

第4章 对象与类

- ▲ 面向对象程序设计概述
- ▲ 使用预定义类
- ▲ 自定义类
- ▲ 静态字段与静态方法
- ▲ 方法参数
- ▲ 对象构造
- ▲ 记录
- ▲ 包
- ▲ JAR 文件
- ▲ 文档注释
- ▲ 类设计技巧

这一章将主要介绍如下内容：

- 面向对象程序设计入门；
- 如何创建标准 Java 类库中类的对象；
- 如何编写自己的类。

如果你没有面向对象程序设计背景，那么一定要认真地阅读本章的内容。面向对象程序设计与面向过程的语言在思维方式上存在着很大的差别。改变思维方式并不是一件很容易的事情，但是为了继续学习 Java，一定要熟悉对象的概念。

对于有经验的 C++ 程序员来说，与上一章一样，对本章的内容不会感到太陌生，但这两种语言还是存在着很多不同之处，所以要认真阅读本章的后半部分内容，你将发现“C++ 注释”对于你转换思维方式会很有帮助。

4.1 面向对象程序设计概述

面向对象程序设计（Object-Oriented Programming, OOP）是当今的主流程序设计范型，它取代了 20 世纪 70 年代的“结构化”或过程式编程技术。由于 Java 是面向对象的，所以你必须熟悉 OOP 才能够很好地使用 Java。

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和隐藏的实现。程序中的很多对象是来自标准类库的“成品”，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，这完全取决于开发项目的预算和时间。但是，从根本上说，只要对象能够满足要求，就不必关心其功能到底是如何实现的。

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。一旦确定了这些过程，下一步往往要考虑存储数据的适当方式。这就是 Pascal 语言的设计者 Niklaus Wirth 将其著作命名为《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 的这个书名中，算法是第一位的，数据结构排

在第二位，这也反映了当时程序员的工作方式。首先，他们会确定操作数据的过程，然后再决定如何组织数据的结构，以便于操作数据。而 OOP 却调换了这个次序，将数据放在第一位，然后再考虑操作数据的算法。

对于一些规模较小的问题，将其分解为过程的做法是合适的，而对象更适合解决规模较大的问题。考虑一个简单的 Web 浏览器，实现这个浏览器可能需要大约 2000 个过程，这些过程需要对一组全局数据进行操作。采用面向对象风格时，可能需要大约 100 个类，每个类平均包含 20 个方法（如图 4-1 所示）。这种结构更容易程序员掌握，也更容易查找 bug。假设一个特定对象的数据出错了，在访问这个数据项的 20 个方法中查找“罪魁祸首”要比在 2000 个过程中查找容易得多。

4.1.1 类

类（class）指定了如何构造对象。可以将类想象成制作小甜饼的模具，将对象想象为小甜饼。由一个类构造（construct）对象的过程称为创建这个类的一个实例（instance）。

正如前面所看到的，用 Java 编写的所有代码都在某个类中。标准 Java 库提供了几千个类，可用于各种目的，如用户界面设计、日期和日历，以及网络编程。尽管如此，在 Java 中你还需要创建一些自己的类，来描述你的应用相应问题领域中的对象。

封装（encapsulation，有时称为信息隐藏）是处理对象的一个重要概念。从形式上看，封装就是将数据和行为组合在一个包中，并对对象的使用者隐藏具体的实现细节。对象中的数据称为实例字段（instance field），操作数据的过程称为方法（method）。作为一个类的实例，一个特定对象有一组特定的实例字段值。这些值的集合就是这个对象的当前状态（state）。只要在对象上调用一个方法，它的状态就有可能发生改变。

实现封装的关键在于，绝对不能让其他类中的方法直接访问这个类的实例字段。程序只能通过对象的方法与对象数据进行交互。封装为对象赋予了“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完全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也不用关心这个类所发生的变化。

OOP 的另一个原则会让用户自定义 Java 类变得更为容易，这就是：可以通过扩展其他类来构建新类。事实上，Java 提供了一个“神通广大的超类”，名为 Object。所有其他类都扩展自这个 Object 类。在下一章中，你会了解更多有关 Object 类的内容。

扩展一个已有的类时，这个新类具有被扩展的那个类的全部属性和方法。你只需要在新类中提供适用于这个新类的新方法和实例字段。通过扩展一个类来得到另外一个类的概念称

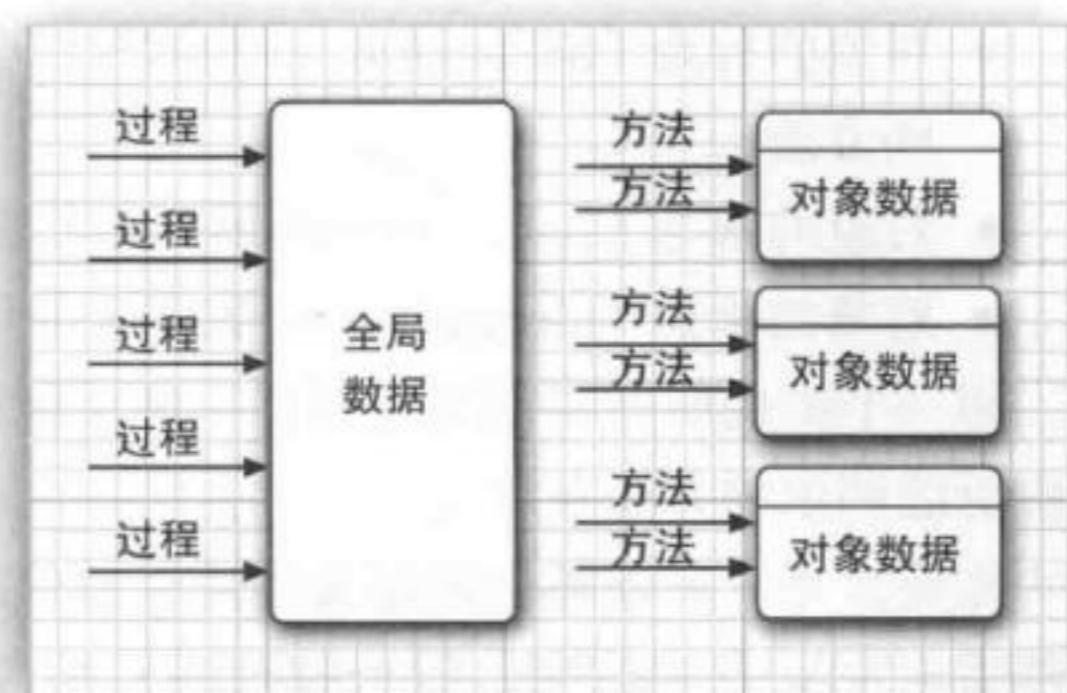


图 4-1 过程式程序设计与面向对象程序设计

为继承 (inheritance)，有关继承的详细内容请参见下一章。

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) ——可以对这个对象做哪些操作，或者可以对这个对象应用哪些方法？
- 对象的状态 (state) ——调用那些方法时，对象会如何响应？
- 对象的标识 (identity) ——如何区分可能有相同行为和状态的不同对象？

