27.             Class supercl = c1.getSuperclass();
28.             String modifiers = Modifier.toString(c1.getModifiers());
29.             if (modifiers.length() > 0) System.out.print(modifiers + " ");
```

```

30.         System.out.print("class " + name);
31.         if (supercl != null && supercl != Object.class) System.out.print(" extends "
32.             + supercl.getName());
33.
34.         System.out.print("\n{\n");
35.         printConstructors(cl);
36.         System.out.println();
37.         printMethods(cl);
38.         System.out.println();
39.         printFields(cl);
40.         System.out.println("}");
41.     }
42.     catch (ClassNotFoundException e)
43.     {
44.         e.printStackTrace();
45.     }
46.     System.exit(0);
47. }
48.
49. /**
50.  * Prints all constructors of a class
51.  * @param cl a class
52.  */
53. public static void printConstructors(Class cl)
54. {
55.     Constructor[] constructors = cl.getDeclaredConstructors();
56.
57.     for (Constructor c : constructors)
58.     {
59.         String name = c.getName();
60.         System.out.print(" ");
61.         String modifiers = Modifier.toString(c.getModifiers());
62.         if (modifiers.length() > 0) System.out.print(modifiers + " ");
63.         System.out.print(name + "(");
64.
65.         // print parameter types
66.         Class[] paramTypes = c.getParameterTypes();
67.         for (int j = 0; j < paramTypes.length; j++)
68.         {
69.             if (j > 0) System.out.print(", ");
70.             System.out.print(paramTypes[j].getName());
71.         }
72.         System.out.println(");");
73.     }
74. }
75.
76. /**
77.  * Prints all methods of a class
78.  * @param cl a class
79.  */
80. public static void printMethods(Class cl)
81. {
82.     Method[] methods = cl.getDeclaredMethods();
83.
84.     for (Method m : methods)
85.     {
86.         Class retType = m.getReturnType();

```

