

第4章 对象与类

面向对象程序设计概述

使用现有类

用户自定义类

静态域和方法

方法参数

对象构造

包

类路径

文档注释

类设计技巧

这一章将主要介绍如下内容：

- 面向对象程序设计
- 如何创建标准Java类库中的类对象
- 如何编写自己的类

如果你没有面向对象程序设计的应用背景，就一定要认真地阅读一下本章的内容。面向对象程序设计与面向过程程序设计在思维方式上存在着很大的差别。改变一种思维方式并不是一件很容易的事情，而且为了继续学习Java也要清楚对象的概念。

对于具有使用C++经历的程序员来说，与上一章相同，对本章的内容不会感到太陌生，但在两种语言之间还是存在着很多不同之处，所以要认真地阅读本章的后半部分内容，你将发现“C++注释”对学习Java语言会很有帮助的。

4.1 面向对象程序设计概述

面向对象程序设计（简称OOP）是当今主流的程序设计范型，它已经取代了70年代的“结构化”过程化程序设计开发技术。Java是完全面向对象的，必须熟悉OOP才能够编写Java程序。

面向对象的程序是由对象组成的，每个对象包含对用户公开的特定功能部分和隐藏的实现部分。程序中的很多对象来自于标准库，还有一些是自定义的。究竟是自己构造对象，还是从外界购买对象完全取决于预算和时间。但是，从根本上说，只要对象能够满足要求，就不必关心其功能的具体实现过程。在OOP中，不必关心对象的具体实现，只要能够满足用户的需求即可。

传统的结构化程序设计通过设计一系列的过程（即算法）来求解问题。这些过程一旦被确定，就要开始考虑存储数据的方式。这就是Pascal语言的设计者Niklaus Wirth将其编著的有关程序设计的著名书籍命名为《算法 + 数据结构 = 程序》（*Algorithms + Data Structures = Programs*, Prentice Hall, 1975）的原因。需要注意的是，在Wirth命名的标题中，算法是第一位的，数据结构是第二位的。这就明确地表述了程序员的工作方式。首先要确定如何操作数据，然后再决定如何组织数据，以便于数据操作。OOP却调换了这个次序，数据被放在第一位，然后再考虑操作数据的算法。

对于一些规模较小的问题，将其分解为过程的开发方式比较理想。而面向对象更加适用于解决规模较大的问题。要想实现一个简单的web浏览器可能需要大约2000个过程，这些过程可能需要对一组全局数据进行操作。采用面向对象的设计风格，可能只需要大约100个类，每个类平均包含20个方法（如图4-1所示）。后者更易于程序员掌握，也容易找到bug。假设给定对象的数据处于一种错误状态，在访问过这个数据项的20个方法中查找错误要比在2000个过程中查找容易得多。

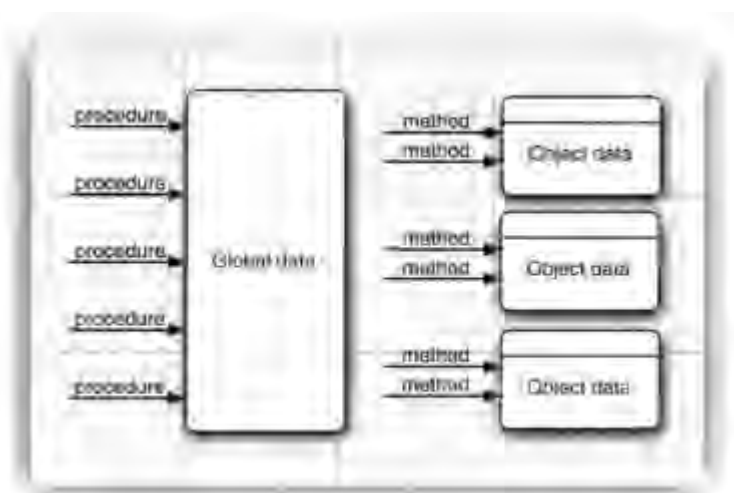


图4-1 面向过程与面向对象的程序设计对比

4.1.1 类

类（class）是构造对象的模板或蓝图。我们可以将类想像成小甜饼的切割机，将对象想像为小甜饼。由类构造（construct）对象的过程称为创建类的实例（instance）。

正如前面所看到的，用Java编写的所有代码都位于某个类的内部。标准的Java库提供了几千个类，可以用于用户界面设计、日期、日历和网络程序设计。尽管如此，还是需要在Java程序中创建一些自己的类，以便描述应用程序所对应的问题域中的对象。

封装（encapsulation，有时称为数据隐藏）是与对象有关的一个重要概念。从形式上看，封装不过是将数据和行为组合在一个包中，并对对象的使用者隐藏了数据的实现方式。对象中的数据称为实例域（instance fields），操纵数据的过程称为方法（method）。对于每个特定的类实例（对象）都有一组特定的实例域值。这些值的集合就是这个对象的当前状态（state）。无论何时，只要向对象发送一个消息，它的状态就有可能发生改变。

实现封装的关键在于绝对不能让类中的方法直接地访问其他类的实例域。程序仅通过对象的方法与对象数据进行交互。封装给予对象了“黑盒”特征，这是提高重用性和可靠性的关键。这意味着一个类可以全面地改变存储数据的方式，只要仍旧使用同样的方法操作数据，其他对象就不会知道或介意所发生的变化。

OOP的另一个原则会让用户自定义Java类变得轻而易举，这就是：类可以通过扩展另一个类来建立。事实上，在Java中，所有的类都源自于一个“神通广大的超类”，它就是Object。在下一章中，读者将可以看到有关Object类的详细介绍。

在对一个已有的类扩展时，这个扩展后的新类具有所扩展的类的全部属性和方法。在新类中，只需要提供那些仅适用于这个类的新方法和数据域就可以了。通过扩展一个类来建立另外一个类的过程被称为继承（inheritance），有关继承的详细内容请参看下一章。

4.1.2 对象

要想使用OOP，一定要清楚对象的三个主要特性：

- 对象的行为（behavior）——可以对对象施加哪些操作，或可以对对象施加哪些方法？
- 对象的状态（state）——当施加那些方法时，对象如何响应？
- 对象标识（identity）——如何辨别具有相同行为与状态的不同对象？

同一个类的所有对象实例，由于支持相同的行为而具有家族相似性。对象的行为是用可调用的方法定义的。

此外，每个对象都保存着描述当前特征的信息。这就是对象的状态。对象的状态可能会随着时间而发生改变，但这种改变不会是自发的。对象状态的改变必须通过调用方法实现（如果不经方法调用就可以改变对象状态，只能说明封装性遭到了破坏）。

但是，对象的状态并不能完全描述一个对象。每个对象都有一个惟一的身份（identity）。例如，在一个订单处理系统中，任何两个订单都存在着不同之处，即使所订购的货物完全相同也是如此。需要注意，作为一个类的实例，每个对象的标识永远是不同的，状态常常也存在着差异。

对象的这些关键特性在彼此之间相互影响着。例如，对象的状态影响它的行为（如果一个订单“已送货”或“已付款”，就应该拒绝调用具有增删订单中条目的方法。反过来，如果订单是“空的”，即还没有加入预订的物品，这个订单就不应该进入“已送货”状态）。

4.1.3 识别类

传统的过程化程序设计，必须从顶部的main函数开始编写程序。在设计面向对象的系统时没有所谓的“顶部”。对于学习OOP的初学者来说常常会感觉无从下手。答案是：首先从设计类开始，然后再往每个类中添加方法。

识别类的简单规则是在分析问题的过程中寻找名词，而方法对应着动词。

例如，在订单处理系统中，有这样一些名词：

- 项目（Item）
- 订单（Order）
- 送货地址（Shipping address）
- 付款（Payment）
- 账户（Account）

这些名词很可能成为类Item、Order等等。

接下来，查看动词。物品项目被添加到订单中。订单被发送或取消。订单货款被支付。对于每一个动词如：“添加”、“发送”、“取消”以及“支付”，都要标识出主要负责完成相应动作的对象。例如，当一个新的条目添加到订单中时，那个订单对象就是被指定的对象，因为它知道如何存储条目以及如何对条目进行排序。也就是说，add应该是Order类的一个方法，而Item

对象是一个参数。

当然，所谓“名词与动词”原则只是一种粗略的方法，在创建类的时候，哪些名词和动词是重要的完全取决于个人的开发经验。

4.1.4 类之间的关系

在类之间，最常见的关系有

- 依赖 (“uses-a”)
- 聚合 (“has-a”)
- 继承 (“is-a”)

依赖 (dependence)，即 “uses-a” 关系，是一种最明显的、最常见的关系。例如，Order类使用Account类是因为Order对象需要访问Account对象查看信用状态。但是Item类不依赖于Account类，这是因为Item对象与客户账户无关。因此，如果一个类的方法操纵另一个类的对象，我们就说一个类依赖于另一个类。

应该尽可能地将相互依赖的类减至最少。如果类A不知道B的存在，它就不会关心B的任何改变（这意味着B的改变不会导致A产生任何bug）。用软件工程的术语来说，就是让类之间的耦合度最小。

聚合 (aggregation)，即 “has-a” 关系，是一种具体且易于理解的关系。例如，一个Order对象包含一些Item对象。聚合关系意味着类A的对象包含类B的对象。



注释：有些方法学家不喜欢聚合这个概念，而更加喜欢使用“关联”。从建模的角度看，这是可以理解的。但对于程序员来说，“has-a”显得更加形象。喜欢使用聚合的另一个理由是关联的标准符号不易区分。请参看表4-1。

表4-1 表达类关系的UML符号

关 系	UML连接符
继承	
接口继承	
依赖	
聚合	
关联	
直接关联	

继承 (inheritance)，即 “is-a” 关系，是一种用于表示特殊与一般关系的。例如，Rush Order类由Order类继承而来。在具有特殊性的RushOrder类中包含了一些用于优先处理的特殊方法，以及一个计算运费的不同方法；而其他的方法，如添加条目、生成账单等等都是从Order类继承来的。一般而言，如果类A扩展类B，类A不但包含从类B继承的方法，还会拥有一些额外的功能（下一章将详细讨论继承，其中会用较多的篇幅讲述这个重要的概念）。

很多程序员采用UML (Unified Modeling Language) 绘制描述类之间关系的类图。图4-2就

是这样一个例子。类用矩形表示，类之间的关系用带有各种修饰的箭头表示。表4-1给出了UML中最常见的箭头样式。



图4-2 类的示意图

✓ 注释：有很多专门用来绘制UML类图的工具。有些开发商提供了一些高性能（当然也是昂贵的）的工具旨在辅助开发过程。其中，有Rational Rose（<http://www.ibm.com/software/awdtools/developer/Rose>）和Together（<http://www.borland.com/us/products/together>）。还有一种选择是使用开放源代码程序ArgoUML（<http://argouml.tigris.org>）。在GentleWare（<http://gentleware.com>）提供了一个商业支持版本。如果只想使用UML最粗浅的内容绘制一个简单的UML类图，就可以试试Violet（<http://violet.sourceforge.net>）。

4.2 使用现有类

在Java中，没有类就无法做任何事情，我们前面曾经接触过几个类。然而，并不是所有的类都具有面向对象特征。例如，Math类。在程序中，可以使用Math类的方法，如Math.random，并只需要知道方法名和参数（如果有的话），而不必了解它的具体实现过程。这正是封装的关键所在，当然所有类都是这样。但遗憾的是，Math类只封装了功能，它不需要也不必隐藏数据。由于没有数据，因此也不必担心生成对象以及初始化实例域。

下一节将会给出一个更加具有代表性的类——Date类。从中可以看到如何构造对象，以及如何调用类的方法。

4.2.1 对象与对象变量

要想使用对象，就必须首先构造对象，并指定其初始状态。然后，对对象施加方法。

在Java程序设计语言中，使用构造器（constructor）构造新实例。构造器是一种特殊的方法，用来构造并初始化对象。下面看一个例子。在标准Java库中包含一个Date类。它的对象将描述一个时间点，例如：“December 31, 1999, 23:59:59 GMT”。

☑ 注释：可能会感到奇怪：为什么用类描述时间，而不像其他语言那样用一个内置的（built-in）类型？例如，在Visual Basic中有一个内置的date类型，程序员可以采用#6/1/1995#格式指定日期。从表面上看，这似乎很方便，因为程序员使用内置的date类型，而不必为设计类而操心。但实际上，Visual Basic这样设计的适应性如何呢？在有些地区日期表示为月/日/年，而另一些地区表示为日/月/年。语言设计者是否能够预见这些问题呢？如果没有处理好这类问题，语言就有可能陷入混乱，对此感到不满的程序员也会丧失使用这种语言的热情。如果使用类，这些设计任务就交给了类库的设计者。如果类设计的不完善，其他的操作员可以很容易地编写自己的类，以便增强或替代（replace）系统提供的类（作为这个问题的印证：Java的日期类库有些混乱，对它的重新设计正在进行中。请参看<http://jcp.org/en/jsr/detail?id=310>）。

构造器的名字应该与类名相同。因此Date类的构造器名为Date。要想构造一个Date对象，需要在构造器前面加上new操作符，如下所示：

```
new Date()
```

这个表达式构造了一个新对象。这个对象被初始化为当前的日期和时间。

如果需要的话，也可以将这个对象传递给一个方法：

```
System.out.println(new Date());
```

相反，也可以将一个方法应用于刚刚创建的对象上。Date类中有一个toString方法。这个方法将返回日期的字符串描述。下面的语句可以说明如何将toString方法应用于新构造的Date对象上。

```
String s = new Date().toString();
```

在这两个例子中，构造的对象仅使用了一次。通常，希望构造的对象可以多次使用，因此，需要将对象存放在一个变量中：

```
Date birthday = new Date();
```

图4-3显示了引用新构造的对象变量birthday。

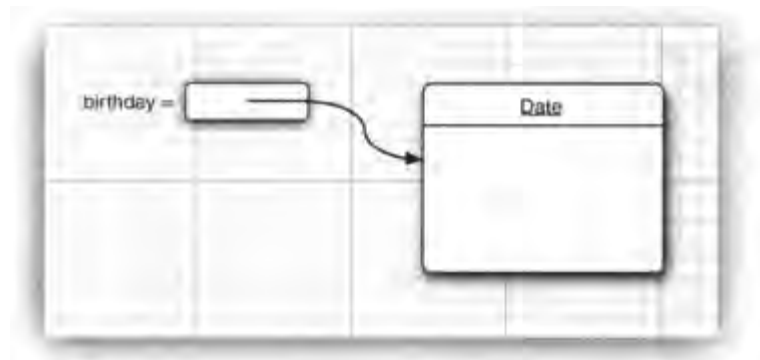


图4-3 创建一个新对象

在对象与对象变量之间存在着一个重要的区别。例如，语句

```
Date deadline; // deadline doesn't refer to any object
```