同一个类的所有实例对象都有一种家族相似性，它们都支持相同的行为。一个对象的行为由所能调用的方法来定义。

此外，每个对象都会保存着描述当前状况的信息，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，但这种改变不是自发的。对象状态的改变必然是调用方法的结果（如果不经过方法调用就可以改变对象状态，这说明破坏了封装性）。

但是，对象的状态并不能完全描述一个对象，因为每个对象都有一个唯一的标识 (identity，或称身份)。例如，在一个订单处理系统中，任何两个订单都是不同的，即使它们订购的商品完全相同。需要注意，作为同一个类的实例，每个对象的标识总是不同的，状态也通常有所不同。

对象的这些关键特性会彼此相互影响。例如，对象的状态会影响它的行为（如果一个订单“已发货”或“已付款”，就应该拒绝要求增删商品的方法调用。反过来，如果订单是“空的”，即还没有订购任何商品，就不应该允许“发货”）。

4.1.3 识别类

传统的程式程序中，必须从最上面的 `main` 函数开始编写程序。设计一个面向对象系统时，则没有所谓的“最上面”。因此，学习 OOP 的初学者常常会感觉无从下手。答案是：首先从识别类开始，然后再为各个类添加方法。

识别类的一个简单经验是在分析问题的过程中寻找名词，而方法对应动词。

例如，在订单处理系统中，有这样一些名词：

- 商品 (Item);
- 订单 (Order);
- 发货地址 (Shipping address);
- 付款 (Payment);
- 账户 (Account)。

从这些名词就可以得到类 `Item`、`Order` 等。

接下来查找动词。商品要添加 (add) 到订单中，订单可以发货 (ship) 或取消 (cancel)，另外可以对订单完成付款 (apply)。对于每一个动词，如“添加”“发货”“取消”或者“完成付款”，要识别出负责完成相应动作的对象。例如，当一个新商品添加到订单中时，订单对

象就是负责的对象，因为它知道如何存储商品以及如何对商品进行排序。也就是说，`add` 应该是 `Order` 类的一个方法，它接受一个 `Item` 对象作为参数。

当然，这种“名词与动词”原则只是一种经验，在创建类的时候，只有经验能帮助你确定哪些名词和动词重要。

4.1.4 类之间的关系

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，是一种最明显的也最一般的关系。例如，`Order` 类使用了 `Account` 类，因为 `Order` 对象需要访问 `Account` 对象来查看信用状态。但是 `Item` 类不依赖于 `Account` 类，因为 `Item` 对象不需要考虑客户账户。因此，如果一个类的方法要使用或操作另一个类的对象，我们就说前一个类依赖于后一个类。

应当尽可能减少相互依赖的类。这里的关键是，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 的改变不会在 A 中引入 bug）。用软件工程的术语来说，就是要尽可能减少类之间的耦合 (coupling)。

聚合 (aggregation)，即“has-a”关系，很容易理解，因为这种关系很具体。例如，一个 `Order` 对象包含一些 `Item` 对象。包含关系意味着类 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，而更喜欢使用更一般的“关联”关系。从建模的角度看，这是可以理解的。但对于程序员来说，“has-a”关系更加形象。我喜欢使用聚合还有另一个原因：关联的标准记法不是很清楚，请参见表 4-1。

表 4-1 表达类关系的 UML 记法

关系	UML 连接符
继承	→
接口实现	→
依赖	→
聚合	◇
关联	—
直接关联	→

继承 (inheritance)，即“is-a”关系，表示一个更特殊的类与一个更一般的类之间的关系。例如，`RushOrder` 类继承了 `Order` 类。在特殊化的 `RushOrder` 类中包含一些用于优先处理的特殊方法，还提供了一个计算运费的不同方法；而其他的方法，如添加商品、生成账单等都是从 `Order` 类继承来的。一般而言，如果类 D 扩展了类 C，类 D 会继承类 C 的方法，另外还会

有一些额外的功能（下一章将详细讨论这个重要的概念）。

很多程序员采用 UML (Unified Modeling Language, 统一建模语言) 绘制类图，来描述类之间的关系。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。