```
87.         String name = m.getName();
88.
89.         System.out.print(" ");
90.         // print modifiers, return type, and method name
91.         String modifiers = Modifier.toString(m.getModifiers());
92.         if (modifiers.length() > 0) System.out.print(modifiers + " ");
93.         System.out.print(retType.getName() + " " + name + "(");
94.
95.         // print parameter types
96.         Class[] paramTypes = m.getParameterTypes();
97.         for (int j = 0; j < paramTypes.length; j++)
98.         {
99.             if (j > 0) System.out.print(", ");
100.            System.out.print(paramTypes[j].getName());
101.        }
102.        System.out.println(");");
103.    }
104. }
105.
106. /**
107.  * Prints all fields of a class
108.  * @param cl a class
109.  */
110. public static void printFields(Class cl)
111. {
112.     Field[] fields = cl.getDeclaredFields();
113.
114.     for (Field f : fields)
115.     {
116.         Class type = f.getType();
117.         String name = f.getName();
118.         System.out.print(" ");
119.         String modifiers = Modifier.toString(f.getModifiers());
120.         if (modifiers.length() > 0) System.out.print(modifiers + " ");
121.         System.out.println(type.getName() + " " + name + ";");
122.     }
123. }
124. }
```

**API** java.lang.Class 1.0

- Field[] getFields() 1.1
- Field[] getDeclaredFields() 1.1

getFields方法将返回一个包含Field对象的数组，这些对象记录了这个类或其超类的公有域。getDeclaredField方法也将返回包含Field对象的数组，这些对象记录了这个类的全部域。如果类中没有域，或者Class对象描述的是基本类型或数组类型，这些方法将返回一个长度为0的数组。

- Method[] getMethods() 1.1
- Method[] getDeclaredMethods() 1.1

返回包含Method对象的数组：getMethods将返回所有的公有方法，包括从超类继承来的公有方法；getDeclaredMethods返回这个类或接口的全部方法，但不包括由超类继承了的

方法。

- `Constructor[] getConstructors()` 1.1
- `Constructor[] getDeclaredConstructors()` 1.1  
返回包含`Constructor`对象的数组，其中包含了`Class`对象所描述的类的所有公有构造器（`getConstructors`）或所有构造器（`getDeclaredConstructors`）。

**API** `java.lang.reflect.Field` 1.1

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

**API** `java.lang.reflect.Constructor` 1.1

- `Class getDeclaringClass()`  
返回一个用于描述类中定义的构造器、方法或域的`Class`对象。
- `Class[] getExceptionTypes()`（在`Constructor`和`Method`类中）  
返回一个用于描述方法抛出的异常类型的`Class`对象数组。
- `int getModifiers()`  
返回一个用于描述构造器、方法或域的修饰符的整型数值。使用`Modifier`类中的这个方法可以分析这个返回值。
- `String getName()`  
返回一个用于描述构造器、方法或域名的字符串。
- `Class[] getParameterTypes()`（在`Constructor`和`Method`类中）  
返回一个用于描述参数类型的`Class`对象数组。
- `Class getReturnType()`（在`Method`类中）  
返回一个用于描述返回类型的`Class`对象。

**API** `java.lang.reflect.Modifier` 1.1

- `static String toString(int modifiers)`  
返回对应`modifiers`位设置的修饰符的字符串表示。
- `static boolean isAbstract(int modifiers)`
- `static boolean isFinal(int modifiers)`
- `static boolean isInterface(int modifiers)`
- `static boolean isNative(int modifiers)`
- `static boolean isPrivate(int modifiers)`
- `static boolean isProtected(int modifiers)`
- `static boolean isPublic(int modifiers)`
- `static boolean isStatic(int modifiers)`
- `static boolean isStrict(int modifiers)`
- `static boolean isSynchronized(int modifiers)`



- `static boolean isVolatile(int modifiers)`

这些方法将检测方法名中对应的修饰符在`modifiers`值中的位。

#### 5.7.4 在运行时使用反射分析对象

从前面一节中，已经知道如何查看任意对象的数据域名称和类型：

- 获得对应的Class对象。
- 通过Class对象调用`getDeclaredFields`。

本节，将进一步查看数据域的实际内容。当然，在编写程序时，如果知道想要查看的域名和类型，查看指定的域是一件很容易的事情。而利用反射机制可以查看在编译时还不清楚的对象域。

查看对象域的关键方法是Field类中的`get`方法。如果`f`是一个Field类型的对象（例如，通过`getDeclaredFields`得到的对象），`obj`是某个包含`f`域的类的对象，`f.get(obj)`将返回一个对象，其值为`obj`域的当前值。这样说起来显得有点抽象，这里看一看下面这个示例的运行。

```
Employee harry = new Employee("Harry Hacker", 35000, 10, 1, 1989);
Class c1 = harry.getClass();
// the class object representing Employee
Field f = c1.getDeclaredField("name");
// the name field of the Employee class
Object v = f.get(harry);
// the value of the name field of the harry object
// i.e., the String object "Harry Hacker"
```

实际上，这段代码存在一个问题。由于`name`是一个私有域，所以`get`方法将会抛出一个`IllegalAccessException`。只有利用`get`方法才能得到可访问域的值。除非拥有访问权限，否则Java安全机制只允许查看任意对象有哪些域，而不允许读取它们的值。

反射机制的默认行为受限于Java的访问控制。然而，如果一个Java程序没有受到安全管理器的控制，就可以覆盖访问控制。为了达到这个目的，需要调用Field、Method或Constructor对象的`setAccessible`方法。例如，

```
f.setAccessible(true); // now OK to call f.get(harry);
```

`setAccessible`方法是`AccessibleObject`类中的一个方法，它是Field、Method和Constructor类的公共超类。这个特性是为调试、持久存储和相似机制提供的。本书稍后将利用它编写一个通用的`toString`方法。

`get`方法还有一个需要解决的问题。`name`域是一个String，因此把它作为Object返回没有什么问题。但是，假定我们想要查看`salary`域。它属于double类型，而Java中数值类型不是对象。要想解决这个问题，可以使用Field类中的`getDouble`方法，也可以调用`get`方法，此时，反射机制将会自动地将这个域值打包到相应的对象包装器中，这里将打包成Double。

当然，可以获得就可以设置。调用`f.set(obj, value)`可以将`obj`对象的`f`域设置成新值。

例5-7显示了如何编写一个可供任意类使用的通用`toString`方法。其中使用`getDeclaredFields`获得所有的数据域，然后使用`setAccessible`将所有的域设置为可访问的。对于每个域，获得了名字和值。例5-7递归调用`toString`方法，将每个值转换成字符串。