定义了一个对象变量deadline，它可以引用Date类型的对象。但是，一定要认识到：变量deadline

不是一个对象，实际上也没有引用对象。此时，不能将任何 Date 方法应用于这个变量上。语句

```
s = deadline.toString(); // not yet
```

将产生编译错误。

必须首先初始化变量deadline。这里有两个选择。当然，可以用新构造的对象初始化这个变量：

```
deadline = new Date();
```

也让这个变量引用一个已存在的对象：

```
deadline = birthday;
```

现在，这两个变量引用同一个对象（请参见图4-4）。

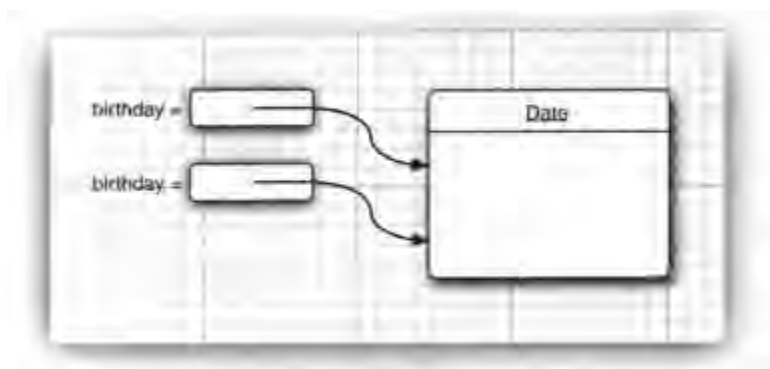


图4-4 引用同一个对象的对象变量

一定要认识到：一个对象变量并没有实际包含一个对象，而仅仅引用一个对象。

在Java中，任何对象变量的值都是对存储在另外一个地方的一个对象的引用。new操作符的返回值也是一个引用。下列语句：

```
Date deadline = new Date();
```

有两个部分。表达式new Date()构造了一个Date类型的对象，并且它的值是对新创建对象的引用。这个引用存储在变量deadline中。

可以显式地将对象变量设置为null，表明这个对象变量目前没有引用任何对象。

```
deadline = null;  
...  
if (deadline != null)  
    System.out.println(deadline);
```

如果将一个方法应用于一个值为null的对象上，那么就会产生运行错误。

```
birthday = null;  
String s = birthday.toString(); // runtime error!
```

变量不会自动地初始化为null，而必须通过调用new 或将它们设置为null进行初始化。



C++注释：很多人错误地认为Java对象变量与C++的引用类似。然而，在C++ 中没有空引用，并且引用不能被赋值。可以将Java的对象变量看作C++的对象指针。例如，

```
Date birthday; // Java
```

实际上，等同于

```
Date* birthday; // C++
```

一旦理解了这一点，一切问题就迎刃而解了。当然，一个Date*指针只能通过调用new进行初始化。就这一点而言，C++与Java的语法几乎是一样的。

```
Date* birthday = new Date(); // C++
```

如果把一个变量的值赋给另一个变量，两个变量就指向同一个日期，即指向同一个对象。在Java中的null引用对应C++中的NULL指针。

所有的Java对象都存储在堆中。当一个对象包含另一个对象变量时，这个变量依然包含着指向另一个堆对象的指针。

在C++中，指针十分令人头疼，并常常导致程序错误。稍不小心就会创建一个错误的指针，或者造成内存溢出。在Java语言中，这些问题都不复存在。如果使用一个没有初始化的指针，运行系统将会产生一个运行时错误，而不是生成一个随机的结果。同时，不必担心内存管理问题，垃圾收集器将会处理相关的事宜。

C++确实做了很大的努力，它通过拷贝型构造器和复制操作符来实现对象的自动拷贝。例如，一个链表（linked list）拷贝的结果将会得到一个新链表，其内容与原始链表相同，但却是一组独立的链接。这使得将同样的拷贝行为内置在类中成为可能。在Java中，必须使用clone方法获得对象的完整拷贝。

4.2.2 Java类库中的GregorianCalendar类

在前面的例子中，已经使用了Java标准类库中的Date类。Date类的实例有一个状态，即特定的时间点。

尽管在使用Date类时不必知道这一点，但时间是用距离一个固定时间点的毫秒数（可正可负）表示的，这个点就是所谓的纪元（epoch），它是UTC时间1970年1月1日 00:00:00。UTC是Coordinated Universal Time的缩写，与大家熟悉的GMT（即Greenwich Mean Time/格林威治时间）一样，是一种具有实际目的的科学标准时间。

但是，Date类所提供的日期处理并没有太大的用途。Java类库的设计者认为：像“December 31, 1999, 23:59:59”这样的日期表示法只是阳历的固有习惯。这种特定的描述法遵循了世界上大多数地区使用的Gregorian阳历表示法。但是，同一时间点采用中国的农历表示和采用希伯来的阴历表示就太不一样，对于火星历来说就更不可想像了。



注释：有史以来，人类的文明与历法的设计紧紧地相连，日历给日期命名、给太阳和月亮的周期排列次序。有关世界上各种日历的有趣解释，从法国革命的日历到玛雅人计算日期的方法等等，请参看Nachum Dershowitz 和 Edward M. Reingold 编写的《Calendrical Calculations》（剑桥大学出版社，第2版，2001年）。

类库设计者决定将保存时间与给时间点命名分开。所以标准Java类库分别包含了两个类：一个是用来表示时间点的Date类；另一个是用来表示大家熟悉的日历表示法的GregorianCalendar类。事实上，GregorianCalendar类扩展了一个更加通用的Calendar类，这个类描述了日历的一般属性。理论上，可以通过扩展Calendar类来实现中国的阴历或者是火星日历。然而，标准类库中只实现了Gregorian日历。

将时间与日历分开是一种很好的面向对象设计。通常，最好使用不同的类表示不同的概念。

Date类只提供了少量的方法用来比较两个时间点。例如 before 和 after 方法分别表示一个时间点是否早于另一个时间点，或者晚于另一个时间点。

```
if (today.before(birthday))
    System.out.println("Still time to shop for a gift.");
```



注释：实际上，Date类还有getDay、getMonth以及getYear等方法，然而并不推荐使用这些方法。当类库设计者意识到某个方法不应该存在时，就把它标记为不鼓励使用。

当类库的设计者意识到单独设计日历类更有实际意义时，这些方法已经是Date类的一部分。引入日历类之后，Date类中的这些方法被标明为不鼓励使用，虽然在程序中仍然可以使用它们，但是如果这样做，编译时会出现警告。最好还是不要使用这部分方法，它们有可能会从未来的类库版本中删去。

GregorianCalendar类所包含的方法要比Date类多得多。特别是有几个很有用的构造器。表达式

```
new GregorianCalendar()
```

构造一个新对象，用于表示对象构造时的日期和时间。

另外，还可以通过提供年、月、日构造一个表示某个特定日期午夜的日历对象：

```
new GregorianCalendar(1999, 11, 31)
```

有些怪异的是，月份从0开始计数。因此11表示十二月。为了清晰起见，也可以使用常量，如：Calendar.DECEMBER。

```
new GregorianCalendar(1999, Calendar.DECEMBER, 31)
```

还可以设置时间：

```
new GregorianCalendar(1999, Calendar.DECEMBER, 31, 23, 59, 59)
```

当然，常常希望将构造的对象存储在对象变量中：

```
GregorianCalendar deadline = new GregorianCalendar(. . .);
```

GregorianCalendar类封装了实例域，这些实例域保存着设置的日期信息。不查看源代码不可能知道日期在类中的具体表达方式。当然，这一点并不重要，重要的是要知道类向外界开放的方法。

4.2.3 更改器方法与访问器方法

现在，可能有人会问：如何从封装在某个GregorianCalendar对象内部的日期中获得当前的日、月、年呢？如果希望对这些内容做一些修改，又该怎么做呢？在联机文档中，以及本章末尾的API注释中可以找到答案。这里只讲述一些最重要的方法。

日历的作用是提供某个时间点的年、月、日等信息。要想查询这些设置信息，应该使用GregorianCalendar类的get方法。为了表达希望得到的项，需要借助于Calendar类中定义的一些常量，如：Calendar.MONTH或Calendar.DAY_OF_WEEK：

```
GregorianCalendar now = new GregorianCalendar();
int month = now.get(Calendar.MONTH);
```

```
int weekday = now.get(Calendar.DAY_OF_WEEK);
API注释列出了可以使用全部常量。
调用set方法，可以改变对象的状态：
deadline.set(Calendar.YEAR, 2001);
deadline.set(Calendar.MONTH, Calendar.APRIL);
deadline.set(Calendar.DAY_OF_MONTH, 15);
```

另外，还有一个便于设置年、月、日的方法，调用方式如下：

```
deadline.set(2001, Calendar.APRIL, 15);
```

最后，还可以为给定的日期对象增加天数、星期数、月份等等：

```
deadline.add(Calendar.MONTH, 3); // move deadline by 3 months
```

如果传递的数值是一个负数，日期就向后移。

get方法与set和add方法在概念上是有区别的。get方法仅仅查看并返回对象的状态，而set和add方法却对对象的状态进行修改。对实例域做出修改的方法被称为更改器方法（mutator method），仅访问实例域而不进行修改的方法被称为访问器方法（accessor method）。



C++注释：在C++中，带有const后缀的方法是访问器方法；默认为更改器方法。但是，在Java语言中，访问器方法与更改器方法在语法上没有明显的区别。

通常的习惯是在访问器方法名前面加上前缀get，在更改器方法前面加上前缀set。例如，在GregorianCalendar类有getTime方法和setTime方法，它们分别用来获得和设置日历对象所表示的时间点。

```
Date time = calendar.getTime();
calendar.setTime(time);
```

这些方法在进行GregorianCalendar和Date类之间的转换时非常有用。这里有一个例子。假定已知年、月、日，并且希望创建一个包含这个时间值的Date对象。由于Date类不知道如何操作日历，所以首先需要构造一个GregorianCalendar对象，然后再调用getTime方法获得一个日期对象：

```
GregorianCalendar calendar = new GregorianCalendar(year, month, day);
Date hireDay = calendar.getTime();
```

与之相反，如果希望获得Date对象中的年、月、日信息，就需要构造一个GregorianCalendar对象、设置时间，然后再调用get方法：

```
GregorianCalendar calendar = new GregorianCalendar();
calendar.setTime(hireDay);
int year = calendar.get(Calendar.YEAR);
```

下面用一个应用GregorianCalendar类的程序来结束本节内容的论述。这个程序将显示当前月的日历，其格式为：

```
Sun Mon Tue Wed Thu Fri Sat
                        1
 2   3   4   5   6   7   8
 9  10  11  12  13  14  15
16  17  18  19* 20  21  22
23  24  25  26  27  28  29
30  31
```

当前的日用一个*号标记。可以看到，这个程序需要解决如何计算某月份的天数以及一个给定日期相应是星期几。

下面看一下这个程序的关键步骤。首先，构造了一个日历对象，并用当前的日期和时间进行初始化。

```
GregorianCalendar d = new GregorianCalendar();
```

经过两次调用get方法获得当时的日、月。

```
int today = d.get(Calendar.DAY_OF_MONTH);
int month = d.get(Calendar.MONTH);
```

然后，将d设置为这个月的第一天，并得到这一天为星期几。

```
d.set(Calendar.DAY_OF_MONTH, 1);
int weekday = d.get(Calendar.DAY_OF_WEEK);
```

如果这个月的第一天是星期日，变量weekday就设置为Calendar.SUNDAY；如果这个月的第一天是星期一，就设置为Calendar.MONDAY，以此类推（实际上，这些值是整数值1，2，...，7，但最好不要依赖背景知识书写代码）。

注意，在日历的第一行是缩进的，因此，月份的第一天指向相应的星期几。由于每个星期的第一天存在着不同的约定习惯，因此需要能够随机应变。在美国，每个星期的第一天是星期日；在欧洲，每个星期的第一天是星期一，最后一天是星期日。

Java虚拟机非常注意当前用户的所在地区，不同的地区存在着不同的格式习惯，包括每星期的起始日以及一星期中每天的命名方式。



提示：如果想看到不同地区程序的输出，应该在main方法的第一行中添加下列代码：

```
Locale.setDefault(Locale.ITALY);
```

getFirstDayOfWeek方法获得当前地区星期的起始日。为了确定所需要的缩进距离，将日历对象的日减1，直到一个星期的第一天为止。

```
int firstDayOfWeek = d.getFirstDayOfWeek();
int indent = 0;
while (weekday != firstDayOfWeek)
{
    indent++;
    d.add(Calendar.DAY_OF_MONTH, -1);
    weekday = d.get(Calendar.DAY_OF_WEEK);
}
```

随后，输出表示星期几名称的头。这个操作可以通过调用DateFormatSymbols类方法实现。

```
String [] weekdayNames = new DateFormatSymbols().getShortWeekdays();
```

getShortWeekdays方法返回用户语种所命名的表示星期几的缩写字符串（例如：英语将返回“Sun”、“Mon”等）。数组用星期数作为索引。下面的循环将打印标题：

```
do
{
    System.out.printf("%4s", weekdayNames[weekday]);
    d.add(Calendar.DAY_OF_MONTH, 1);
    weekday = d.get(Calendar.DAY_OF_WEEK);
}
```

```
while (weekday != firstDayOfWeek);
System.out.println();
```

现在，已经做好打印日历内容的准备了。第一行缩进，并将日期对象设置为月份的起始日。在循环中用d记录一个月中的每一天。

在每次迭代过程中，打印日期值。如果d是当前日期，打印日期之后再打印一个*标记。每个星期的第一天，重新换行打印。而后，将d设置为下一天：

```
d.add(Calendar.DAY_OF_MONTH, 1);
```

什么时候结束呢？我们并不知道这个月有31天、30天、29天，还是28天。实际上，只要d还指示当月就应该继续迭代。

```
do
{
    ...
}
while (d.get(Calendar.MONTH) == month);
```

一旦d进入下一个月，程序就终止执行。

例4-1给出了完整的程序。

正如前面所看到的，日历程序包含了一些复杂问题，例如：某一天是星期几，每个月有多少天等等。有了 `GregorianCalendar` 类一切就变得简单了。我们并不必知道 `GregorianCalendar` 类是如何计算星期数和每个月的天数，而只需要使用类提供的接口：`get`方法、`set`方法以及 `add`方法就可以了。