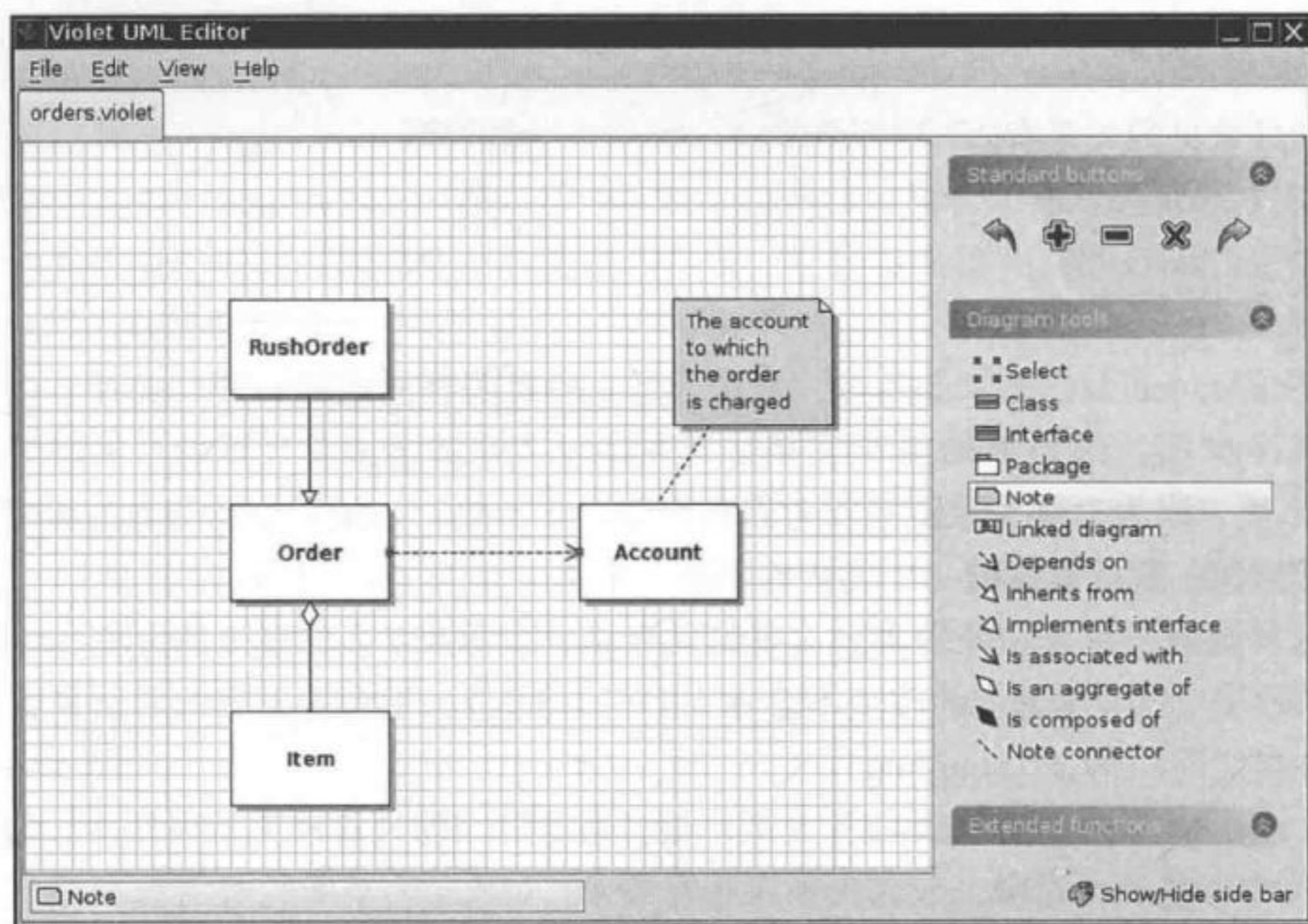


图 4-2 类图

4.2 使用预定义类

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，并不是所有的类都表现出面向对象的典型特征。以 `Math` 类为例。你已经看到，可以直接使用 `Math` 类的方法，如 `Math.random`，而不必了解它具体是如何实现的，你只需要知道方法名和参数（如果有的话）。这正是封装的关键所在，当然所有类都是这样。但 `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，因此也不必考虑创建对象和初始化它们的实例字段，因为根本没有实例字段！

下一节将会介绍一个更典型的类——`Date` 类，从中可以了解如何构造对象，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，首先必须构造对象，并指定其初始状态。然后对对象应用方法。

在 Java 程序设计语言中，要使用构造器（constructor，或称构造函数）构造新实例。构造器是一种特殊的方法，其作用是构造并初始化对象。下面来看一个例子。标准 Java 库中包含

一个 Date 类。它的对象可以描述一个时间点，例如，“December 31, 1999, 23:59:59 GMT”。

注释：你可能会感到奇怪：为什么用类表示日期，而不是像其他语言中那样用一个内置（built-in）类型来表示？例如，Visual Basic 中有一个内置的 date 类型，程序员可以采用 #12/31/1999# 格式指定日期。看起来这似乎很方便，程序员只需要使用内置的 date 类型，而不用考虑类。但实际上，Visual Basic 这样设计合适吗？在有些地区，日期表示为月/日/年，而另外一些地区则表示为日/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。如果类设计得不完善，那么其他程序员可以很容易地编写自己的类，改进或替代（replace）这些系统类（作为印证：Java 的日期类库开始时有些混乱，现在已经重新设计了两次）。

构造器总是与类同名。因此，Date 类的构造器就名为 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()
```

这个表达式会构造一个新对象。这个对象初始化为当前的日期和时间。

如果需要的话，可以将这个对象传递给一个方法：

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

或者，可以对刚构造的对象应用一个方法。Date 类中有一个 `toString` 方法。这个方法将生成日期的一个字符串描述。可以如下对新构造的 Date 对象应用 `toString` 方法：

```
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，你可能希望保留所构造的对象从而能继续使用，为此，需要将对象存放在一个变量中：

```
Date rightNow = new Date();
```

图 4-3 显示了对象变量 `rightNow`，它引用了新构造的对象。

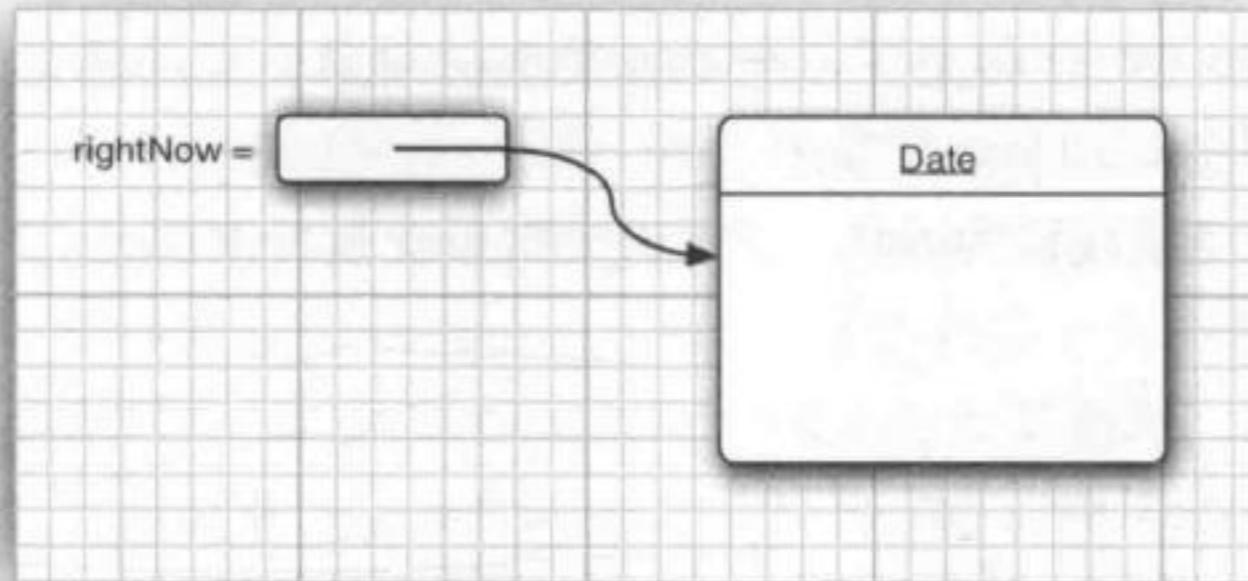


图 4-3 创建一个新对象

对象与对象变量之间存在着一个重要的区别。例如，以下语句

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

定义了一个对象变量 `startTime`，它可以引用 `Date` 类型的对象。但是，一定要认识到：变量 `startTime` 不是一个对象，而且实际上它甚至还没有引用任何对象。此时不能在这个变量上使用任何 `Date` 方法。下面的语句