```

class ObjectAnalyzer
{
    public String toString(Object obj)
    {
        Class cl = obj.getClass();
        . . .
        String r = cl.getName();
        // inspect the fields of this class and all superclasses
        do
        {
            r += "[";
            Field[] fields = cl.getDeclaredFields();
            AccessibleObject.setAccessible(fields, true);
            // get the names and values of all fields
            for (Field f : fields)
            {
                if (!Modifier.isStatic(f.getModifiers()))
                {
                    if (!r.endsWith("(")) r += ", "
                    r += f.getName() + "=";
                    try
                    {
                        Object val = f.get(obj);
                        r += toString(val);
                    }
                    catch (Exception e) { e.printStackTrace(); }
                }
            }
            r += "];";
            cl = cl.getSuperclass();
        }
        while (cl != null);
        return r;
    }
    . . .
}

```

在例5-7的全部代码中，需要解释几个复杂的问题。循环引用将有可能导致无限递归。因此，ObjectAnalyzer将保存已经被访问过的对象。另外，为了能够查看数组内部，需要采用一种不同的方式。有关这种方式的具体内容将在下一节中详细地论述。

可以使用toString方法查看任意对象的内部信息。例如，下面这个调用

```

ArrayList<Integer> squares = new ArrayList<Integer>();
for (int i = 1; i <= 5; i++) squares.add(i * i);
System.out.println(new ObjectAnalyzer().toString(squares));

```

将会产生下面的打印结果：

```

java.util.ArrayList[elementData=class java.lang.Object[]{java.lang.Integer[value=1][[]],
java.lang.Integer[value=4][[]], java.lang.Integer[value=9][[]], java.lang.Integer[value=16][[]],
java.lang.Integer[value=25][[]], null, null, null, null, null}, size=5][modCount=5][[]]

```

还可以使用通用的toString方法实现自己类中的toString方法，如下所示：

```

public String toString()
{
    return new ObjectAnalyzer().toString(this);
}

```

这是一种公认的提供toString方法的手段，在编写程序时会发现，它是非常有用的。

#### 例5-7 ObjectAnalyzerTest.java

```
1. import java.lang.reflect.*;
2. import java.util.*;
3.
4. /**
5.  * This program uses reflection to spy on objects.
6.  * @version 1.11 2004-02-21
7.  * @author Cay Horstmann
8.  */
9. public class ObjectAnalyzerTest
10. {
11.     public static void main(String[] args)
12.     {
13.         ArrayList<Integer> squares = new ArrayList<Integer>();
14.         for (int i = 1; i <= 5; i++)
15.             squares.add(i * i);
16.         System.out.println(new ObjectAnalyzer().toString(squares));
17.     }
18. }
19.
20. class ObjectAnalyzer
21. {
22.     /**
23.      * Converts an object to a string representation that lists all fields.
24.      * @param obj an object
25.      * @return a string with the object's class name and all field names and
26.      * values
27.      */
28.     public String toString(Object obj)
29.     {
30.         if (obj == null) return "null";
31.         if (visited.contains(obj)) return "...";
32.         visited.add(obj);
33.         Class cl = obj.getClass();
34.         if (cl == String.class) return (String) obj;
35.         if (cl.isArray())
36.         {
37.             String r = cl.getComponentType() + "[]{";
38.             for (int i = 0; i < Array.getLength(obj); i++)
39.             {
40.                 if (i > 0) r += ",";
41.                 Object val = Array.get(obj, i);
42.                 if (cl.getComponentType().isPrimitive()) r += val;
43.                 else r += toString(val);
44.             }
45.             return r + "}";
46.         }
47.
48.         String r = cl.getName();
49.         // inspect the fields of this class and all superclasses
50.         do
51.         {
52.             r += "[";
53.             Field[] fields = cl.getDeclaredFields();
```