这个示例程序关键告诉我们：可以使用类的接口解决复杂任务，而不必知道其中的实现细节。

例4-1 CalendarTest.java

```
1. import java.text.DateFormatSymbols;
2. import java.util.*;
3.
4. /**
5.  * @version 1.4 2007-04-07
6.  * @author Cay Horstmann
7.  */
8.
9. public class CalendarTest
10. {
11.     public static void main(String[] args)
12.     {
13.         // construct d as current date
14.         GregorianCalendar d = new GregorianCalendar();
15.
16.         int today = d.get(Calendar.DAY_OF_MONTH);
17.         int month = d.get(Calendar.MONTH);
18.
19.         // set d to start date of the month
20.         d.set(Calendar.DAY_OF_MONTH, 1);
21.
22.         int weekday = d.get(Calendar.DAY_OF_WEEK);
23.
24.         // get first day of week (Sunday in the U.S.)
25.         int firstDayOfWeek = d.getFirstDayOfWeek();
```

```
26.
27.     // determine the required indentation for the first line
28.     int indent = 0;
29.     while (weekday != firstDayOfWeek)
30.     {
31.         indent++;
32.         d.add(Calendar.DAY_OF_MONTH, -1);
33.         weekday = d.get(Calendar.DAY_OF_WEEK);
34.     }
35.
36.     // print weekday names
37.     String[] weekdayNames = new DateFormatSymbols().getShortWeekdays();
38.     do
39.     {
40.         System.out.printf("%4s", weekdayNames[weekday]);
41.         d.add(Calendar.DAY_OF_MONTH, 1);
42.         weekday = d.get(Calendar.DAY_OF_WEEK);
43.     }
44.     while (weekday != firstDayOfWeek);
45.     System.out.println();
46.
47.     for (int i = 1; i <= indent; i++)
48.         System.out.print(" ");
49.
50.     d.set(Calendar.DAY_OF_MONTH, 1);
51.     do
52.     {
53.         // print day
54.         int day = d.get(Calendar.DAY_OF_MONTH);
55.         System.out.printf("%3d", day);
56.
57.         // mark current day with *
58.         if (day == today) System.out.print("*");
59.         else System.out.print(" ");
60.
61.         // advance d to the next day
62.         d.add(Calendar.DAY_OF_MONTH, 1);
63.         weekday = d.get(Calendar.DAY_OF_WEEK);
64.
65.         // start a new line at the start of the week
66.         if (weekday == firstDayOfWeek) System.out.println();
67.     }
68.     while (d.get(Calendar.MONTH) == month);
69.     // the loop exits when d is day 1 of the next month
70.
71.     // print final end of line if necessary
72.     if (weekday != firstDayOfWeek) System.out.println();
73. }
74. }
```

java.util.GregorianCalendar 1.1

- `GregorianCalendar()`
构造一个日历对象，用来表示默认地区、默认时区的当前时间。
- `GregorianCalendar(int year, int month, int day)`

- `GregorianCalendar(int year, int month, int day, int hour, int minutes, int seconds)`

用给定的日期和时间构造一个Gregorian 日历对象。

参数：year 该日期所在的年份
 month 该日期所在的月份。此值以0为基准；例如，0表示一月
 day 该月份中的日期
 hour 小时（0到23之间）
 minutes 分钟（0到59之间）
 seconds 秒（0到59之间）

- `int get(int field)`

返回给定域的值。

参数：field 可以是下述选项之一：`Calendar.ERA`、`Calendar.YEAR`、`Calendar.MONTH`、`Calendar.WEEK_OF_YEAR`、`Calendar.WEEK_OF_MONTH`、`Calendar.DAY_OF_MONTH`、`Calendar.DAY_OF_YEAR`、`Calendar.DAY_OF_WEEK`、`Calendar.DAY_OF_WEEK_IN_MONTH`、`Calendar.AM_PM`、`Calendar.HOUR`、`Calendar.HOUR_OF_DAY`、`Calendar.MINUTE`、`Calendar.SECOND`、`Calendar.MILLISECOND`、`Calendar.ZONE_OFFSET`、`Calendar.DST_OFFSET`

- `void set(int field, int value)`

设置特定域的值。

参数：field get接收的常量之一
 value 新值

- `void set(int year, int month, int day)`

- `void set(int year, int month, int day, int hour, int minutes, int seconds)`

将日期域和时间域设置为新值。

参数：year 该日期所在的年份
 month 该日期所在的月份。此值以0为基准；例如，0表示一月
 day 该月份中的日期
 hour 小时（0到23）
 minutes 分钟（0到59）
 seconds 秒（0到59）

- `void add(int field, int amount)`

这是一个可以对日期信息实施算术运算的方法。对给定的时间域增加指定数量的时间。例如，可以通过调用`c.add(Calendar.DAY_OF_MONTH, 7)`，将当前的日历日期加上7。

参数：field 需要修改的域（可以使用get方法文档中给出的一个常量）
 amount 域被改变的数量（可以是负值）

- `int getFirstDayOfWeek()`
获得当前用户所在地区，一个星期中的第一天。例如：在美国一个星期中的第一天是 `Calendar.SUNDAY`。
- `void setTime(Date time)`
将日历设置为指定的时间点。
参数：`time` 时间点
- `Date getTime()`
获得这个日历对象当前值所表达的时间点。

API `java.text.DateFormatSymbols 1.1`

- `String[] getShortWeekdays()`
 - `String[] getShortMonths()`
 - `String[] getWeekdays()`
 - `String[] getMonths()`
- 获得当前地区的星期几或月份的名称。利用 `Calendar` 的星期和月份常量作为数组索引值。

4.3 用户自定义类

在第3章中，已经开始编写了一些简单的类。但是，那些类都只包含一个简单的 `main` 方法。现在开始学习如何设计复杂应用程序所需要的各种主力类（workhorse class）。通常，这些类没有 `main` 方法，而却有自定义的实例域和实例方法。要想创建一个完整的程序，应该将若干类组合在一起，其中只有一个类有 `main` 方法。

4.3.1 一个 `Employee` 类

在Java中，最简单的类定义形式为：

```
class ClassName
{
    constructor1
    constructor2
    . . .
    method1
    method2
    . . .
    field1
    field2
    . . .
}
```



注释：这里编写类所采用的风格是类的方法在前面，域在后面。这种风格有易于促使人们更加关注接口的概念，削减对实现的注意。

下面看一个非常简单的 `Employee` 类。在编写薪金管理系统时可能会用到。

```
class Employee
{
```

```
// constructor
public Employee(String n, double s, int year, int month, int day)
{
    name = n;
    salary = s;
    GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
    hireDay = calendar.getTime();
}

// a method
public String getName()
{
    return name;
}

// more methods
...

// instance fields
private String name;
private double salary;
private Date hireDay;
}
```

这里将这个类的实现细节分成以下几个部分，并分别在稍后的几节中给予介绍。下面先看看例4-2，它显示了一个使用Employee类的程序代码。

在这个程序中，构造了一个Employee数组，并填入了三个雇员对象：

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

staff[0] = new Employee("Carl Cracker", ...);
staff[1] = new Employee("Harry Hacker", ...);
staff[2] = new Employee("Tony Tester", ...);
```

接下来，利用Employee类的raiseSalary方法将每个雇员的薪水提高5%：

```
for (Employee e : staff)
    e.raiseSalary(5);
```

最后，调用getName方法、getSalary方法和getHireDay方法将每个雇员的信息打印出来：

```
for (Employee e : staff)
    System.out.println("name=" + e.getName()
        + ", salary=" + e.getSalary()
        + ", hireDay=" + e.getHireDay());
```

注意，在这个示例程序中包含两个类：一个Employee类；一个带有public访问修饰符的EmployeeTest类。EmployeeTest类包含了main方法，其中使用了前面介绍的指令。

源文件名是EmployeeTest.java，这是因为文件名必须与public类的名字相匹配。在一个源文件中，只能有一个公有类，但可以有任意数目的非公有类。

接下来，当编译这段源代码的时候，编译器将在目录下创建两个类文件：EmployeeTest.class和Employee.class。

将程序中包含main方法的类名字提供给字节码解释器，以便启动这个程序：

```
java EmployeeTest
```


字节码解释器开始运行EmployeeTest类的主方法中的代码。在这段代码中，先后构造了三个新Employee对象，并显示它们的状态。

例4-2 EmployeeTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program tests the Employee class.
5.  * @version 1.11 2004-02-19
6.  * @author Cay Horstmann
7.  */
8. public class EmployeeTest
9. {
10.     public static void main(String[] args)
11.     {
12.         // fill the staff array with three Employee objects
13.         Employee[] staff = new Employee[3];
14.
15.         staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
16.         staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
17.         staff[2] = 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.         // GregorianCalendar uses 0 for January
38.         hireDay = calendar.getTime();
39.     }
40.
41.     public String getName()
42.     {
43.         return name;
44.     }
45.
46.     public double getSalary()
47.     {
48.         return salary;
49.     }
50.
51.     public Date getHireDay()
```

```
52. {  
53.     return hireDay;  
54. }  
55.  
56. public void raiseSalary(double byPercent)  
57. {  
58.     double raise = salary * byPercent / 100;  
59.     salary += raise;  
60. }  
61.  
62. private String name;  
63. private double salary;  
64. private Date hireDay;  
65. }
```

4.3.2 多个源文件的使用

在例4-2中，一个源文件包含了两个类。许多程序员习惯于将每一个类存在一个单独的源文件中。例如，将Employee类存放在文件Employee.java中，将EmployeeTest类存放在文件EmployeeTest.java中。

如果喜欢这样组织文件，将可以有两种编译源程序的方法。一种是使用通配符调用Java编译器：

```
javac Employee*.java
```

于是，所有与通配符匹配的源文件都将被编译成类文件。或者键入下列命令：

```
javac EmployeeTest.java
```

读者可能会感到惊讶，使用第二种方式，并没有显式地编译Employee.java。然而，当Java编译器发现EmployeeTest.java使用了Employee类时会查找名为Employee.class的文件。如果没有找到这个文件，就会自动地搜索Employee.java，然后，对它进行编译。更重要的是：如果Employee.java版本较已有的Employee.class文件版本新，Java编译器就会自动地重新编译这个文件。



注释：如果熟悉UNIX的“make”工具（或者是Windows中的“nmake”等工具），就可以认为Java编译器内置了“make”功能。

4.3.3 解析Employee类

下面对Employee类进行一下剖析。首先从这个类的方法开始。通过查看源代码会发现，这个类包含一个构造器和4个方法：

```
public Employee(String n, double s, int year, int month, int day)  
public String getName()  
public double getSalary()  
public Date getHireDay()  
public void raiseSalary(double byPercent)
```

这个类的所有方法都被标记为public。关键字public意味着任何类的任何方法都可以调用这些方法（共有4种访问级别，将在本章稍后和下一章中介绍）。

接下来，需要注意在Employee类的实例中有三个实例域用来存放将要操作的数据：

```
private String name;  
private double salary;  
private Date hireDay;
```

关键字`private`确保只有`Employee`类自身的方法能够访问这些实例域，而其他类的方法不能够读写这些域。



注释：可以用`public`标记实例域，但这是一种极为不提倡的做法。`public`数据域允许程序中的任何方法对其进行读取和修改。这就完全破坏了封装。任何类的任何方法都可以修改`public`域，在我们的经历中，某些代码将使用这种存取权限，而这并不我们所希望的，因此，这里强烈建议将实例域标记为`private`。

最后，请注意，有两个实例域本身就是对象：`name`域是`String`类对象，`hireDay`域是`Date`类对象。这种情形十分常见：类通常包括类型属于某个类的实例域。

4.3.4 从构造器开始

下面先看看`Employee`类的构造器：

```
public Employee(String n, double s, int year, int month, int day)  
{  
    name = n;  
    salary = s;  
    GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);  
    hireDay = calendar.getTime();  
}
```

已经看到，构造器与类同名。在构造`Employee`类的对象时，构造器被运行，以便将实例域初始化为所希望的状态。

例如，当使用下面这条代码创建`Employee`类实例时：

```
new Employee("James Bond", 100000, 1950, 1, 1);
```

将会把实例域设置为：

```
name = "James Bond";  
salary = 100000;  
hireDay = January 1, 1950;
```


构造器与其他的方法有一个重要的不同。构造器总是伴随着`new`操作符的执行被调用，而不能对一个已经存在的对象调用构造器来达到重新设置实例域的目的。例如，

```
james.Employee("James Bond", 250000, 1950, 1, 1); // ERROR
```

将产生编译错误。


本章稍后，还会更加详细地介绍有关构造器的内容。现在只需要记住：

- 构造器与类同名
- 每个类可以有一个以上的构造器
- 构造器可以有0个、1个或1个以上的参数
- 构造器没有返回值
- 构造器总是伴随着`new`操作一起调用

 C++注释：Java构造器的工作方式与C++一样。但是，要记住所有的Java对象都是在堆中构造的，构造器总是伴随着new操作符一起使用。C++程序员最易犯的错误就是忘记new操作符：

```
Employee number007("James Bond", 100000, 1950, 1, 1);  
// C++, not Java
```

这条语句在C++中能够正常运行，但在Java中却不行。

 警告：请注意，不要在构造器中定义与实例域重名的局部变量。例如，下面的构造器将无法设置salary。

```
public Employee(String n, double s, . . . )  
{  
    String name = n; // ERROR  
    double salary = s; // ERROR  
    . . .  
}
```

这个构造器声明了局部变量name和salary。这些变量只能在构造器内部访问。这些变量屏蔽了同名的实例域。有些程序设计者（例如，本书的作者）常常不假思索地写出这类代码，因为他们习惯于增加这类数据类型。这种错误很难被检查出来，因此，必须注意在所有的方方法中不要命名与实例域同名的变量。

4.3.5 隐式参数与显式参数

方法用于操作对象以及存取它们的实例域。例如，方法：

```
public void raiseSalary(double byPercent)  
{  
    double raise = salary * byPercent / 100;  
    salary += raise;  
}
```

将调用这个方法的对象的salary实例域设置为新值。看看下面这个调用：

```
number007.raiseSalary(5);
```

它的结果将number007.salary域的值增加5%。具体地说，这个调用将执行下列指令：

```
double raise = number007.salary * 5 / 100;  
number007.salary += raise;
```

raiseSalary方法有两个参数。第一个参数被称为隐式（implicit）参数，是出现在方法名前的Employee类对象。第二个参数位于方法名后面括号中的数值，这是一个显式（explicit）参数。

已经看到，显式参数是明显地列在方法声明中的显示参数，例如double byPercent。隐式参数没有出现在方法声明中。

在每一个方法中，关键字this表示隐式参数。如果需要的话，可以用下列方式编写raiseSalary方法：

```
public void raiseSalary(double byPercent)  
{  
    double raise = this.salary * byPercent / 100;  
    this.salary += raise;  
}
```