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

将产生编译错误。

必须首先初始化 `startTime` 变量，这里有两个选择。当然，可以初始化这个变量，让它引用一个新构造的对象：

```
startTime = new Date();
```

也可以设置这个变量，让它引用一个已有的对象：

```
startTime = rightNow;
```

现在，这两个变量都引用同一个对象（如图 4-4 所示）。

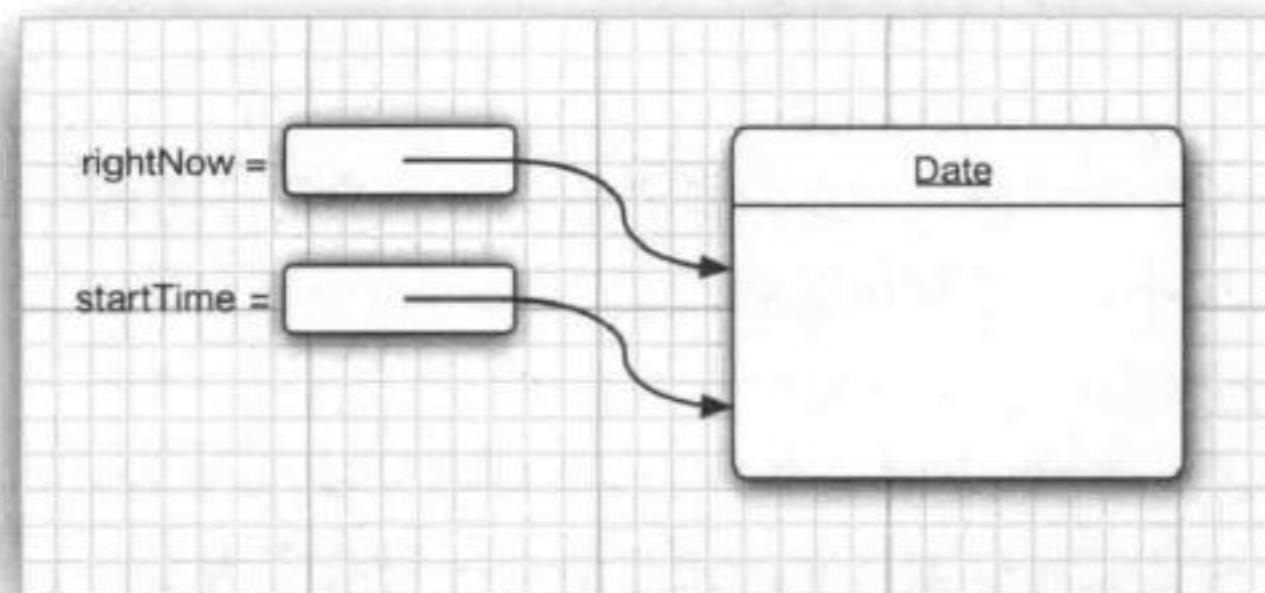


图 4-4 对象变量引用同一个对象

要认识到重要的一点：对象变量并不实际包含一个对象，它只是引用一个对象。

在 Java 中，任何对象变量的值都是一个引用，指向存储在另外一个地方的某个对象。`new` 操作符的返回值也是一个引用。下面的语句：

```
Date startTime = new Date();
```

有两个部分。表达式 `new Date()` 构造了一个 `Date` 类型的对象，它的值是新创建对象的一个引用。再将这个引用存储在 `startTime` 变量中。

可以显式地将对象变量设置为 `null`，指示这个对象变量目前没有引用任何对象。

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

我们将在 4.3.6 节更详细地讨论 `null`。

C++ 注释：很多人错误地认为 Java 中的对象变量就相当于 C++ 中的引用。然而，C++ 中没有 `null` 引用，而且引用不能赋值。应当把 Java 中的对象变量看作类似于 C++ 的对象指针。例如，

```
Date rightNow; // Java
```

实际上等同于

```
Date* rightNow; // C++
```

一旦建立了这种关联，一切就清楚了。当然，只有使用了 new 调用后 Date* 指针才会初始化。就这一点而言，C++ 与 Java 的语法几乎是一样的。

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

如果把一个变量复制到另一个变量，两个变量就指向同一个日期，即它们是同一个对象的指针。Java 中的 null 引用对应于 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，它只是包含另一个堆对象的指针。

在 C++ 中，指针十分令人头疼，因为它们很容易出错。稍不小心就会创建一个错误的指针，或者使内存管理出问题。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，那么运行时系统将会产生一个运行时错误，而不是生成随机的结果。另外，你不必担心内存管理问题，垃圾回收器会处理相关的事宜。

C++ 确实做了很大的努力，它通过支持复制构造器和赋值运算符来实现对象的自动复制。例如，一个链表（linked list）的副本是一个新链表，其内容与原始链表相同，但是有一组独立的链接。这样一来就可以适当地设计类，使它们与内置类型有相同的复制行为。在 Java 中，必须使用 clone 方法获得一个对象的完整副本。

4.2.2 Java 类库中的 LocalDate 类

在前面的例子中，我们使用了 Java 标准类库中的 Date 类。Date 类的实例有一个状态，也就是一个特定的时间点。

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，这个时间点就是所谓的纪元（epoch），它是 UTC 时间 1970 年 1 月 1 日 00:00:00。UTC 就是 Coordinated Universal Time（国际协调时间），与大家熟悉的 GMT（即 Greenwich Mean Time，格林尼治时间）一样，是一种实用的科学标准时间。

但是，Date 类对于处理人类记录日期的日历信息并不是很有用，如“December 31, 1999”。这个特定的日期描述遵循 Gregorian 阳历，这是世界上大多数国家使用的日历。但是，同样的这个时间点采用中国或希伯来的阴历来描述会大不相同，倘若我们有来自火星的顾客，基于他们使用的火星历来描述这个时间点就更不一样了。

注释：有史以来，人类的文明与历法的设计息息相关，日历要为日期指定名字，指定太阳和月亮的周期次序。要了解有关世界上各种日历的有趣解释，从法国大革命的日历到玛雅人计算日期的方法，请参见 Nachum Dershowitz 和 Edward M. Reingold 编写的《Calendrical Calculations》第 4 版（剑桥大学出版社，2018 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；另一个是用大家熟悉的日历表示法表示日期的 `LocalDate` 类。Java 8 引入了另外一些类来处理日期和时间的不同方面——有关内容参见卷 II 第 6 章。