```
54.     AccessibleObject.setAccessible(fields, true);
55.     // get the names and values of all fields
56.     for (Field f : fields)
57.     {
58.         if (!Modifier.isStatic(f.getModifiers()))
59.         {
60.             if (!r.endsWith("[") r += ",";
61.             r += f.getName() + "=";
62.             try
63.             {
64.                 Class t = f.getType();
65.                 Object val = f.get(obj);
66.                 if (t.isPrimitive()) r += val;
67.                 else r += toString(val);
68.             }
69.             catch (Exception e)
70.             {
71.                 e.printStackTrace();
72.             }
73.         }
74.         r += "1";
75.         cl = cl.getSuperclass();
76.     }
77.     while (cl != null);
78.
79.     return r;
80. }
81.
82.
83. private ArrayList<Object> visited = new ArrayList<Object>();
84. }
```

#### **API** java.lang.reflect.AccessibleObject 1.2

- void setAccessible(boolean flag)  
为反射对象设置可访问标志。flag为true表明屏蔽Java语言的访问检查，使得对象的私有属性也可以被查询和设置。
- boolean isAccessible()  
返回反射对象的可访问标志的值。
- static void setAccessible(AccessibleObject[] array, boolean flag)  
是一种设置对象数组可访问标志的快捷方法。

#### **API** java.lang.Class 1.1

- Field getField(String name)
- Field[] getFields()  
返回指定名称的公有域，或包含所有域的数组。
- Field getDeclaredField(String name)
- Field[] getDeclaredFields()

返回类中声明的给定名称的域，或者包含声明的全部域的数组。

#### API java.lang.reflect.Field 1.1

- Object get(Object obj)  
返回obj对象中用Field对象表示的域值。
- void set(Object obj, Object newValue)  
用newValue设置Obj对象中用Field对象表示的域。

### 5.7.5 使用反射编写泛型数组代码

java.lang.reflect包中的Array类允许动态地创建数组。例如，将这个特性应用到第3章中讲到的arrayCopy方法时，可以在保留当前数组内容的同时动态地扩展现有数组。

这里要解决一个十分具有代表性的问题。假设有一个元素为某种类型且已经被填满数组。现在希望扩展它的长度，但不想手工地编写那些扩展、拷贝元素的代码，而是想编写一个用于扩展数组的通用方法：

```
Employee[] a = new Employee[100];  
...  
// array is full  
a = (Employee[]) arrayGrow(a);
```

如何编写这样一个通用的方法呢？正好能够将Employee[]数组转换为Object[]数组，这让人感觉很有希望。下面试着编写一个通用的方法，其功能是将数组扩展到10%+10个元素（这是因为对于小型数组来说，扩展10%显得有点少）。

```
static Object[] badArrayGrow(Object[] a) // not useful  
{  
    int newLength = a.length * 11 / 10 + 10;  
    Object[] newArray = new Object[newLength];  
    System.arraycopy(a, 0, newArray, 0, a.length);  
    return newArray;  
}
```

然而，在实际使用结果数组时会遇到一个问题。这段代码返回的数组类型是对象数组（Object[]）类型，这是由于使用下面这行代码创建的数组：

```
new Object[newLength]
```

一个对象数组不能转换成雇员数组（Employee[]）。如果这样做，则在运行时Java将会产生ClassCastException异常。前面已经看到，Java数组会记住每个元素的类型，即创建数组时new表达式中使用的元素类型。将一个Employee[]临时地转换成Object[]数组，然后再把它转换回来是可以的，但一个从开始就是Object[]的数组却永远不能转换成Employee[]数组。为了编写这类通用的数组代码，需要能够创建与原数组类型相同的新数组。为此，需要java.lang.reflect包中Array类的一些方法。其中最关键的是Array类中的静态方法newInstance，它能够构造新数组。在调用它时必须提供两个参数，一个是数组的元素类型，一个是数组的长度。