有些程序员更偏爱这样的风格，因为这样可以将实例域与局部变量明显地区分开来。



C++注释：在C++中，通常在类的外面定义方法：

```
void Employee::raiseSalary(double byPercent) // C++, not Java
{
    . . .
}
```

如果在类的内部定义方法，这个方法将自动地成为内联（inline）方法。

```
class Employee
{
    . . .
    int getName() { return name; } // inline in C++
}
```

在Java程序设计语言中，所有的方法都必须在类的内部定义，但并不表示它们是内联方法。是否将某个方法设置为内联方法是Java虚拟机的任务。即时编译器会监视调用那些简洁、经常被调用、没有被重载以及可优化的方法。

4.3.6 封装的优点

最后，再仔细地看一下非常简单的getName方法、getSalary方法和getHireDay方法。

```
public String getName()
{
    return name;
}

public double getSalary()
{
    return salary;
}

public Date getHireDay()
{
    return hireDay;
}
```

这些都是典型的访问器方法。由于它们只返回实例域值，因此又被称为域访问器。

将name、salary和hireDay域标记为public，以此来取代独立的访问器方法会不会更容易些呢？

关键在于name是一个只读域。一旦在构造器中设置完毕，就没有任何一个办法可以对它进行修改，这样来确保name域不会受到外界的干扰。

虽然salary不是只读域，但是它只能用raiseSalary方法修改。特别是一旦这个域值出现了错误，只要调试这个方法就可以了。如果salary域是public的，破坏这个域值的捣乱者有可能会出没在任何地方。

在有些时候，需要获得或设置实例域的值。因此，应该提供下面三项内容：

- 一个私有的数据域；
- 一个公有的域访问器方法；
- 一个公有的域更改器方法。

这样做要比提供一个简单的公有数据域复杂些，但是却有着下列明显的好处：

1) 可以改变内部实现，除了该类的方法之外，不会影响其他代码。

例如，如果将存储名字的域改为：

```
String firstName;
String lastName;
```

那么getName方法可以改为返回

```
firstName + " " + lastName
```

对于这点改变，程序的其他部分完全不可见。

当然，为了进行新旧数据表示之间的转换，访问器方法和更改器方法有可能需要做许多工作。但是，这将为我们带来了第二点好处。

2) 更改器方法可以执行错误检查，然而直接对域进行赋值将不会进行这些处理。

例如，setSalary方法可以检查薪金是否小于0。



警告：注意不要编写返回引用可变对象的访问器方法。在Employee类中就违反了这个设计原则，其中的getHireDay方法返回了一个Date类对象：

```
class Employee
{
    . . .
    public Date getHireDay()
    {
        return hireDay;
    }
    . . .
    private Date hireDay;
}
```

这样会破坏封装性！请看下面这段代码：

```
Employee harry = . . . ;
Date d = harry.getHireDay();
double tenYearsInMilliseconds = 10 * 365.25 * 24 * 60 * 60 * 1000;
d.setTime(d.getTime() - (long) tenYearsInMilliseconds)
// let's give Harry ten years added seniority
```

出错的原因很微妙。d和harry.hireDay引用同一个对象（请参见图4-5）。对d调用更改器方法就可以自动地改变这个雇员对象的私有状态！

如果需要返回一个可变对象的引用，应该首先对它进行克隆（clone）。对象克隆是指存放在另一个位置上的对象副本。有关对象克隆的详细内容将在第6章中讨论。下面是修改后的代码：

```
class Employee
{
    . . .
    public Date getHireDay()
    {
        return (Date) hireDay.clone();
    }
    . . .
}
```

凭经验可知，如果需要返回一个可变数据域的拷贝，就应该使用克隆。

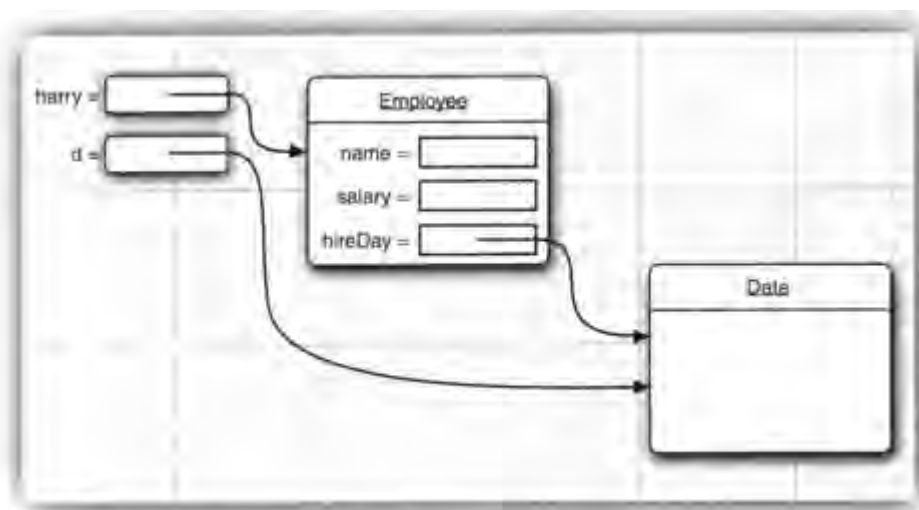


图4-5 返回可变数据域的引用

4.3.7 基于类的访问权限

从前面已经知道，方法可以访问所调用对象的私有数据。一个方法可以访问所属类的所有对象的私有数据，这令很多人感到奇怪！例如，下面看一下用来比较两个雇员的equals方法。

```
class Employee
{
    ...
    boolean equals(Employee other)
    {
        return name.equals(other.name);
    }
}
```

典型的调用方式是

```
if (harry.equals(boss)) ...
```

这个方法访问harry的私有域，这点并不会引发奇怪。然而，还访问boss的私有域。这是合法的，其原因是boss是Employee类对象，而Employee类的方法可以访问Employee类的任何一个对象的私有域。



C++注释：C++也有同样的原则。方法可以访问所属类的私有特性（feature），而不仅限于访问隐式参数的私有特性。

4.3.8 私有方法

在实现一个类时，由于公有数据非常危险，所以应该将所有的数据域都设置为私有的。然而，方法又应该如何设计呢？尽管绝大多数方法都被设计为公有的，但在某些特殊情况下，也可能将它们设计为私有的。有时，可能希望将一个计算代码划分成若干个独立的辅助方法。通常，这些辅助方法不应该成为公有接口的一部分，这是由于它们往往与当前的实现机制非常紧密，或者需要一个特别的协议以及一个特别的调用次序。这样的方法最好被设计为private的。

在Java中，为了实现一个私有的方法，只需要将关键字public改为private即可。

对于私有方法，如果改用其他方法实现相应的操作，则不必保留原有的方法。如果数据的表达方式发生了变化，这个方法可能会变得难以实现，或者不再需要。然而，只要方法是私有的，类的设计者就可以确信：它不会被外部的其他类操作调用，可以将其删去。如果方法是公有的，就不能将其删去，因为其他的代码很可能调用它。

4.3.9 Final实例域

可以将实例域定义为final。构建对象时必须初始化这样的域。也就是说，必须确保在每一个构造器执行之后，这个域的值被设置，并且在后面的操作中，不能够再对它进行修改。例如，可以将Employee类中的name域声明为final，因为在对象构建之后，这个值不会再被修改，即没有setName方法。

```
class Employee
{
    ...
    private final String name;
}
```

final修饰符大都应用于基本数据（primitive）类型域，或不可变（immutable）类的域（如果类中的每个方法都不会改变其对象，这种类就是不可变的类。例如，String类就是一个不可变的类）。对于可变的类，使用final修饰符可能会对读者造成混乱。例如，

```
private final Date hiredate;
```

仅仅意味着存储在hiredate变量中的对象引用在对象构造之后不能被改变，而并不意味着hiredate对象是一个常量。任何方法都可以对hiredate引用的对象调用setTime更改器。

4.4 静态域与静态方法

在前面给出的示例程序中，main方法都被标记为static修饰符。下面讨论一下这个修饰符的含义。

4.4.1 静态域

如果将域定义为static，每个类中只有一个这样的域。而每一个对象对于所有的实例域却都有自己的一份拷贝。例如，假定需要给每一个雇员赋予惟一的标识码。这里给Employee类添加一个实例域id和一个静态域nextId：

```
class Employee
{
    ...
    private int id;
    private static int nextId = 1;
}
```

现在，每一个雇员对象都有一个自己的id域，但这个类的所有实例将共享一个nextId域。换句话说，如果有1000个Employee类的对象，则有1000个实例域id。但是，只有一个静态域nextId。即使没有一个雇员对象，静态域nextId也存在。它属于类，而不属于任何独立的对象。



注释：在绝大多数的面向对象程序设计语言中，静态域被称为类域。术语“static”只是沿用了C++的叫法，并无实际意义。

下面实现一个简单的方法：

```
public void setId()
{
    id = nextId;
    nextId++;
}
```

假定为harry设定雇员标识码：

```
harry.setId();
```

harry的id域被设置为静态域nextId当前的值，并且静态域nextId的值加1：

```
harry.id = Employee.nextId;
Employee.nextId++;
```

4.4.2 静态常量

静态变量使用得比较少，但静态常量却使用得比较多。例如，在Math类中定义了一个静态常量：

```
public class Math
{
    . . .
    public static final double PI = 3.14159265358979323846;
    . . .
}
```

在程序中，可以采用Math.PI的形式获得这个常量。

如果关键字static被省略，PI就变成了Math类的一个实例域。需要通过Math类的对象访问PI，并且每一个Math对象都有它自己的一份PI拷贝。

另一个多次使用的静态常量是System.out。它在System类中声明：

```
public class System
{
    . . .
    public static final PrintStream out = . . .;
    . . .
}
```

前面曾经提到过，由于每个类对象都可以对公有域进行修改，所以，最好不要将域设计为public。然而，公有常量（即final域）却没问题。因为out被声明为final，所以，不允许再将其其他打印流赋给它：

```
System.out = new PrintStream(. . .); // ERROR--out is final
```



注释：如果查看一下System类，就会发现有一个setOut方法，它可以将System.out设置为不同的流。读者可能会感到奇怪，为什么这个方法可以修改final变量的值。原因在于，setOut方法是一个本地方法，而不是用Java语言实现的。本地方法可以绕过Java语言的存取控制机制。这是一种特殊的方法，在自己编写程序时，不应该这样处理。

4.4.3 静态方法

静态方法是一种不能向对象实施操作的方法。例如，Math类的pow方法就是一个静态方法。表达式

```
Math.pow(x, a)
```

计算 x^a 。在运算时，不使用任何Math对象。换句话说，没有隐式的参数。

可以认为静态方法是没有this参数的方法（在一个非静态的方法中，this参数表示这个方法的隐式参数）。

因为静态方法不能操作对象，所以不能在静态方法中访问实例域。但是，静态方法可以访问自身类中的静态域。下面是使用这种静态方法的一个示例：

```
public static int getNextId()
{
    return nextId; // returns static field
}
```

可以通过类名调用这个方法：

```
int n = Employee.getNextId();
```

这个方法可以省略关键字static吗？答案是肯定的。但是，需要通过Employee类对象的引用调用这个方法。



注释：可以使用对象调用静态方法。例如，如果harry是一个Employee对象，可以用harry.getNextId()代替Employee.getNextId()。不过，这种方式很容易造成混淆，其原因是getNextId方法计算的结果与harry毫无关系。我们建议使用类名，而不是对象来调用静态方法。

在下面两种情况下使用静态方法：

- 一个方法不需要访问对象状态，其所需参数都是通过显式参数提供（例如：Math.pow）。
- 一个方法只需要访问类的静态域（例如：Employee.getNextId）。



C++注释：Java中的静态域与静态方法在功能上与C++相同。但是，语法书写上却稍有所不同。在C++中，使用::操作符访问自身作用域之外的静态域和静态方法，如Math::PI。术语“static”有一段不寻常的历史。起初，C引入关键字static是为了表示退出一个块后依然存在的局部变量。在这种情况下，术语“static”是有意义的：变量一直存在，当再次进入该块时仍然存在。随后，static在C中有了第二种含义，表示不能被其他文件访问的全局变量和函数。为了避免引入一个新的关键字，关键字static被重用了。最后，C++第三次重用了这个关键字，与前面赋予的含义完全不一样，这里将其解释为：属于类且不属于类对象的变量和函数。这个含义与Java相同。

4.4.4 Factory方法

静态方法还有一种常见的用途。NumberFormat类使用factory方法产生不同风格的格式对象。

```
NumberFormat currencyFormatter = NumberFormat.getCurrencyInstance();
NumberFormat percentFormatter = NumberFormat.getPercentInstance();
double x = 0.1;
System.out.println(currencyFormatter.format(x)); // prints $0.10
System.out.println(percentFormatter.format(x)); // prints 10%
```

为什么NumberFormat类不利用构造器完成这些操作呢？这主要有两个原因：

- 无法命名构造器。构造器的名字必须与类名相同。但是，这里希望将得到的货币实例和

百分比实例采用不同的名字。

- 当使用构造器时，无法改变所构造的对象类型。而Factory方法将返回一个DecimalFormat类对象，这是NumberFormat的子类（有关继承的详细内容请参看第5章）。

4.4.5 Main方法

需要注意，不需要使用对象调用静态方法。例如，不需要构造Math类对象就可以调用Math.pow。

同理，main方法也是一个静态方法。

```
public class Application
{
    public static void main(String[] args)
    {
        // construct objects here
        . . .
    }
}
```

main方法不对任何对象进行操作。事实上，在启动程序时还没有任何一个对象。静态的main方法将执行并创建程序所需要的对象。



提示：每一个类可以有一个main方法。这是一个常用于对类进行单元测试的技巧。例如，可以在Employee类中添加一个main方法：

```
class Employee
{
    public Employee(String n, double s, int year, int month, int day)
    {
        name = n;
        salary = s;
        GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
        hireDay = calendar.getTime();
    }
    . . .
    public static void main(String[] args) // unit test
    {
        Employee e = new Employee("Romeo", 50000, 2003, 3, 31);
        e.raiseSalary(10);
        System.out.println(e.getName() + " " + e.getSalary());
    }
    . . .
}
```

如果想要独立地测试Employee类，只需要执行

```
java Employee
```

如果雇员类是大型应用程序的一部分，就可以使用下面这条语句运行程序

```
java Application
```

并且Employee类的main方法永远不会被执行。

例4-3中的程序包含了Employee类的一个简单版本，其中有一个静态域nextId和一个静态方法getNextId。这里将三个Employee对象写入数组，然后打印雇员信息。最后，打印出下一个可