将时间度量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。

不要使用构造器来构造 `LocalDate` 类的对象。实际上，应当使用静态工厂方法（factory method），它会代表你调用构造器。下面的表达式：

```
LocalDate.now()
```

会构造一个新对象，表示构造这个对象时的日期。

可以提供年、月和日来构造对应一个特定日期的对象：

```
LocalDate.of(1999, 12, 31)
```

当然，通常我们都希望将构造的对象保存在一个对象变量中：

```
LocalDate newYearsEve = LocalDate.of(1999, 12, 31);
```

一旦有了一个 `LocalDate` 对象，可以用方法 `getYear`、`getMonthValue` 和 `getDayOfMonth` 得到年、月和日：

```
int year = newYearsEve.getYear(); // 1999
int month = newYearsEve.getMonthValue(); // 12
int day = newYearsEve.getDayOfMonth(); // 31
```

看起来这似乎没有多大的意义，因为这正是构造对象时使用的那些值。不过，有时可能有一个计算得到的日期，然后你希望调用这些方法来了解它的更多信息。例如，`plusDays` 方法会生成一个新的 `LocalDate`，如果把应用这个方法的对象称为当前对象，那么这个新日期对象则是距当前对象指定天数的一个新日期：

```
LocalDate aThousandDaysLater = newYearsEve.plusDays(1000);
year = aThousandDaysLater.getYear(); // 2002
month = aThousandDaysLater.getMonthValue(); // 09
day = aThousandDaysLater.getDayOfMonth(); // 26
```

`LocalDate` 类封装了一些实例字段来维护所设置的日期。如果不查看源代码，就不可能知道类内部的日期表示。当然，封装的意义就在于内部表示并不重要，重要的是类对外提供的方法。

注释：实际上，`Date` 类也有得到日、月、年的方法，分别是 `getDay`、`getMonth` 以及 `getYear`，不过这些方法已经废弃。当类库设计者意识到某个方法最初就不该引入时，就把它标记为废弃，不鼓励使用。

类库设计者意识到应当单独提供类来处理日历，不过在此之前这些方法已经是 `Date` 类的一部分了。Java 1.1 中引入较早的一组日历类时，`Date` 方法被标记为废弃。虽然仍然可以在程序中使用这些方法，不过如果这样做，编译时会出现警告。最好不要使用废弃的方法，因为将来的某个类库版本很有可能会将它们完全删除。

 提示：JDK 提供了 jdeprscan 工具来检查你的代码中是否使用了 Java API 已经废弃的特性。有关说明参见 <https://docs.oracle.com/en/java/javase/17/docs/specs/man/jdeprscan.html>。

4.2.3 更改器方法与访问器方法

再来看上一节中的 `plusDays` 方法调用：

```
LocalDate aThousandDaysLater = newYearsEve.plusDays(1000);
```

这个调用之后 `newYearsEve` 会有什么变化？它会改为 1000 天之后的日期吗？事实上，并没有。`plusDays` 方法会生成一个新的 `LocalDate` 对象，然后把这个新对象赋给 `aThousandDaysLater` 变量。原来的对象不做任何改动。我们说 `plusDays` 方法没有更改（mutate）调用这个方法的对象。（这类似于第 3 章中见过的 `String` 类的 `toUpperCase` 方法。在一个字符串上调用 `toUpperCase` 时，这个字符串仍保持不变，并返回一个包含大写字符的新字符串。）

Java 库的一个较早版本曾经有另一个处理日历的类，名为 `GregorianCalendar`。可以如下为这个类表示的一个日期增加 1000 天：

```
GregorianCalendar someDay = new GregorianCalendar(1999, 11, 31);
// odd feature of that class: month numbers go from 0 to 11
someDay.add(Calendar.DAY_OF_MONTH, 1000);
```

与 `LocalDate.plusDays` 方法不同，`GregorianCalendar.add` 方法是一个更改器方法（mutator method）。调用这个方法后，`someDay` 对象的状态会改变。可以如下查看新状态：

```
year = someDay.get(Calendar.YEAR); // 2002
month = someDay.get(Calendar.MONTH) + 1; // 09
day = someDay.get(Calendar.DAY_OF_MONTH); // 26
```

正是因为这个原因，我们将变量命名为 `someDay` 而不是 `newYearsEve`——调用这个更改器方法之后，它不再是新年前夜。

相反，只访问对象而不修改对象的方法有时称为访问器方法（accessor method）。例如，`LocalDate.getYear` 和 `GregorianCalendar.get` 就是访问器方法。

 C++ 注释：在 C++ 中，带有 `const` 后缀的方法是访问器方法；没有声明为 `const` 的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。

下面用一个具体应用 `LocalDate` 类的程序来结束这一节。这个程序将显示当前月的日历，格式如下：

Mon	Tue	Wed	Thu	Fri	Sat	Sun
					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						

当前日期标记有一个 * 号。可以看到，这个程序需要知道如何计算某月份的天数以及一个给定日期是星期几。

下面来看这个程序的关键步骤。首先构造一个对象，并用当前的日期初始化。

```
LocalDate date = LocalDate.now();
```

下面获得当前的月份和日期。

```
int month = date.getMonthValue();
int today = date.getDayOfMonth();
```

然后，将 date 设置为这个月的第一天，并得到这一天为星期几。

```
date = date.minusDays(today - 1); // set to start of month
DayOfWeek weekday = date.getDayOfWeek();
int value = weekday.getValue(); // 1 = Monday, . . . , 7 = Sunday
```

变量 weekday 设置为 DayOfWeek 类型的对象。我们调用这个对象的 getValue 方法来得到对应星期几的一个数值。这会得到一个整数，这里遵循国际惯例，即周末是一周的结束，星期一就返回 1，星期二返回 2，依此类推。星期日则返回 7。

注意，日历的第一行是缩进的，使当月第一天对应正确的星期几。下面的代码会打印表头和第一行的缩进：

```
System.out.println("Mon Tue Wed Thu Fri Sat Sun");
for (int i = 1; i < value; i++)
    System.out.print("    ");
```

现在我们来打印日历的主体。进入一个循环，其中 date 遍历一个月中的每一天。

每次迭代中，我们要打印日期值。如果 date 是当前日期，这个日期则用一个 * 标记。接下来，把 date 推进到下一天。如果到达新的一周的第一天，则换行打印：

```
while (date.getMonthValue() == month)
{
    System.out.printf("%3d", date.getDayOfMonth());
    if (date.getDayOfMonth() == today)
        System.out.print("*");
    else
        System.out.print(" ");
    date = date.plusDays(1);
    if (date.getDayOfWeek().getValue() == 1) System.out.println();
}
```