```
Object newArray = Array.newInstance(componentType, newLength);
```

为了能够实际地运行，需要获得新数组的长度和元素类型。

可以通过调用`Array.getLength(a)`获得数组的长度，也可以通过`Array`类的静态`getLength`方法的返回值得到任意数组的长度。而要获得新数组元素类型，就需要进行以下工作：

1) 首先获得`a`数组的类对象。

2) 确认它是一个数组。

3) 使用`Class`类（只能定义表示数组的类对象）的`getComponentType`方法确定数组对应的类型。

为什么`getLength`是`Array`的方法，而`getComponentType`是`Class`的方法呢？我们也不清楚。反射方法的分类有时确实显得有点古怪。下面是这段代码：

```
static Object goodArrayGrow(Object a) // useful
{
    Class cl = a.getClass();
    if (!cl.isArray()) return null;
    Class componentType = cl.getComponentType();
    int length = Array.getLength(a);
    int newLength = length * 11 / 10 + 10;
    Object newArray = Array.newInstance(componentType, newLength);
    System.arraycopy(a, 0, newArray, 0, length);
    return newArray;
}
```

请注意，`arrayGrow`方法可以用来扩展任意类型的数组，而不仅是对象数组。

```
int[] a = { 1, 2, 3, 4 };
a = (int[]) goodArrayGrow(a);
```

为了能够实现上述操作，应该将`goodArrayGrow`的参数声明为`Object`类型，而不要声明为对象型数组（`Object[]`）。整型数组类型`int[]`可以被转换成`Object`，但不能转换成对象数组。

例5-8显示了两个扩展数组的方法。请注意，将`badArrayGrow`的返回值进行类型转换将会抛出一个异常。



注释：这段程序显示了通过反射，数组的工作过程。如果只希望扩大数组，利用`Arrays`类的`copyOf`方法就可以。

```
Employee[] a = new Employee[100];
...
// array is full
a = Arrays.copyOf(a, a.length * 11 / 10 + 10);
```

#### 例5-8 ArrayGrowTest.java

```
1. import java.lang.reflect.*;
2.
3. /**
4.  * This program demonstrates the use of reflection for manipulating arrays.
5.  * @version 1.01 2004-02-21
6.  * @author Cay Horstmann
7.  */
8. public class ArrayGrowTest
9. {
10.     public static void main(String[] args)
11.     {
```

```
12.     int[] a = { 1, 2, 3 };
13.     a = (int[]) goodArrayGrow(a);
14.     arrayPrint(a);
15.
16.     String[] b = { "Tom", "Dick", "Harry" };
17.     b = (String[]) goodArrayGrow(b);
18.     arrayPrint(b);
19.
20.     System.out.println("The following call will generate an exception.");
21.     b = (String[]) badArrayGrow(b);
22. }
23.
24. /**
25.  * This method attempts to grow an array by allocating a new array and copying all elements.
26.  * @param a the array to grow
27.  * @return a larger array that contains all elements of a. However, the returned array has
28.  * type Object[], not the same type as a
29.  */
30. static Object[] badArrayGrow(Object[] a)
31. {
32.     int newLength = a.length * 11 / 10 + 10;
33.     Object[] newArray = new Object[newLength];
34.     System.arraycopy(a, 0, newArray, 0, a.length);
35.     return newArray;
36. }
37.
38. /**
39.  * This method grows an array by allocating a new array of the same type and
40.  * copying all elements.
41.  * @param a the array to grow. This can be an object array or a primitive
42.  * type array
43.  * @return a larger array that contains all elements of a.
44.  */
45. static Object goodArrayGrow(Object a)
46. {
47.     Class cl = a.getClass();
48.     if (!cl.isArray()) return null;
49.     Class componentType = cl.getComponentType();
50.     int length = Array.getLength(a);
51.     int newLength = length * 11 / 10 + 10;
52.
53.     Object newArray = Array.newInstance(componentType, newLength);
54.     System.arraycopy(a, 0, newArray, 0, length);
55.     return newArray;
56. }
57.
58. /**
59.  * A convenience method to print all elements in an array
60.  * @param a the array to print. It can be an object array or a primitive type array
61.  */
62. static void arrayPrint(Object a)
63. {
64.     Class cl = a.getClass();
65.     if (!cl.isArray()) return;
66.     Class componentType = cl.getComponentType();
67.     int length = Array.getLength(a);
```