用的员工标识码来作为对静态方法使用的演示。

需要注意，Employee类也有一个静态的main方法用于单元测试。试试运行

```
java Employee
```

和

```
java StaticTest
```

执行两个main方法。

例4-3 StaticTest.java

```
1. /**
2.  * This program demonstrates static methods.
3.  * @version 1.01 2004-02-19
4.  * @author Cay Horstmann
5.  */
6. public class StaticTest
7. {
8.     public static void main(String[] args)
9.     {
10.        // fill the staff array with three Employee objects
11.        Employee[] staff = new Employee[3];
12.
13.        staff[0] = new Employee("Tom", 40000);
14.        staff[1] = new Employee("Dick", 60000);
15.        staff[2] = new Employee("Harry", 65000);
16.
17.        // print out information about all Employee objects
18.        for (Employee e : staff)
19.        {
20.            e.setId();
21.            System.out.println("name=" + e.getName() + ",id=" + e.getId() + ",salary="
22.                               + e.getSalary());
23.        }
24.
25.        int n = Employee.getNextId(); // calls static method
26.        System.out.println("Next available id=" + n);
27.    }
28. }
29.
30. class Employee
31. {
32.     public Employee(String n, double s)
33.     {
34.         name = n;
35.         salary = s;
36.         id = 0;
37.     }
38.
39.     public String getName()
40.     {
41.         return name;
42.     }
43.
44.     public double getSalary()
45.     {
46.         return salary;
```

```
47. }
48.
49. public int getId()
50. {
51.     return id;
52. }
53.
54. public void setId()
55. {
56.     id = nextId; // set id to next available id
57.     nextId++;
58. }
59.
60. public static int getNextId()
61. {
62.     return nextId; // returns static field
63. }
64.
65. public static void main(String[] args) // unit test
66. {
67.     Employee e = new Employee("Harry", 50000);
68.     System.out.println(e.getName() + " " + e.getSalary());
69. }
70.
71. private String name;
72. private double salary;
73. private int id;
74. private static int nextId = 1;
75. }
```

4.5 方法参数

首先回顾一下在程序设计语言中有关将参数传递给方法（或函数）的一些专业术语。值调用（call by value）表示方法接收的是调用者提供的值。而引用调用（call by reference）表示方法接收的是调用者提供的变量地址。一个方法可以修改传递引用所对应的变量值，而不能修改传递值调用所对应的变量值。“.....调用”（call by）是一个标准的计算机科学术语，它用来描述各种程序设计语言中方法参数的传递方式（事实上，以前还有称（call by name），Algol程序设计语言是最古老的高级程序设计语言之一，它使用的就是这种参数传递方式。不过，对于今天，这种传递方式已经成为历史）。

Java程序设计语言总是采用值调用。也就是说，方法得到的是所有参数值的一个拷贝，特别是，方法不能修改传递给它的任何参数变量的内容。

例如，考虑下面的调用：

```
double percent = 10;
harry.raiseSalary(percent);
```

不必理睬这个方法的具体实现，在方法调用之后，percent的值还是10。

下面再仔细地研究一下这种情况。假定一个方法试图将一个参数值增加至3倍：

```
public static void tripleValue(double x) // doesn't work
{
    x = 3 * x;
}
```

然后调用这个方法：

```
double percent = 10;  
tripleValue(percent);
```

可以看到，无论怎样，调用这个方法之后，percent的值还是10。下面看一下具体的执行过程：

- 1) x被初始化为percent值的一个拷贝（也就是10）。
- 2) x被乘以3后等于30。但是percent仍然是10（如图4-6所示）。
- 3) 这个方法结束之后，参数变量x不再使用。

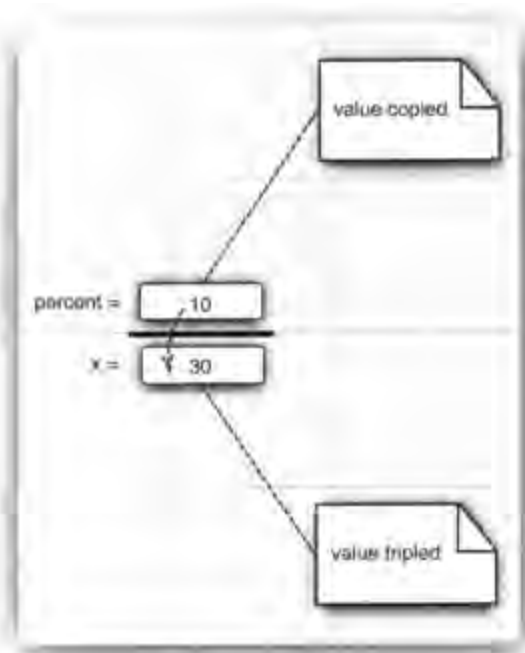


图4-6 对值参数的修改没有保留下来

然而，方法参数共有两种类型：

- 基本数据类型（数字、布尔值）。
- 对象引用。

读者已经看到，一个方法不可能修改一个基本数据类型的参数。而对象引用作为参数就不同了，可以很容易地利用下面这个方法实现将一个雇员的薪金提高两倍的操作：

```
public static void tripleSalary(Employee x) // works  
{  
    x.raiseSalary(200);  
}
```

当调用

```
harry = new Employee(. . .);  
tripleSalary(harry);
```

时，具体的执行过程为：

- 1) x被初始化为harry值的拷贝，这里是一个对象的引用。
- 2) `raiseSalary`方法应用于这个对象引用。x和harry同时引用的那个Employee对象的薪金提

高了200%。

3) 方法结束后，参数变量x不再使用。当然，对象变量harry继续引用那个薪金增至3倍的雇员对象（如图4-7所示）。

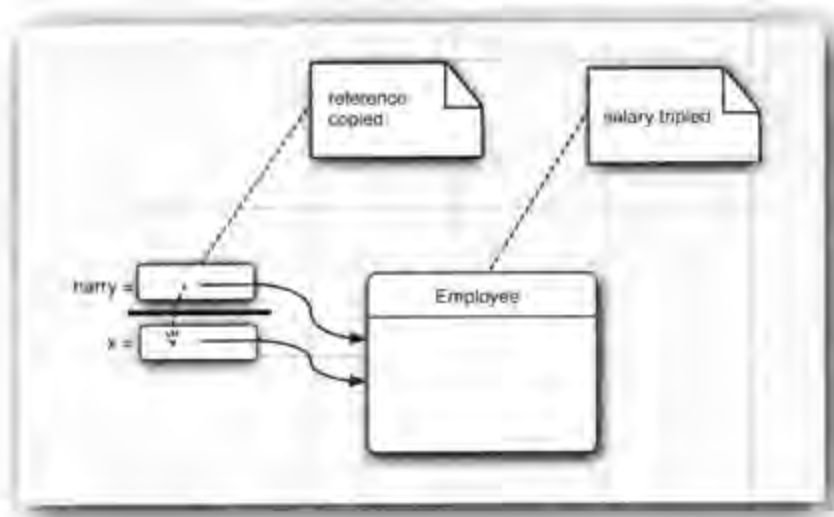


图4-7 对对象参数的修改保留了下来

读者已经看到，实现一个改变对象参数状态的方法并不是一件难事。理由很简单，方法得到的是对象引用的拷贝，对象引用及其他的拷贝同时引用同一个对象。

很多程序设计语言（特别是，C++和Pascal）提供了两种参数传递的方式：值调用和引用调用。有些程序员（甚至本书的作者）认为Java 程序设计语言对对象采用的是引用调用，实际上，这种理解是不对的。由于这种误解具有一定的普遍性，所以下面给出一个反例来详细地阐述一下这个问题。

首先，编写一个交换两个雇员对象的方法：

```
public static void swap(Employee x, Employee y) // doesn't work
{
    Employee temp = x;
    x = y;
    y = temp;
}
```

如果Java程序设计语言对对象采用的是引用调用，那么这个方法就应该能够实现交换数据的效果：

```
Employee a = new Employee("Alice", . . .);
Employee b = new Employee("Bob", . . .);
swap(a, b);
// does a now refer to Bob, b to Alice?
```

但是，方法并没有改变存储在变量a和b中的对象引用。swap方法的参数x和y被初始化为两个对象引用的拷贝，这个方法交换的是这两个拷贝。

```
// x refers to Alice, y to Bob
Employee temp = x;
x = y;
y = temp;
// now x refers to Bob, y to Alice
```

最终，白费力气。在方法结束时参数变量x和y被丢弃了。原来的变量a和b仍然引用这个方法调用之前所引用的对象（如图4-8所示）。

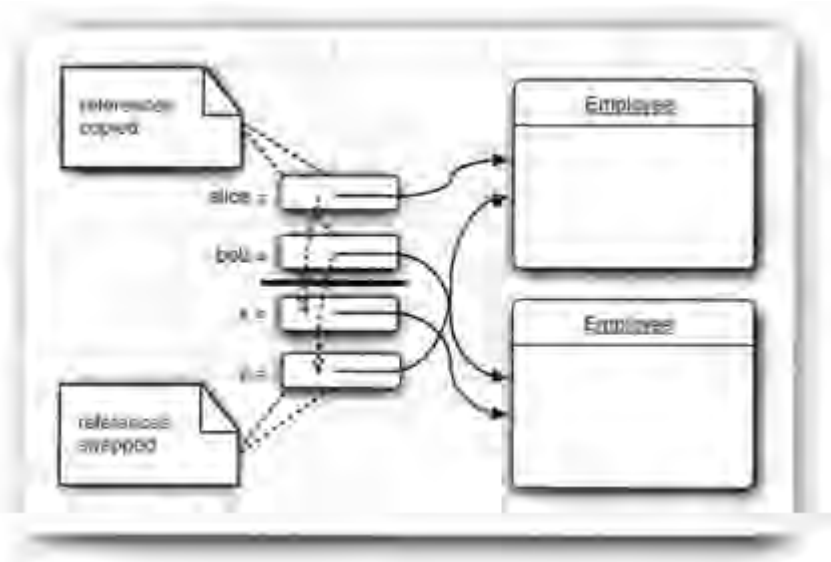


图4-8 交换对象参数的结果没有被保下来

这个过程说明：Java程序设计语言对对象采用的不是引用调用，实际上，对象引用进行的是值传递。

下面总结一下在Java程序设计语言中，方法参数的使用情况：

- 一个方法不能修改一个基本数据类型的参数（即数值型和布尔型）。
- 一个方法可以改变一个对象参数的状态。
- 一个方法不能实现让对象参数引用一个新的对象。

例4-4中的程序给出了相应的演示。在这个程序中，首先试图将一个值参数的值提高两倍，但没有成功：

```
Testing tripleValue:
Before: percent=10.0
End of method: x=30.0
After: percent=10.0
```

随后，成功地将一个雇员的薪金提高了两倍：

```
Testing tripleSalary:
Before: salary=50000.0
End of method: salary=150000.0
After: salary=150000.0
```

方法结束之后，harry引用的对象状态发生了改变。这是因为这个方法可以通过对象引用的拷贝修改所引用的对象状态。

最后，程序演示了swap方法的失败效果：

```
Testing swap:
Before: a=Alice
Before: b=Bob
End of method: x=Bob
```


End of method: y=Alice
After: a=Alice
After: b=Bob

可以看出，参数变量x和y被交换了，但是变量a和b没有受到影响。



C++注释：C++有值调用和引用调用。引用参数标有&符号。例如，可以轻松地实现void tripleValue(double& x) 方法或void swap(Employee& x, Employee& y) 方法实现修改它们的引用参数的目的。

例4-4 ParamTest.java

```
1. /**
2.  * This program demonstrates parameter passing in Java.
3.  * @version 1.00 2000-01-27
4.  * @author Cay Horstmann
5.  */
6. public class ParamTest
7. {
8.     public static void main(String[] args)
9.     {
10.         /*
11.          * Test 1: Methods can't modify numeric parameters
12.          */
13.         System.out.println("Testing tripleValue:");
14.         double percent = 10;
15.         System.out.println("Before: percent=" + percent);
16.         tripleValue(percent);
17.         System.out.println("After: percent=" + percent);
18.
19.         /*
20.          * Test 2: Methods can change the state of object parameters
21.          */
22.         System.out.println("\nTesting tripleSalary:");
23.         Employee harry = new Employee("Harry", 50000);
24.         System.out.println("Before: salary=" + harry.getSalary());
25.         tripleSalary(harry);
26.         System.out.println("After: salary=" + harry.getSalary());
27.
28.         /*
29.          * Test 3: Methods can't attach new objects to object parameters
30.          */
31.         System.out.println("\nTesting swap:");
32.         Employee a = new Employee("Alice", 70000);
33.         Employee b = new Employee("Bob", 60000);
34.         System.out.println("Before: a=" + a.getName());
35.         System.out.println("Before: b=" + b.getName());
36.         swap(a, b);
37.         System.out.println("After: a=" + a.getName());
38.         System.out.println("After: b=" + b.getName());
39.     }
40.
41.     public static void tripleValue(double x) // doesn't work
42.     {
43.         x = 3 * x;
44.         System.out.println("End of method: x=" + x);
```

```
45.     }
46.
47.     public static void tripleSalary(Employee x) // works
48.     {
49.         x.raiseSalary(200);
50.         System.out.println("End of method: salary=" + x.getSalary());
51.     }
52.
53.     public static void swap(Employee x, Employee y)
54.     {
55.         Employee temp = x;
56.         x = y;
57.         y = temp;
58.         System.out.println("End of method: x=" + x.getName());
59.         System.out.println("End of method: y=" + y.getName());
60.     }
61. }
62.
63. class Employee // simplified Employee class
64. {
65.     public Employee(String n, double s)
66.     {
67.         name = n;
68.         salary = s;
69.     }
70.
71.     public String getName()
72.     {
73.         return name;
74.     }
75.
76.     public double getSalary()
77.     {
78.         return salary;
79.     }
80.
81.     public void raiseSalary(double byPercent)
82.     {
83.         double raise = salary * byPercent / 100;
84.         salary += raise;
85.     }
86.
87.     private String name;
88.     private double salary;
89. }
```

4.6 对象构造

前面已经学会了编写简单的构造器，以便对定义的对象进行初始化。但是，由于对象构造非常重要，所以Java提供了多种编写构造器的方式。下面将详细地介绍一下。

4.6.1 重载

从前面可以看到，GregorianCalendar类有多个构造器。可以使用：

```
GregorianCalendar today = new GregorianCalendar();
```

或者

```
GregorianCalendar deadline = new GregorianCalendar(2099, Calendar.DECEMBER, 31);
```

这种特征叫做重载 (overloading)。如果多个方法 (比如, `GregorianCalendar` 构造器方法) 有相同的名字、不同的参数, 便产生了重载。编译器必须挑选出具体执行哪个方法, 它通过用各个方法给出的参数类型与特定方法调用所使用的值类型进行匹配来挑选出相应的方法。如果编译器找不到匹配的参数, 或者找出多个可能的匹配, 就会产生编译时错误 (这个过程被称为重载解析 (overloading resolution))。