什么时候结束呢？我们不知道这个月有几天，是 31 天、30 天、29 天还是 28 天？实际上，只要 date 还在当月就要继续迭代。

程序清单 4-1 给出了完整的程序。

可以看到，利用 LocalDate 类可以编写一个日历程序，它能处理星期几以及各月天数不同等复杂问题。你并不需要知道 LocalDate 类如何计算月和星期几，只需要使用这个类的接口，也就是诸如 plusDays 和 getDayOfWeek 等方法。

这个示例程序的重点是向你展示如何使用一个类的接口来完成相当复杂的任务，而无须了解实现细节。

程序清单 4-1 CalendarTest/CalendarTest.java

```

1 import java.time.*;
2
3 /**
4 * @version 1.5 2015-05-08
5 * @author Cay Horstmann
6 */
7 public class CalendarTest
8 {
9     public static void main(String[] args)
10    {
11        LocalDate date = LocalDate.now();
12        int month = date.getMonthValue();
13        int today = date.getDayOfMonth();
14
15        date = date.minusDays(today - 1); // set to start of month
16        DayOfWeek weekday = date.getDayOfWeek();
17        int value = weekday.getValue(); // 1 = Monday, . . . , 7 = Sunday
18
19        System.out.println("Mon Tue Wed Thu Fri Sat Sun");
20        for (int i = 1; i < value; i++)
21            System.out.print("   ");
22        while (date.getMonthValue() == month)
23        {
24            System.out.printf("%3d", date.getDayOfMonth());
25            if (date.getDayOfMonth() == today)
26                System.out.print("*");
27            else
28                System.out.print(" ");
29            date = date.plusDays(1);
30            if (date.getDayOfWeek().getValue() == 1) System.out.println();
31        }
32        if (date.getDayOfWeek().getValue() != 1) System.out.println();
33    }
34 }

```

API java.time.LocalDate 8

- static LocalDate now()

构造一个表示当前日期的对象。

- static LocalDate of(int year, int month, int day)

构造一个表示给定日期的对象。

- int getYear()

- int getMonthValue()

- int getDayOfMonth()

得到当前日期的年、月和日。

- DayOfWeek getDayOfWeek()

得到当前日期是星期几，作为 DayOfWeek 类的一个实例返回。在 DayOfWeek 实例上调用

`getValue` 来得到 1 ~ 7 之间的一个数，表示这是星期几，1 表示星期一，7 表示星期日。

- `LocalDate plusDays(int n)`
- `LocalDate minusDays(int n)`

生成当前日期之后或之前 n 天的日期。

4.3 自定义类

在第 3 章中，我们已经开始编写一些简单的类。但是，那些类都只包含一个简单的 `main` 方法。现在来学习如何编写更复杂的应用所需要的那种主力类（workhorse class）。通常，这些类没有 `main` 方法，而有自己的实例字段和实例方法。要想构建一个完整的程序，会结合使用多个类，其中只有一个类有 `main` 方法。

4.3.1 Employee 类

在 Java 中，最简单的类定义形式为：

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

下面看一个非常简单的 `Employee` 类，编写工资管理系统时可能会用到：

```
class Employee
{
    // instance fields
    private String name;
    private double salary;
    private LocalDate hireDay;

    // constructor
    public Employee(String n, double s, int year, int month, int day)
    {
        name = n;
        salary = s;
        hireDay = LocalDate.of(year, month, day);
    }

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

```

// more methods
...
}

```

这里将这个类的实现分成以下几个部分，并分别在稍后的几节中介绍。不过，首先来看程序清单 4-2，这个程序展示了 Employee 类的实际使用。

程序清单 4-2 EmployeeTest/EmployeeTest.java

```

1 import java.time.*;
2
3 /**
4  * This program tests the Employee class.
5  * @version 1.13 2018-04-10
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     private String name;
33     private double salary;
34     private LocalDate hireDay;
35
36     public Employee(String n, double s, int year, int month, int day)
37     {
38         name = n;
39         salary = s;
40         hireDay = LocalDate.of(year, month, day);
41     }
42
43     public String getName()
44     {
45         return name;
46     }
47

```

```

48     public double getSalary()
49     {
50         return salary;
51     }
52
53     public LocalDate getHireDay()
54     {
55         return hireDay;
56     }
57
58     public void raiseSalary(double byPercent)
59     {
60         double raise = salary * byPercent / 100;
61         salary += raise;
62     }
63 }
```

在这个程序中，我们构造了一个 Employee 数组，并填入了 3 个 Employee 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，使用 Employee 类的 raiseSalary 方法将每个员工的薪水提高 5%：

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

最后，调用 getName 方法、getSalary 方法和 getHireDay 方法打印各个员工的信息：

```
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：Employee 类和带有 public 访问修饰符的 EmployeeTest 类。EmployeeTest 类包含 main 方法，其中使用了前面介绍的代码。

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

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

然后启动这个程序，为字节码解释器提供程序中包含 main 方法的那个类的类名：

```
java EmployeeTest
```

字节码解释器开始运行 EmployeeTest 类的 main 方法中的代码。这个代码会先后构造 3 个新 Employee 对象，并显示它们的状态。

4.3.2 使用多个源文件

在程序清单 4-2 中，一个源文件包含了两个类。许多程序员习惯将各个类放在一个单独

的源文件中。例如，将 Employee 类存放在文件 Employee.java 中，而将 EmployeeTest 类存放在文件 EmployeeTest.java 中。

如果喜欢这样组织文件，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
```

这样一来，所有与通配符匹配的源文件都将被编译成类文件。或者可以简单地键入以下命令：