```
68.     System.out.print(componentType.getName() + "[" + length + "] = { ");
69.     for (int i = 0; i < Array.getLength(a); i++)
70.         System.out.print(Array.get(a, i) + " ");
71.     System.out.println("}");
72. }
73. }
```

#### java.lang.reflect.Array 1.1

- static Object get(Object array, int index)
- static Xxx getXxx(Object array, int index)  
(xxx是boolean、byte、char、double、float、int、long、short之中的一种基本类型。) 这些方法将返回存储在给定位置上的给定数组的内容。
- static void set(Object array, int index, Object newValue)
- static setXxx(Object array, int index, Xxx newValue)  
(xxx是boolean、byte、char、double、float、int、long、short之中的一种基本类型。) 这些方法将一个新值存储到给定位置上的给定数组中。
- static int getLength(Object array)  
返回数组的长度。
- static Object newInstance(Class componentType, int length)
- static Object newInstance(Class componentType, int[] lengths)  
返回一个具有给定类型、给定维数的新数组。

#### 5.7.6 方法指针

从表面上看，Java没有提供方法指针，即将一个方法的存储地址传给另外一个方法，以便第二个方法能够随后调用它。事实上，Java的设计者曾说过：方法指针是很危险的，并且常常会带来隐患。他们认为Java提供的接口（interface）（将在下一章讨论）是一种更好的解决方案。然而，在Java 1.1中方法指针已经作为反射包的（也许是）副产品出现了。

 注释：微软公司为自己的非标准Java语言J++（以及后来的C#）增加了另一种被称为委托（delegate）的方法指针类型，它与本节讨论的Method类不同。然而，在下一章中讨论的内部类比委托更加有用。

为了能够看到方法指针的工作过程，先回忆一下利用Field类的get方法查看对象域的过程。与之类似，在Method类中有一个invoke方法，它允许调用包装在当前Method对象中的方法。invoke方法的签名是：

```
Object invoke(Object obj, Object... args)
```

第一个参数是隐式参数，其余的对象提供了显式参数（在Java SE 5.0以前的版本中，必须传递一个对象数组，如果没有显式参数就传递一个null）。

对于静态方法，第一个参数可以被忽略，即可以将它设置为null。

例如，假设用ml代表Employee类的getName方法，下面这条语句显示了如何调用这个方法：



```
String n = (String) m1.invoke(harry);
```

如果参数或返回类型不是类而是基本类型，那么在调用Field类的get和set方法时会存在一些问题。需要依靠自动打包功能将其打包，或者在Java SE 5.0之前将基本类型打包成对应的包装器。

相反地，如果返回类型是一种基本类型，则invoke方法将返回包装器类型。例如，假设m2代表Employee类的getSalary方法，返回的实际类型是Double，因此必须相应地进行类型转换。对于Java SE 5.0来说，这项操作可以由自动拆包来完成。

```
double s = (Double) m2.invoke(harry);
```

如何得到Method对象呢？当然，可以通过调用getDeclaredMethods方法，然后对返回的Method对象数组进行查找，直到发现想要的方法为止。也可以通过调用Class类中的getMethod方法得到想要的方法。它与getField方法类似。getField方法根据表示域名的字符串，返回一个Field对象。然而，有可能存在若干个相同名字的方法，因此要格外小心，以确保能够准确地得到想要的那个方法。有鉴于此，还必须提供想要的方法的参数类型。getMethod的签名是：

```
Method getMethod(String name, Class... parameterTypes)
```

例如，下面说明了如何获得Employee类的getName方法和raiseSalary方法的方法指针。

```
Method m1 = Employee.class.getMethod("getName");  
Method m2 = Employee.class.getMethod("raiseSalary", double.class);
```

（对于Java SE 5.0以前的版本，必须将Class对象包装到一个数组中，如果没有参数，需要给出一个null）。

到此为止，读者已经学习了使用Method对象的规则。下面看一下如何将它们组织在一起。例5-9是一个打印诸如Math.sqrt、Math.sin这样的数学函数值表的程序。打印的结果如下所示：