注释: Java 允许重载任何方法, 而不只是构造器方法。因此, 要完整地描述一个方法, 需要指出方法名以及参数类型。这叫做方法的签名 (signature)。例如, `String` 类有 4 个称为 `indexOf` 的公有方法。它们的签名是

```
indexOf(int)
indexOf(int, int)
indexOf(String)
indexOf(String, int)
```

返回类型不是方法签名的一部分。也就是说, 不能有两个名字相同、参数类型也相同却返回不同类型值的方法。

4.6.2 默认域初始化

如果在构造器中没有显式地给域赋予初值, 那么就会被自动地赋为默认值: 数值为 0、布尔值为 `false`、对象引用为 `null`。然而, 只有缺少程序设计经验的人才会这样做。确实, 如果不明确地对域进行初始化, 就会影响程序代码的可读性。



注释: 这是域与局部变量的主要不同点。必须明确地初始化方法中的局部变量。但是, 如果没有初始化类中的域, 将会被初始化为默认值 (0、`false` 或 `null`)。

例如, 仔细看一下 `Employee` 类。假定没有在构造器中对某些域进行初始化, 就会默认地将 `salary` 域初始化为 0, 将 `name`、`hireDay` 域初始化为 `null`。

但是, 这并不是一种良好的编程习惯。如果此时调用 `getName` 方法或 `getHireDay` 方法, 则会得到一个 `null` 引用, 这应该不是我们所希望的结果:

```
Date h = harry.getHireDay();
calendar.setTime(h); // throws exception if h is null
```

4.6.3 默认构造器

所谓默认构造器是指没有参数的构造器。例如, `Employee` 类的默认构造器:

```
public Employee()
{
    name = "";
    salary = 0;
    hireDay = new Date();
}
```

如果在编写一个类时没有编写构造器, 那么系统就会提供一个默认构造器。这个默认构造

器将所有的实例域设置为默认值。于是，实例域中的数值型数据设置为0、布尔型数据设置为false、所有对象变量将设置为null。

如果类中提供了至少一个构造器，但是没有提供默认的构造器，则在构造对象时如果没有提供构造参数就会被视为不合法。例如，在例4-2中的Employee类提供了一个简单的构造器：

```
Employee(String name, double salary, int y, int m, int d)
```

对于这个类，构造默认的雇员属于不合法。也就是，调用

```
e = new Employee();
```

将会产生错误。



警告：请记住，仅当类没有提供任何构造器的时候，系统才会提供一个默认的构造器。如果在编写类的时候，给出了一个构造器，哪怕是很简单的，要想让这个类的用户能够采用下列方式构造实例：

```
new ClassName()
```

就必须提供一个默认的构造器（即不带参数的构造器）。当然，如果希望所有域被赋予默认值，可以采用下列格式：

```
public ClassName()
{
}
```

4.6.4 显式域初始化

由于类的构造器方法可以重载，所以可以采用多种形式设置类的实例域的初始状态。确保不管怎样调用构造器，每个实例域都可以被设置为一个有意义的初值。这是一种很好的设计习惯。

可以在类定义中，直接将一个值赋给任何域。例如：

```
class Employee
{
    . . .
    private String name = "";
}
```

在执行构造器之前，先执行赋值操作。当一个类的所有构造器都希望把相同的值赋予某个特定的实例域时，这种方式特别有用。

初始值不一定是常量。在下面的例子中，可以调用方法对域进行初始化。仔细看一下Employee类，其中每个雇员有一个id域。可以使用下列方式进行初始化：

```
class Employee
{
    . . .
    static int assignId()
    {
        int r = nextId;
        nextId++;
        return r;
    }
    . . .
    private int id = assignId();
}
```



C++注释：在C++中，不能直接初始化实例域。所有的域必须在构造器中设置。但是，有一个特殊的初始化器列表语法，如下所示：

```
Employee::Employee(String n, double s, int y, int m, int d) // C++
: name(n),
  salary(s),
  hireDay(y, m, d)
{
}
```

C++使用这种特殊的语法来调用域构造器。在Java中没有这种必要，因为对象没有子对象，只有指向其他对象的指针。

4.6.5 参数名

在编写很小的构造器时（这是十分常见的），常常在参数命名上出现错误。通常，参数用单个字符命名：

```
public Employee(String n, double s)
{
    name = n;
    salary = s;
}
```

但这样做有一个缺陷：只有阅读代码才能够了解参数n和参数s的含义。于是，有些程序员在每个参数前面加上一个前缀“a”：

```
public Employee(String aName, double aSalary)
{
    name = aName;
    salary = aSalary;
}
```

这样很清晰。每一个读者一眼就能够看懂参数的含义。

还一种常用的技巧，它基于这样的事实：参数变量用同样的名字将实例域屏蔽起来。例如，如果将参数命名为salary，salary将引用这个参数，而不是实例域。但是，可以采用this.salary的形式访问实例域。回想一下，this指示隐式参数，也就是被构造的对象。下面是一个示例

```
public Employee(String name, double salary)
{
    this.name = name;
    this.salary = salary;
}
```



C++注释：在C++中，经常用下划线或某个固定的字母（一般选用m或x）作为实例域的前缀。例如，salary域可能被命名为_salary、mSalary或xSalary。Java程序员通常不这样做。

4.6.6 调用另一个构造器

关键字this引用方法的隐式参数。然而，这个关键字还有另外一个含义。

如果构造器的第一个语句形如this(...)，这个构造器将调用同一个类的另一个构造器。下面是一个典型的例子：

```
public Employee(double s)
{
    // calls Employee(String, double)
    this("Employee #" + nextId, s);
    nextId++;
}
```

当调用`new Employee(60000)`时，`Employee(double)`构造器将调用`Employee(String, double)`构造器。

采用这种方式使用`this`关键字非常有用，这样对公共的构造器代码部分只编写一次即可。



C++注释：在Java中，`this`引用等价于C++的`this`指针。但是，在C++中，一个构造器不能调用另一个构造器。在C++中，必须将抽取出的公共初始化代码编写成一个独立的方法。

4.6.7 初始化块

前面已经讲过两种初始化数据域的方法：

- 在构造器中设置值
- 在声明中赋值

实际上，Java还有第三种机制，称为初始化块（`initialization block`）。在一个类的声明中，可以包含多个代码块。只要构造类的对象，这些块就会被执行。例如，

```
class Employee
{
    public Employee(String n, double s)
    {
        name = n;
        salary = s;
    }

    public Employee()
    {
        name = "";
        salary = 0;
    }
    ...
    private static int nextId;

    private int id;
    private String name;
    private double salary;
    ...

    // object initialization block
    {
        id = nextId;
        nextId++;
    }
}
```

在这个示例中，无论使用哪个构造器构造对象，`id`域都在对象初始化块中被初始化。首先运行初始化块，然后才运行构造器的主体部分。

这种机制不是必须的，也不常见。通常，直接将初始化代码放在构造器中。



注释：即使域定义在类的后半部分，在初始化块中仍然可以为它设置值。Sun的Java编译器的某些版本错误地处理了这种情况（bug # 4459133）。这个bug在Java SE 1.4.1中已经得到修正。但是，为了避免循环定义，不要读取在后面初始化的域。具体的规则请参看Java语言规范的8.3.2.3节（<http://java.sun.com/docs/books/jls>）。这个规则的复杂度足以使编译器的实现者头疼，因此建议将初始化块放在域定义之后。

由于初始化数据域有多种途径，所以列出构造过程的所有路径可能相当混乱。下面是调用构造器的具体处理步骤：

- 1) 所有数据域被初始化为默认值（0、false或null）。
- 2) 按照在类声明中出现的次序，依次执行所有域初始化语句和初始化块。
- 3) 如果构造器第一行调用了第二个构造器，则执行第二个构造器主体。
- 4) 执行这个构造器的主体。

当然，应该精心地组织好初始化代码，这样有利于其他程序员的理解。例如，如果让类的构造器行为依赖于数据域声明的顺序，那就会显得很奇怪并且容易引起错误。

可以通过提供一个初始化值，或者使用一个静态的初始化块来对静态域进行初始化。前面已经介绍过第一种机制：

```
static int nextId = 1;
```

如果对类的静态域进行初始化的代码比较复杂，那么可以使用静态的初始化块。

将代码放在一个块中，并标记关键字static。下面是一个示例。其功能是将雇员ID的起始值赋予一个小于10 000的随机整数。

```
// static initialization block
static
{
    Random generator = new Random();
    nextId = generator.nextInt(10000);
}
```

在类第一次加载的时候，将会进行静态域的初始化。与实例域一样，除非将它们显式地设置成其他值，否则默认的初始值是 0、false或null。所有的静态初始化语句以及静态初始化块都将依照类定义的顺序执行。



注释：使用下面这种方式，在同伴们的面前显露一手：可以使用Java编写一个没有main方法的“Hello, World”程序。

```
public class Hello
{
    static
    {
        System.out.println("Hello, World");
    }
}
```

当用java Hello调用这个类时，这个类就被加载，静态初始化块将会打印“Hello, World”。在此之后，会得到一个“main is not defined（没有定义）”的错误信息。不过，可以在静态初始化块的尾部调用System.exit(0)避免这一缺陷。

例4-5中的程序展示了本节论述的很多特性：

- 重载构造器
- 用 `this(...)` 调用另一个构造器
- 默认构造器
- 对象初始化块
- 静态初始化块
- 实例域初始化

例4-5 ConstructorTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates object construction.
5.  * @version 1.01 2004-02-19
6.  * @author Cay Horstmann
7.  */
8. public class ConstructorTest
9. {
10.     public static void main(String[] args)
11.     {
12.         // fill the staff array with three Employee objects
13.         Employee[] staff = new Employee[3];
14.
15.         staff[0] = new Employee("Harry", 40000);
16.         staff[1] = new Employee(60000);
17.         staff[2] = new Employee();
18.
19.         // print out information about all Employee objects
20.         for (Employee e : staff)
21.             System.out.println("name=" + e.getName() + ",id=" + e.getId() + ",salary="
22.                 + e.getSalary());
23.     }
24. }
25.
26. class Employee
27. {
28.     // three overloaded constructors
29.     public Employee(String n, double s)
30.     {
31.         name = n;
32.         salary = s;
33.     }
34.
35.     public Employee(double s)
36.     {
37.         // calls the Employee(String, double) constructor
38.         this("Employee #" + nextId, s);
39.     }
40.
41.     // the default constructor
42.     public Employee()
43.     {
```



```
44.     // name initialized to ""--see below
45.     // salary not explicitly set--initialized to 0
46.     // id initialized in initialization block
47. }
48.
49. public String getName()
50. {
51.     return name;
52. }
53.
54. public double getSalary()
55. {
56.     return salary;
57. }
58.
59. public int getId()
60. {
61.     return id;
62. }
63.
64. private static int nextId;
65.
66. private int id;
67. private String name = ""; // instance field initialization
68. private double salary;
69.
70. // static initialization block
71. static
72. {
73.     Random generator = new Random();
74.     // set nextId to a random number between 0 and 9999
75.     nextId = generator.nextInt(10000);
76. }
77.
78. // object initialization block
79. {
80.     id = nextId;
81.     nextId++;
82. }
83. }
```

java.util.Random 1.0

- Random()
构造一个新的随机数生成器。
- int nextInt(int n) 1.2
返回一个0 ~ n - 1之间的随机数。

4.6.8 对象析构与finalize方法

有些面向对象的程序设计语言，特别是C++，有显式的析构器方法，其中放置一些当对象不再使用时需要执行的清理代码。在析构器中，最常见的操作是回收分配给对象的存储空间。由于Java有自动的垃圾回收器，不需要人工回收内存，所以Java不支持析构器。

当然，某些对象使用了内存之外的其他资源，例如，文件或使用了系统资源的另一个对象的句柄。在这种情况下，当资源不再需要时，将其回收和再利用将显得十分重要。

可以为任何一个类添加finalize方法。finalize方法将在垃圾回收器清除对象之前调用。在实际应用中，不要依赖于使用finalize方法回收任何短缺的资源，这是因为很难知道这个方法什么时候才能够调用。



注释：有个名为System.runFinalizersOnExit(true)的方法能够确保finalizer方法在Java关闭前被调用。不过，这个方法并不安全，也不鼓励大家使用。有一种代替的方法是使用方法Runtime.addShutdownHook添加“关闭钩”(shutdown hook)，详细内容请参看API文档。

如果某个资源需要在使用完毕后立刻被关闭，那么就需要由人工来管理。可以应用一个类似dispose或close的方法完成相应的清理操作。特别需要说明，如果一个类使用了这样的方法，当对象不再被使用时一定要调用它。

4.7 包

Java允许使用包(package)将类组织起来。借助于包可以方便地组织自己的代码，并将自己的代码与别人提供的代码库分开管理。

标准的Java类库分布在多个包中，包括java.lang、java.util和java.net等。标准的Java包具有一个层次结构。如同硬盘的目录嵌套一样，也可以使用嵌套层次组织包。所有标准的Java包都处于java和javax包层次中。

使用包的主要原因是确保类名的惟一性。假如两个程序员不约而同地建立了Employee类。只要将这些类放置在不同的包中，就不会产生冲突。事实上，为了保证包名的绝对惟一性，Sun公司建议将公司的因特网域名（这显然是独一无二的）以逆序的形式作为包名，并且对于不同的项目使用不同的子包。例如，horstmann.com是本书作者之一注册的域名。逆序形式为com.horstmann。这个包还可以被进一步地划分成子包，如com.horstmann.corejava。

从编译器的角度来看，嵌套的包之间没有任何关系。例如，java.util包与java.util.jar包毫无关系。每一个都拥有独立的类集合。

4.7.1 类的导入

一个类可以使用所属包中的所有类，以及其他包中的公有类(public class)。我们可以采用两种方式访问另一个包中的公有类。第一种方式是在每个类名之前添加完整的包名。例如：

```
java.util.Date today = new java.util.Date();
```

这显然很令人生厌。更简单且更常用的方式是使用import语句。import语句是一种引用包含在包中的类的简明描述。一旦使用了import语句，在使用类时，就不必写出包的全名了。

可以使用import语句导入一个特定的类或者整个包。import语句应该位于源文件的顶部（但位于package语句的后面）。例如，可以使用下面这条语句导入java.util包中所有的类。

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

然后，就可以使用