```
javac EmployeeTest.java
```

你可能会感到惊讶，使用第二种方式时并没有显式地编译 Employee.java。不过，当 Java 编译器发现 EmployeeTest.java 中使用了 Employee 类时，它会查找名为 Employee.class 的文件。如果没有找到这个类文件，就会自动搜索 Employee.java 并编译这个文件。另外，如果 Employee.java 的版本较已有的 Employee.class 文件版本更新，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 中的相应工具，如 nmake），那么可以认为 Java 编译器内置了 make 功能。

4.3.3 剖析 Employee 类

下面各小节将对 Employee 类进行剖析。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 LocalDate 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，将在本章和下一章中介绍）。

接下来，需要注意在 Employee 类的实例中有 3 个实例字段，用来存放将要操作的数据：

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

关键字 private 确保只有 Employee 类本身的方法能够访问这些实例字段，任何其他类的方法都不能读写这些字段。

注释：可以用 public 标记实例字段，但这是一种很不好的做法。public 实例字段允许程序的任何部分都能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 字段，从我们的经验来看，有些代码将利用这种做法存取权限，而这是我们最不希望看到的。因此，这里强烈建议将实例字段标记为 private。

最后，请注意，有两个实例字段本身就是对象：name 字段是 String 类对象的引用，

`hireDay` 字段是 `LocalDate` 类对象的引用。类经常包含类类型的实例字段。

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;
    hireDay = LocalDate.of(year, month, day);
}
```

可以看到，构造器与类同名。构造 `Employee` 类的对象时，构造器会运行，这会将实例字段初始化为所希望的初始状态。

例如，使用下面这个代码创建 `Employee` 类的一个实例时：

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

将如下设置实例字段：

```
name = "James Bond";
salary = 100000;
hireDay = LocalDate.of(1950, 1, 1); // January 1, 1950
```

构造器与其他方法有一个重要的不同。构造器总是结合 `new` 操作符来调用。不能对一个已经存在的对象调用构造器来重新设置实例字段。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 个或多个参数。
- 构造器没有返回值。
- 构造器总是结合 `new` 操作符一起调用。

 **C++ 注释：**Java 中构造器的工作方式与 C++ 中相同。但是，要记住所有 Java 对象都是在堆中构造的，构造器总是结合 `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，不要引入与实例字段同名的局部变量。例如，下面的构造器将不会设置 `name` 或 `salary` 实例字段：

```
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，它们会遮蔽（shadow）同名的实例字段。有些程序员偶尔会不假思索地写出这类代码，因为他们的手指会不自觉地增加数据类型。这种错误很难检查出来，因此，必须注意在所有的方法中都不要使用与实例字段同名的变量。

4.3.5 用 var 声明局部变量

在 Java 10 中，如果可以从变量的初始值推导出它们的类型，那么可以用 var 关键字声明局部变量，而无须指定类型。例如，可以不这样声明：

```
Employee harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
```

只需要写为：

```
var harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
```

这一点很好，因为这样可以避免重复写类型名 Employee。

从现在开始，倘若无须了解 Java API 就能从等号右边明显看出类型，在这种情况下我们都将使用 var 表示法。不过我们不会对数值类型使用 var，如 int、long 或 double，这样你就不用当心 0、0L 和 0.0 之间的区别。对 Java API 有了更多经验后，你可能会希望更多地使用 var 关键字。

注意 var 关键字只能用于方法中的局部变量。参数和字段的类型必须声明。

4.3.6 使用 null 引用

在 4.2.1 节中我们已经了解到，对象变量包含一个对象的引用，或者包含一个特殊值 null，后者表示没有引用任何对象。

听上去这是一种处理特殊情况的便捷机制，如未知的名字或雇用日期。不过使用 null 值时要非常小心。

如果对 null 值应用一个方法，会产生一个 NullPointerException 异常。

```
LocalDate rightNow = null;
String s = rightNow.toString(); // NullPointerException
```

这是一个很严重的错误，类似于“索引越界”异常。如果你的程序没有“捕获”异常，那么程序就会终止。正常情况下，程序并不捕获这些异常，而是依赖于程序员从一开始就不需要带来异常。

 提示：程序因 NullPointerException 异常终止时，栈轨迹会显示问题出现在哪一行代码中。从 Java 17 开始，错误消息会包含有 null 值的变量或方法名。例如，在以下调用中：

```
String s = e.getHireDay().toString();
```

错误消息会告诉你 e 是否为 null 或者 getHireDay 是否返回 null。

定义一个类时，最好清楚地知道哪些字段可能为 `null`。在我们的例子中，我们不希望 `name` 或 `hireDay` 字段为 `null`。（不用担心 `salary` 字段，这个字段是基本类型，所以不可能是 `null`。）

`hireDay` 字段肯定是非 `null` 的，因为它初始化为一个新的 `LocalDate` 对象。但是 `name` 可能为 `null`，如果调用构造器时为 `n` 提供的实参是 `null`，`name` 就会是 `null`。

对此有两种解决方法。“宽容”方法是把 `null` 参数转换为一个适当的非 `null` 值：

```
if (n == null) name = "unknown"; else name = n;
```

`Objects` 类对此提供了一个便利方法：

```
public Employee(String n, double s, int year, int month, int day)
{
    name = Objects.requireNonNullElse(n, "unknown");
    ...
}
```

“严格”方法则干脆拒绝 `null` 参数：

```
public Employee(String n, double s, int year, int month, int day)
{
    name = Objects.requireNonNull(n, "The name cannot be null");
    ...
}
```

如果用 `null` 名字构造一个 `Employee` 对象，就会产生 `NullPointerException` 异常。乍看上去这种补救方法好像不太有用，不过这种方法有两个好处：

1. 异常报告会提供这个问题的描述。
2. 异常报告会准确地指出问题所在的位置，否则 `NullPointerException` 异常会出现在其他地方，而很难追踪到真正导致问题的构造器参数。

注释：如果要接受一个对象引用作为构造参数，就要问问自己：是不是真的希望接受可有可无的值。如果不是，那么“严格”方法更合适。

4.3.7 隐式参数与显式参数

方法会操作对象并访问它们的实例字段。例如，以下方法

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

将调用这个方法的对象的 `salary` 实例字段设置为一个新值。考虑下面这个调用：

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

其作用是将 `number007.salary` 字段的值增加 5%。具体地说，这个调用将执行以下指令：

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

`raiseSalary` 方法有两个参数。第一个参数称为隐式（`implicit`）参数，是出现在方法名

前的 Employee 类型的对象。第二个参数是位于方法名后面括号中的数值，这是一个显式 (explicit) 参数。(有人把隐式参数称为方法调用的目标或接收者。)

可以看到，显式参数显式地列在方法声明中，例如 double byPercent。隐式参数则没有出现在方法声明中。

在每一个方法中，关键字 this 指示隐式参数。如果愿意，可以如下改写 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字段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，那么这个方法将自动成为内联 (inline) 方法。

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

在 Java 中，所有的方法都必须在类的内部定义，但这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。即时编译器会关注那些简短、经常调用而且没有被覆盖的方法调用，并进行优化。

4.3.8 封装的优点

最后再仔细看一下非常简单的 getName 方法、getSalary 方法和 getHireDay 方法。

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

public double getSalary()
{
    return salary;
}

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

这些都是典型的访问器方法。由于它们只返回实例字段的值，因此又称为字段访问器

(field accessor)。

如果将 name、salary 和 hireDay 字段标记为公共，而不是编写单独的访问器方法，不是更容易一些吗？

不过，name 是一个只读字段，一旦在构造器中设置，就没有办法能够修改这个字段。这样我们可以确保 name 字段不会受到外界的破坏。

虽然 salary 不是只读字段，但是它只能用 raiseSalary 方法修改。具体地，如果这个值出现了错误，那么只需要调试这个 raiseSalary 方法就可以了。如果 salary 字段是公共的，破坏这个字段值的罪魁祸首有可能出没在任何地方（那就很难调试了）。

有些时候，可能想要获得或设置实例字段的值，那么你需要提供下面三项内容：

- 一个私有的实例字段；
- 一个公共的字段访问器方法；
- 一个公共的字段更改器方法。

这样做要比提供一个简单的公共实例字段复杂些，但有很多明显的好处。

首先，可以改变内部实现，而不影响该类方法之外的任何其他代码。例如，如果将存储姓名的字段改为：

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

那么 getName 方法可以改为返回

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

这个修改对于程序的其余部分是完全不可见的。

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可能需要做许多工作。这将为我们带来第二点好处：更改器方法可以完成错误检查，而只对字段赋值的代码不会费心这么做。例如，setSalary 方法可以检查工资是否小于 0。

警告：注意不要编写返回可变对象引用的访问器方法。在本书之前的一版中，我们的 Employee 类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
{
    private Date hireDay;
    ...
    public Date getHireDay()
    {
        return hireDay; // BAD
    }
    ...
}
```