```
public static native double java.lang.Math.sqrt(double)  
1.0000 | 1.0000  
2.0000 | 1.4142  
3.0000 | 1.7321  
4.0000 | 2.0000  
5.0000 | 2.2361  
6.0000 | 2.4495  
7.0000 | 2.6458  
8.0000 | 2.8284  
9.0000 | 3.0000  
10.0000 | 3.1623
```

当然，这段打印数学函数表格的代码与具体打印的数学函数无关。

```
double dx = (to - from) / (n - 1);  
for (double x = from; x <= to; x += dx)  
{  
    double y = (Double) f.invoke(null, x);  
    System.out.printf("%10.4f | %10.4f\n", x, y);  
}
```

在这里，f是一个Method类型的对象。由于正在调用的方法是一个静态方法，所以invoke的第一个参数是null。

为了将Math.sqrt函数表格化，需要将f设置为：

```
Math.class.getMethod("sqrt", double.class)
```

这是Math类中的一个方法，通过参数向它提供了一个函数名sqrt和一个double类型的参数。

例5-9给出了通用制表和两个测试程序的全部代码。

#### 例5-9 MethodPointerTest.java

```
1. import java.lang.reflect.*;
2.
3. /**
4.  * This program shows how to invoke methods through reflection.
5.  * @version 1.1 2004-02-21
6.  * @author Cay Horstmann
7.  */
8. public class MethodPointerTest
9. {
10.     public static void main(String[] args) throws Exception
11.     {
12.         // get method pointers to the square and sqrt methods
13.         Method square = MethodPointerTest.class.getMethod("square", double.class);
14.         Method sqrt = Math.class.getMethod("sqrt", double.class);
15.
16.         // print tables of x- and y-values
17.
18.         printTable(1, 10, 10, square);
19.         printTable(1, 10, 10, sqrt);
20.     }
21.
22.     /**
23.      * Returns the square of a number
24.      * @param x a number
25.      * @return x squared
26.      */
27.     public static double square(double x)
28.     {
29.         return x * x;
30.     }
31.
32.     /**
33.      * Prints a table with x- and y-values for a method
34.      * @param from the lower bound for the x-values
35.      * @param to the upper bound for the x-values
36.      * @param n the number of rows in the table
37.      * @param f a method with a double parameter and double return value
38.      */
39.     public static void printTable(double from, double to, int n, Method f)
40.     {
41.         // print out the method as table header
42.         System.out.println(f);
43.
44.         double dx = (to - from) / (n - 1);
45.
46.         for (double x = from; x <= to; x += dx)
47.         {
48.             try
49.             {
50.                 double y = (Double) f.invoke(null, x);
51.                 System.out.printf("%10.4f | %10.4f%n", x, y);
```

```
52.     }
53.     catch (Exception e)
54.     {
55.         e.printStackTrace();
56.     }
57. }
58. }
59. }
```

上述程序清楚地表明，可以使用method对象实现C（或C#中的委派）语言中函数指针的所有操作。同C一样，这种程序设计风格并不太简便，出错的可能性也比较大。如果在调用方法的时候提供了一个错误的参数，那么invoke方法将会抛出一个异常。

另外，invoke的参数和返回值必须是Object类型的。这就意味着必须进行多次的类型转换。这样做将会使编译器错过检查代码的机会。因此，等到测试阶段才会发现这些错误，找到并改正它们将会更加困难。不仅如此，使用反射获得方法指针的代码要比仅仅直接调用方法明显慢一些。

有鉴于此，建议仅在必要的时候才使用Method对象，而最好使用接口和内部类（第6章中介绍）。特别要重申：建议Java开发者不要使用Method对象的回调功能。使用接口进行回调（第6章介绍）会使得代码的执行速度更快，更易于维护。



java.lang.reflect.Method 1.1

- public Object invoke(Object implicitParameter, Object[] explicitParameters)  
调用这个对象所描述的方法，传递给定参数，并返回方法的返回值。对于静态方法，把null作为隐式参数传递。在使用包装器传递基本类型的值时，基本类型的返回值必须是未包装的。

## 5.8 继承设计的技巧

下面给出一些对设计继承关系很有帮助的建议，以此结束本章的内容。

1) 将公共操作和域放在超类。

这就是为什么将姓名域放在person类中，而没有将它放在Employee和Student类中的原因。

2) 不要使用受保护的域。

有些程序员认为，将大多数的实例域定义为protected是一个不错的主意，只有这样，子类才能够在需要的时候直接访问它们。然而，protected机制并不能够带来更好的保护，其原因主要有两点。第一，子类集合是无限制的，任何一个人都能够由某个类派生一个子类，并编写代码以直接访问protected的实例域，从而破坏了封装性。第二，在Java程序设计语言中，在同一个包中的所有类都可以访问protected域，而不管它是否为这个类的子类。

3) 使用继承实现“is-a”关系。

使用继承很容易达到节省代码的目的，但有时候也被人们滥用了。例如，假设需要定义一个钟点工类。钟点工的信息包含姓名和雇佣日期，但是没有薪水。他们按小时计薪，并且不会因为拖延时间而获得加薪。这似乎在诱导人们由Employee派生出子类Contractor，然后再增加

一个hourlyWage域。