```
Date today = new Date();
```

而无需在前面加上包前缀。还可以导入一个包中的特定类：

```
import java.util.Date;
```

java.util.*的语法比较简单，对代码的大小也没有任何负面影响。当然，如果能够明确地指出所导入的类，将会使代码的读者更加准确地知道加载了哪些类。



提示：在Eclipse中，可以使用菜单选项Source→Organize Imports。Package语句，如import java.util.*，将会自动地扩展指定的导入列表，如：import java.util.ArrayList; import java.util.Date; 这是一个十分便捷的特性。

但是，需要注意的是，只能使用星号（*）导入一个包，而不能使用import java.*或import java.*.* 导入以java为前缀的所有包。

在大多数情况下，只导入所需的包，并不必过多地埋睬它们。但在发生命名冲突的时候，就不能不注意包的名字了。例如，java.util和java.sql包都有日期（Date）类。如果在程序中导入了这两个包：

```
import java.util.*;
import java.sql.*;
```

在程序使用Date类的时候，就会出现一个编译错误：

```
Date today; // ERROR--java.util.Date or java.sql.Date?
```

此时编译器无法确定程序使用的是哪一个Date类。可以采用增加一个特定的import语句来解决这个问题：

```
import java.util.*;
import java.sql.*;
import java.util.Date;
```

如果这两个Date类都需要使用，又该怎么办呢？答案是，在每个类名的前面加上完整的包名。

```
java.util.Date deadline = new java.util.Date();
java.sql.Date today = new java.sql.Date(...);
```

在包中定位类是编译器（compiler）的工作。类文件中的字节码肯定使用完整的包名来引用其他类。



C++注释：C++程序员经常将import与#include弄混。实际上，这两者之间并没有共同之处。在C++中，必须使用#include将外部特性的声明加载进来，这是因为C++编译器无法查看任何文件的内部，除了正在编译的文件以及在头文件中明确包含的文件。Java编译器可以查看其他文件的内部，只要告诉它到哪里去查看就可以了。

在Java中，通过显式地给出包名，如java.util.Date，就可以不使用import；而在C++中，无法避免使用#include。

Import语句的惟一的好处是简捷。可以使用简短的名字而不是完整的包名来引用一个类。例如，在import java.util.*（或import java.util.Date）语句之后，可以仅仅用Date引用java.util.Date类。

在C++中，与包机制类似的是命名空间（namespace）。在Java中，package与import语句类

似于C++中的namespace和using指令 (directive)。

4.7.2 静态导入

从Java SE 5.0开始, import语句不仅可以导入类, 还增加了导入静态方法和静态域的功能。

例如, 如果在源文件的顶部, 添加一条指令:

```
import static java.lang.System.*;
```

就可以使用System类的静态方法和静态域, 而不必加类名前缀:

```
out.println("Goodbye, World!"); // i.e., System.out
exit(0); // i.e., System.exit
```

另外, 还可以导入特定的方法或域:

```
import static java.lang.System.out;
```

实际上, 是否有更多的程序员采用System.out或System.exit的简写形式, 似乎是一件值得怀疑的事情。这种编写形式不利于代码的清晰度。不过, 静态导入有两个实际的应用。

- 算术函数: 如果对Math类使用静态导入, 就可以采用更加自然的方式使用算术函数。例如,

```
sqrt(pow(x, 2) + pow(y, 2))
```

看起来比

```
Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2))
```

清晰得多。

- 笨重的常量: 如果需要使用大量带有冗长名字的常量, 就应该使用静态导入。例如,

```
if (d.get(DAY_OF_WEEK) == MONDAY)
```

看起来比

```
if (d.get(Calendar.DAY_OF_WEEK) == Calendar.MONDAY)
```

容易得多。

4.7.3 将类放入包中

要想将一个类放入包中, 就必须将包的名字放在源文件的开头, 包中定义类的代码之前。

例如, 例4-7中的文件Employee.java开头是这样的:

```
package com.horstmann.corejava;
```

```
public class Employee
{
    . . .
}
```

如果没有在源文件中放置package语句, 这个源文件中的类就被放置在一个默认包 (default package) 中。默认包是一个没有名字的包。在此之前, 我们定义的所有类都在默认包中。

将包中的文件放到与完整的包名匹配的子目录中。例如, com.horstmann.corejava包中的所有源文件应该被放置在子目录com/horstmann/corejava (Windows中com\horstmann\corejava) 中。编译器将类文件也放在相同的目录结构中。

例4-6和例4-7中的程序分放在两个包中: PackageTest类放置在默认包中; Employee类放置

在com.horstmann.corejava包中。因此，Employee.class文件必须包含在子目录com/horstmann/corejava中。换句话说，目录结构如下所示：

```
. 基目录
├── PackageTest.java
├── PackageTest.class
└── com/
    └── horstmann/
        └── corejava/
            ├── Employee.java
            └── Employee.class
```

要想编译这个程序，只需改变基目录，并运行命令

```
javac PackageTest.java
```

编译器就会自动地查找文件com/horstmann/corejava/Employee.java并进行编译。

下面看一个更加实际的例子。在这里不使用默认包，而是将类分别放在不同的包中（com.horstmann.corejava和com.mycompany）。

```
. 基目录
└── com/
    ├── horstmann/
    │   └── corejava/
    │       ├── Employee.java
    │       └── Employee.class
    └── mycompany/
        ├── PayrollApp.java
        └── PayrollApp.class
```

在这种情况下，仍然要从基目录编译和运行类，即包含com目录：

```
javac com/mycompany/PayrollApp.java
java com.mycompany.PayrollApp
```

需要注意，编译器对文件（带有文件分隔符和扩展名.java的文件）进行操作。而Java解释器加载类（带有分隔符）。



警告：编译器在编译源文件的时候不检查目录结构。例如，假定有一个源文件开头有下列语句：

```
package com.mycompany;
```

即使这个源文件没有在子目录com/mycompany下，也可以进行编译。如果它不依赖于其他包，就不会出现编译错误。但是，最终的程序将无法运行，这是因为虚拟机找不到类文件。

例4-6 PackageTest.java

```
1. import com.horstmann.corejava.*;
2. // the Employee class is defined in that package
3.
4. import static java.lang.System.*;
5.
6. /**
```

```
7.  * This program demonstrates the use of packages.
8.  * @author cay
9.  * @version 1.11 2004-02-19
10. * @author Cay Horstmann
11. */
12. public class PackageTest
13. {
14.     public static void main(String[] args)
15.     {
16.         // because of the import statement, we don't have to use com.horstmann.corejava.Employee here
17.         Employee harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
18.
19.         harry.raiseSalary(5);
20.
21.         // because of the static import statement, we don't have to use System.out here
22.         out.println("name=" + harry.getName() + ", salary=" + harry.getSalary());
23.     }
24. }
```

例4-7 Employee.java

```
1. package com.horstmann.corejava;
2.
3. // the classes in this file are part of this package
4.
5. import java.util.*;
6.
7. // import statements come after the package statement
8.
9. /**
10.  * @version 1.10 1999-12-18
11.  * @author Cay Horstmann
12.  */
13. public class Employee
14. {
15.     public Employee(String n, double s, int year, int month, int day)
16.     {
17.         name = n;
18.         salary = s;
19.         GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
20.         // GregorianCalendar uses 0 for January
21.         hireDay = calendar.getTime();
22.     }
23.
24.     public String getName()
25.     {
26.         return name;
27.     }
28.
29.     public double getSalary()
30.     {
31.         return salary;
32.     }
33.
34.     public Date getHireDay()
35.     {
```

```
36.     return hireDay;
37. }
38.
39. public void raiseSalary(double byPercent)
40. {
41.     double raise = salary * byPercent / 100;
42.     salary += raise;
43. }
44.
45. private String name;
46. private double salary;
47. private Date hireDay;
48. }
```

4.7.4 包作用域

前面已经接触过访问修饰符public和private。标记为public的部分可以被任意的类使用；标记为private的部分只能被定义它们的类使用。如果没有指定public或private，这个部分（类、方法或变量）可以被同一个包中的所有方法访问。

下面再仔细地看一下例4-2中的程序。在这个程序中，没有将Employee类定义为公有类，因此只有在同一个包（在此是默认包）中的其他类可以访问，例如EmployeeTest。对于类来说，这种默认是合乎情理的。但是，对于变量来说就有些不适宜了，因此变量必须显式地标记为private，不然的话将默认为包可见。显然，这样做会破坏封装性。问题主要出于人们经常忘记键入关键字private。在java.awt包中的Window类就是一个典型的示例。java.awt包是JDK提供的部分源代码：

```
public class Window extends Container
{
    String warningString;
    . . .
}
```

请注意，这里的warningString变量不是private！这意味着java.awt包中的所有类的方法都可以访问该变量，并将它设置为任意值（例如，“Trust me!”）。实际上，只有Window类的方法访问它，因此应该将它设置为私有变量。我们猜测可能是程序员匆忙之中忘记键入private修饰符了（为防止程序员内疚，我们没有说出他的名字，感兴趣的话，可以查看一下源代码）。



注释：奇怪的是，这个问题至今还没有得到纠正，即使我们在这本书的8个版本中已经指出了这一点。很明显，类库的实现者并没有读这本书。不仅如此，这个类还增加了一些新域，其中大约一半也不是私有的。

这真的会成为一个问题吗？答案是：视具体情况而定。在默认情况下，包不是一个封闭的实体。也就是说，任何人都可以向包中添加更多的类。当然，有敌意或低水平的程序员很可能利用包的可见性添加一些具有修改变量功能的代码。例如，在Java程序设计语言的早期版本中，只需要将下列这条语句放在类文件的开头，就可以很容易地将其他类混入java.awt包中：

```
package java.awt;
```

然后，把结果类文件放置在类路径某处的java/awt子目录下，就可以访问java.awt包的内部了。使用这一手段，可以对警告框进行设置（如图4-9所示）。

从1.2版开始，JDK的实现者修改了类加载器，明确地禁止加载用户自定义的、包名以“java.”开始的类！当然，用户自定义的类无法从这种保护中受益。然而，可以通过包密封（package sealing）机制来解决将各种包混杂在一起的问题。如果将一个包密封起来，就不能再向这个包添加类了。在第10章中，将介绍制作包含密封包的JAR文件的方法。



图4-9 修改在applet窗口中的警告字符串

4.8 类路径

在前面已经看到，类存储在文件系统的子目录中。类的路径必须与包名匹配。

另外，类文件也可以存储在JAR(Java归档)文件中。在一个JAR文件中，可以包含多个压缩形式的类文件和子目录，这样既可以节省又可以改善性能。在程序中用到第三方（third-party）的库文件时，通常会给出一个或多个需要包含的JAR文件。JDK也提供了许多的JAR文件，例如，在jre/lib子目录下包含数千个类库文件。有关创建JAR文件的详细内容将在第10章中讨论。



提示：JAR文件使用ZIP格式组织文件和子目录。可以使用所有ZIP实用程序查看内部的rt.jar以及其他的JAR文件。

为了使类能够被多个程序共享，需要做到下面几点：

1) 把类放到一个目录中，例如/home/user/classdir。需要注意，这个目录是包树状结构的基目录。如果希望将com.horstmann.corejava.Employee类添加到其中，这个Employee.class类文件就必须位于子目录/home/user/classdir/com/horstmann/corejava中。

2) 将JAR文件防在一个目录中，例如：/home/user/archives。

3) 设置类路径（class path）。类路径是所有包含类文件的路径的集合。

在UNIX环境中，类路径中的不同项目之间采用冒号（:）分隔：

```
/home/user/classdir:./home/user/archives/archive.jar
```

而在Windows环境中，则以分号（;）分隔：

```
c:\classdir;.;c:\archives\archive.jar
```

在上述两种情况中，句点（.）表示当前目录。

类路径包括：

- 基目录 /home/user/classdir 或 c:\classes；
- 当前目录(.);
- JAR文件/home/user/archives/archive.jar或c:\archives\archive.jar。

从Java SE 6开始，可以在JAR文件目录中指定通配符，如下：

```
/home/user/classdir:./home/user/archives/'*'
```

或者

```
c:\classdir;.;c:\archives\*
```


但在UNIX中，禁止使用* 以防止shell命令进一步扩展。

在归档目录中的所有JAR文件（但不包括.class文件）都包含在类路径中。

由于运行时库文件（rt.jar和在jre/lib与jre/lib/ext目录下的一些其他的JAR文件）会被自动地搜索，所以不必将它们显式地列在类路径中。



警告：javac编译器总是在当前的目录中查找文件，但Java虚拟机仅在类路径中有“.”目录的时候才查看当前目录。如果没有设置类路径，那也并不会产生什么问题，默认的路径包含“.”目录。然而如果设置了类路径却忘记了包含“.”目录，则程序仍然可以通过编译，但不能运行。

类路径所列出的目录和归档文件是搜寻类的起始点。下面看一个类路径示例：

```
/home/user/classdir:./home/user/archives/archive.jar
```

假定虚拟机要搜寻com.horstmann.corejava.Employee类文件。它首先要查看存储在jre/lib和jre/lib/ext目录下的归档文件中所存放的系统类文件。显然，在那里找不到相应的类文件，然后再查看类路径。于是查看：

- /home/user/classdir/com/horstmann/corejava/Employee.class
- com/horstmann/corejava/Employee.class从当前目录开始
- com/horstmann/corejava/Employee.class inside /home/user/archives/archive.jar

编译器定位文件要比虚拟机复杂得多。如果引用了一个类，而没有指出这个类所在的包，那么编译器将首先查找包含这个类的包，并询查所有的import指令，确定其中是否包含了被引用的类。例如，假定源文件包含指令：

```
import java.util.*;
import com.horstmann.corejava.*;
```

并且源代码引用了Employee类。编译器将试图查找java.lang.Employee（因为java.lang包被默认导入）、java.util.Employee、com.horstmann.corejava.Employee和当前包中的Employee。对这个类路径的所有位置中所列出的每一个类进行逐一查看。如果找到了一个以上的类，就会产生编译错误（因为类必须是惟一的，而import语句的次序却无关紧要）。

编译器的任务不止这些，它还要查看源文件（Source files）是否比类文件新。如果是这样的话，那么源文件就会被自动地重新编译。在前面已经知道，仅可以导入其他包中的公有类。一个源文件只能包含一个公有类，并且文件名必须与公有类匹配。因此，编译器很容易定位公有类所在的源文件。当然，也可以从当前包中导入非公有类。这些类有可能定义在与类名不同的源文件中。如果从当前包中导入一个类，编译器就要搜索当前包中的所有源文件，以便确定哪个源文件定义了这个类。

设置类路径

最好采用-classpath（或-cp）选项指定类路径：

```
java -classpath /home/user/classdir:./home/user/archives/archive.jar MyProg.java
```

或者

```
java -classpath c:\classdir;.;c:\archives\archive.jar MyProg.java
```

所有的指令应该书写在一行中。将这样一个长的命令行放在一个shell脚本或一个批处理文件中是一个不错的主意。

利用-classpath选项设置类路径是首选的方法，也可以通过设置CLASSPATH环境变量完成这个操作。其详细情况依赖于所使用的shell。命令格式如下：

```
export CLASSPATH=/home/user/classdir:./home/user/archives/archive.jar
```

在C shell中，命令格式如下：

```
setenv CLASSPATH /home/user/classdir:./home/user/archives/archive.jar
```

在Windows shell，命令格式如下：

```
set CLASSPATH=c:\classdir;.;c:\archives\archive.jar
```

直到退出shell为止，类路径设置均有效。



警告：有人建议将CLASSPATH环境变量设置为永久不变的值。总的来说这是一个很糟糕的主意。人们有可能会忘记全局设置，因此，当使用的类没有正确地加载进来时，会感到很奇怪。一个应该受到谴责的示例是Windows中Apple的QuickTime安装程序。它进行了全局设置，CLASSPATH指向一个所需要的JAR文件，但并没有在类路径上包含当前路径。因此，当程序编译后却不能运行时，众多的Java程序员花费了很多精力去解决这个问题。



警告：有人建议绕开类路径，将所有的文件放在jre/lib/ext路径。这是一个极坏的主意，其原因主要有两个：当手工地加载其他的类文件时，如果将它们存放在扩展路径上，则不能正常地工作（有关类加载器的详细信息，请参看卷II第9章）。此外，程序员经常会忘记3个月前所存放文件的位置。当类加载器忽略了曾经仔细设计的类路径时，程序员会毫无头绪地在头文件中查找。事实上，加载的是扩展路径上已长时间遗忘的类。

4.9 文档注释

JDK包含一个很有用的工具，叫做javadoc，它可以由源文件生成一个HTML文档。事实上，在第3章讲述的联机API文档就是通过对标准Java类库的源代码运行javadoc生成的。

如果在源代码中添加以专用的定界符/**开始的注释，那么可以很容易地生成一个看上去具有专业水准的文档。这是一种很好的方式，因为这种方式可以将代码与注释保存在一个地方。如果将文档存入一个独立的文件中，就有可能会随着时间的推移，出现代码和注释不一致的问题。然而，由于文档注释与源代码在同一个文件中，在修改源代码的同时，重新运行javadoc就可以轻而易举地保持两者的一致性。

4.9.1 注释的插入

javadoc实用程序（utility）从下面几个特性中抽取信息：

- 包
- 公有类与接口
- 公有的和受保护的方法

- 公有的和受保护的域

在第5章中将介绍受保护特性，在第6章将介绍接口。

应该为上面几部分编写注释。注释应该放置在所描述特性的前面。注释以 `/**` 开始，并以 `*/` 结束。

每个 `/** ... */` 文档注释在标记之后紧跟着自由格式文本 (free-form text)。标记由 `@` 开始，如 `@author` 或 `@param`。

自由格式文本的第一句应该是一个概要性的句子。javadoc实用程序自动地将这些句子抽取出来形成概要页。

在自由格式文本中，可以使用HTML修饰符，例如，用于强调的 `...`、用于设置等宽“打字机”字体的 `<code>...</code>`、用于着重强调的 `...` 以及包含图像的 `` 等。不过，一定不要使用 `<h1>` 或 `<hr>`，因为它们会与文档的格式产生冲突。



注释：如果文档中有到其他文件的链接，例如，图像文件（用户界面的组件的图表或图像等），就应该将这些文件放到子目录 `doc-files` 中。javadoc实用程序将从源目录拷贝这些目录及其中的文件到文档目录中。在链接中需要使用 `doc-files` 目录，例如：``。

4.9.2 类注释

类注释必须放在 `import` 语句之后，类定义之前。

下面是一个类注释的例子：

```
/**
 * A Card object represents a playing card, such
 * as "Queen of Hearts". A card has a suit (Diamond, Heart,
 * Spade or Club) and a value (1 = Ace, 2 . . . 10, 11 = Jack,
 * 12 = Queen, 13 = King)
 */
public class Card
{
    . . .
}
```



注释：没有必要在每一行的开始用星号*，例如：

```
/**
  A Card object represents a playing card, such
  as "Queen of Hearts". A card has a suit (Diamond, Heart,
  Spade or Club) and a value (1 = Ace, 2 . . . 10, 11 = Jack,
  12 = Queen, 13 = King).
*/
```

然而，大部分IDE提供了自动添加星号*，并且当注释行改变时，自动重新排列这些星号的功能。

4.9.3 方法注释

每一个方法注释必须放在所描述的方法之前。除了通用标记之外，还可以使用下面的标记：

- @param variable description

这个标记将对当前方法的 “ param ” (参数) 部分添加一个条目。这个描述可以占据多行，并可以使用HTML标记。一个方法的所有 @param 标记必须放在一起。

- @return description

这个标记将对当前方法添加 “ return ” (返回) 部分。这个描述可以跨越多行，并可以使用HTML标记。

- @throws class description

这个标记将添加一个注释，用于表示这个方法有可能抛出异常。有关异常的详细内容将在第11章中讨论。

下面是一个方法注释的示例：

```
/**
 * Raises the salary of an employee.
 * @param byPercent the percentage by which to raise the salary (e.g. 10 = 10%)
 * @return the amount of the raise
 */
public double raiseSalary(double byPercent)
{
    double raise = salary * byPercent / 100;
    salary += raise;
    return raise;
}
```

4.9.4 域注释

只需要对公有域（通常指的是静态常量）建立文档。例如，

```
/**
 * The "Hearts" card suit
 */
public static final int HEARTS = 1;
```

4.9.5 通用注释

下面的标记可以用在类文档的注释中。

- @author name

这个标记将产生一个 “ author ” (作者) 条目。可以使用多个 @author 标记，每个 @author 标记对应一名作者。

- @version text

这个标记将产生一个 “ version ” (版本) 条目。这里的text可以是对当前版本的任何描述。

下面的标记可以用于所有的文档注释。

- @since text

这个标记将产生一个 “ since ” (始于) 条目。这里的text可以是对引入特性的版本描述。例如，@since version 1.7.1。

- @deprecated text

这个标记将对类、方法或变量添加一个不再使用的注释。text中给出了取代的建议。例

如，

```
@deprecated Use <code>setVisible(true)</code> instead
```

通过@see和@link标记，可以使用超级链接，链接到javadoc文档的相关部分或外部文档。

- @see reference

这个标记将在“see also”部分增加一个超级链接。它可以用于类中，也可以用于方法中。这里的reference可以选择下列情形之一：

```
package.class#feature label  
<a href="...">label</a>  
"text"
```

第一种情况是最常见的。只要提供类、方法或变量的名字，javadoc就在文档中插入一个超链接。例如，

```
@see com.horstmann.corejava.Employee#raiseSalary(double)
```

建立一个链接到com.horstmann.corejava.Employee类的raiseSalary(double)方法的超链接。可以省略包名，甚至把包名和类名都省去，此时，链接将定位于当前包或当前类。

需要注意，一定要使用井号（#），而不要使用句号（.）分隔类名与方法名，或类名与变量名。Java编译器本身可以熟练地断定句点在分隔包、子包、类、内部类与方法和变量时的不同含义。但是javadoc实用程序就没有这么聪明了，因此必须对它提供帮助。

如果@see标记后面有一个<字符，就需要指定一个超链接。可以超链接到任何URL。例如：

```
@see <a href="www.horstmann.com/corejava.html">The Core Java home page</a>
```

在上述各种情况下，都可以指定一个可选的标签（label）作为锚链接（link anchor）。如果省略了label，用户看到的锚的名称就是目标代码名或URL。

如果@see标记后面有一个双引号（"）字符，文本就会显示在“see also”部分。例如，

```
@see "Core Java 2 volume 2"
```

可以为一个特性添加多个@see标记，但必须将它们放在一起。

- 如果愿意的话，还可以在注释中的任何位置放置指向其他类或方法的超级链接，以及插入一个专用的标记，例如，

```
{@link package.class#feature label}
```

这里的特性描述规则与@see标记规则一样。

4.9.6 包与概述注释

可以直接将类、方法和变量的注释放置在Java源文件中，只要用/**...*/文档注释界定就可以了。但是，要想产生包注释，就需要在每一个包目录中添加一个单独的文件。可以有如下两个选择：

1) 提供一个以package.html命名的HTML文件。在标记<BODY>...</BODY>之间的所有文本都会被抽取出来。

2) 提供一个以package-info.java命名的Java文件。这个文件必须包含一个初始的以/**和*/界定的Javadoc注释，跟随在一个包语句之后。它不应该包含更多的代码或注释。

还可以为所有的源文件提供一个概述性的注释。这个注释将被放置在一个名为overview.html

的文件中，这个文件位于包含所有源文件的父目录中。标记<BODY>... </BODY>之间的所有文本将被抽取出来。当用户从导航栏中选择“Overview”时，就会显示出这些注释内容。

4.9.7 注释的抽取

这里，假设HTML文件将被存放在目录docDirectory下。执行以下步骤：

1) 切换到包含想要生成文档的源文件目录。如果有嵌套的包要生成文档，例如com.horstmann.corejava，就必须切换到包含子目录com的目录（如果存在overview.html文件的话，这也是它的所在目录）。

2) 如果是一个包，应该运行命令：

```
javadoc -d docDirectory nameOfPackage
```

或对于多个包生成文档，运行：

```
javadoc -d docDirectory nameOfPackage1 nameOfPackage2...
```

如果文件在默认包中，就应该运行：

```
javadoc -d docDirectory *.java
```

如果省略了-d docDirectory选项，那HTML文件就会被提取到当前目录下。这样有可能会带来混乱，因此不提倡这种做法。

可以使用多种形式的命令行选项对javadoc程序进行调整。例如，可以使用-author和-version选项在文档中包含@author和@version标记（默认情况下，这些标记会被省略）。另一个很有用的选项是-link，用来为标准类添加超链接。例如，如果使用命令

```
javadoc -link http://java.sun.com/javase/6/docs/api *.java
```

那么，所有的标准类库类都会自动地链接到Sun网站的文档。

如果使用-linksource选项，则每个源文件被转换为HTML（没有颜色编码，但包含行编号），并且每个类和方法名将转变为指向源代码的超链接。

有关其他的选项，请查阅javadoc实用程序的联机文档，<http://java.sun.com/javase/javadoc>。



注释：如果需要进一步的定制，例如，生成非HTML格式的文档，可以提供自定义的doclet，以便生成想要的任何输出形式。显然，这是一种特殊的需求，有关细节内容请查阅<http://java.sun.com/j2se/javadoc>的联机文档。



提示：DocCheck是一个很有用的doclet，在<http://java.sun.com/j2se/javadoc/doccheck/>上。它可以为遗漏的文档注释搜索一组源程序文件。

4.10 类设计技巧

在结束本章之前，简单地介绍几点技巧。应用这些技巧可以使得设计出来的类更具有OOP的专业水准。

1) 一定将数据设计为私有。

最重要的是：绝对不要破坏封装性。有时候，需要编写一个访问器方法或更改器方法，但是最好还是保持实例域的私有性。很多惨痛的经验告诉我们，数据的表示形式很可能会改变，

但它们的使用方式却不会经常发生变化。当数据保持私有时，它们的表示形式的变化不会对类的使用者产生影响，即使出现bug也易于检测。

2) 一定要对数据初始化。

Java不对局部变量进行初始化，但是会对对象的实例域进行初始化。最好不要依赖于系统的默认值，而是应该显式地初始化所有的数据，具体的初始化方式可以是提供默认值，也可以是在所有构造器中设置默认值。

3) 不要在类中使用过多的基本数据类型。

就是说，用其他的类代替多个相关的基本数据类型的使用。这样会使类更加易于理解且易于修改。例如，用一个称为Address的新的类替换下面的Customer类中的实例域：

```
private String street;  
private String city;  
private String state;  
private int zip;
```

这样，可以很容易地顺应地址的变化，例如，需要增加对国际地址的处理。

4) 不是所有的域都需要独立的域访问器和域更改器。

或许，需要获得或设置雇员的薪金。而一旦构造了雇员对象，就应该禁止更改雇用日期，并且在对象中，常常包含一些不希望别人获得或设置的实例域，例如，在Address类中，存放州缩写的数组。

5) 使用标准格式进行类的定义。

一定采用下面的顺序书写类的内容：

公有访问特性部分

包作用域访问特性部分

私有访问特性部分

在每一部分中，应该按照下列顺序列出：

实例方法

静态方法

实例域

静态域

毕竟，类的使用者对公有接口要比对私有的实现细节更感兴趣，并且对方法要比对数据更感兴趣。

但是，哪一种风格更好并没有达成共识。Sun的程序设计风格建议 Java程序设计语言先书写域，后书写方法。无论采用哪种风格，重要的一点是要保持一致。

6) 将职责过多的类进行分解。

这样说似乎有点含糊不清，究竟多少算是“过多”？每个人的看法不同。但是，如果明显地可以将一个复杂的类分解成两个更为简单的类，就应该将其分解（但另一方面，也不要走极端。设计10个类，每个类只有一个方法，显然也太小了）。

下面是一个反面的设计示例。

```
public class CardDeck // bad design
```

```
{
    public CardDeck() { . . . }
    public void shuffle() { . . . }
    public int getTopValue() { . . . }
    public int getTopSuit() { . . . }
    public void draw() { . . . }

    private int[] value;
    private int[] suit;
}
```

实际上，这个类实现了两个独立的概念：一副牌（含有shuffle方法和draw方法）和一张牌（含有查看面值和花色的方法）。另外，引入一个表示单张牌的Card类。现在有两个类，每个类完成自己的职责：

```
public class CardDeck
{
    public CardDeck() { . . . }
    public void shuffle() { . . . }
    public Card getTop() { . . . }
    public void draw() { . . . }

    private Card[] cards;
}

public class Card
{
    public Card(int aValue, int aSuit) { . . . }
    public int getValue() { . . . }
    public int getSuit() { . . . }

    private int value;
    private int suit;
}
```

7) 类名和方法名要能够体现它们的职责。

与变量应该有一个能够反映其含义的名字一样，类也应该如此（在标准类库中，也存在着一些含义不明确的例子，如：Date类实际上是一个用于描述时间的类）。

命名类名的良好习惯是采用一个名词（Order）、前面有形容词修饰的名词（RushOrder）或动名词（有“-ing”后缀）修饰名词（例如，BillingAddress）。对于方法来说，习惯是访问器方法用小写get开头（getSalary），更改器方法用小写的set开头（setSalary）。

本章介绍了Java这种面向对象语言的有关对象和类的基础知识。为了真正做到面向对象，程序设计语言还必须支持继承和多态。Java提供了对这些特性的支持，具体内容将在下一章中介绍。