```
class Contractor extends Employee
{
    ...
    private double hourlyWage;
}
```

这并不是一个好主意。因为这样一来，每个钟点工对象中都包含了薪水和计时工资这两个域。在实现打印支票或税单方法的时候，会带来无尽的麻烦，并且与不采用继承，会多写很多代码。

钟点工与雇员之间不属于“is-a”关系。钟点工不是特殊的雇员。

4) 除非所有继承的方法都有意义，否则不要使用继承。

假设想编写一个Holiday类。毫无疑问，每个假日也是一日，并且一日可以用GregorianCalendar类的实例表示，因此可以使用继承。

```
class Holiday extends GregorianCalendar { ... }
```

很遗憾，在继承的操作中，假日集不是封闭的。在GregorianCalendar中有一个公有方法add，可以将假日转换成非假日：

```
Holiday christmas;
christmas.add(Calendar.DAY_OF_MONTH, 12);
```

因此，继承对于这个例子来说并不太适宜。

5) 在覆盖方法时，不要改变预期的行为。

置换原则不仅应用于语法，而且也可以应用于行为，这似乎更加重要。在覆盖一个方法的时候，不应该毫无原由地改变行为的内涵。就这一点而言，编译器不会提供任何帮助，即编译器不会检查重新定义的方法是否有意义。例如，可以重定义Holiday类中add方法“修正”原方法的问题，或什么也不做，或抛出一个异常，或继续到下一个假日。然而这些都违反了置换原则。语句序列

```
int d1 = x.get(Calendar.DAY_OF_MONTH);
x.add(Calendar.DAY_OF_MONTH, 1);
int d2 = x.get(Calendar.DAY_OF_MONTH);
System.out.println(d2 - d1);
```

不管x属于GregorianCalendar类，还是属于Holiday类，执行上述语句后都应该得到预期的行为。

当然，这样可能会引起某些争议。人们可能就预期行为的含义争论不休。例如，有些人争论说，置换原则要求Manager.equals不处理bonus域，因为Employee.equals没有它。实际上，凭空讨论这些问题毫无意义。关键在于，在覆盖子类中的方法时，不要偏离最初的设计想法。

6) 使用多态，而非类型信息。

无论什么时候，对于下面这种形式的代码

```
if (x is of type 1)
    action1(x);
else if (x is of type 2)
    action2(x);
```

都应该考虑使用多态性。

action1与action2表示的是相同的概念吗？如果是相同的概念，就应该为这个概念定义一个方法，并将其放置在两个类的超类或接口中，然后，就可以调用

```
x.action();
```

以便使用多态性提供的动态分派机制执行相应的动作。

使用多态方法或接口编写的代码比使用对多种类型进行检测的代码更加易于维护和扩展。

7) 不要过多地使用反射。

反射机制使得人们可以通过在运行时查看域和方法，让人们编写出更具有通用性的程序。这种功能对于编写系统程序来说极其实用，但是通常不适于编写应用程序。反射是很脆弱的，即编译器很难帮助人们发现程序中的错误。任何错误只能在运行时才被发现，并导致异常。