LocalDate 类没有更改器方法，与之不同，Date 类有一个更改器方法 setTime，可以设置毫秒数。

Date 对象是可变的，这一点就破坏了封装性！考虑下面这段有问题的代码：

```

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 of added seniority

```

出错的原因很微妙。d 和 harry.hireDay 引用同一个对象（如图 4-5 所示）。对 d 调用更改器方法会自动改变这个 Employee 对象的私有状态！

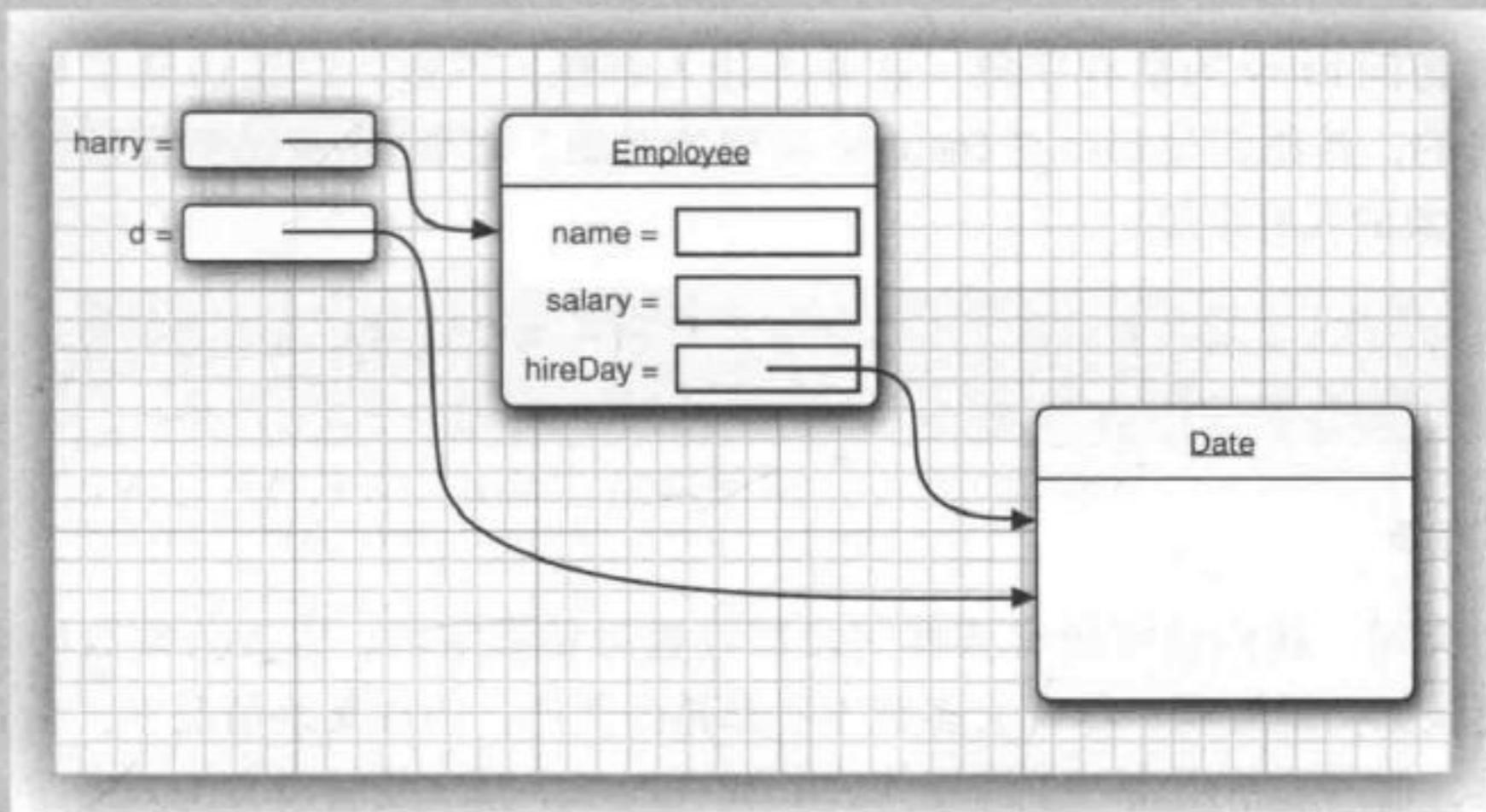


图 4-5 返回可变实例字段的引用

如果需要返回一个可变对象的引用，首先应该对它进行克隆（clone）。对象克隆是指存放在另一个新位置上的对象副本。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(); // OK
    }
    . . .
}

```

这里有一个经验，如果需要返回一个可变字段的副本，就应该使用 clone。

4.3.9 基于类的访问权限

从前面已经知道，方法可以访问调用这个方法的对象的私有数据。一个类的方法可以访问这个类的所有对象的私有数据，这令很多人感到奇怪。例如，下面来看用来比较两个员工的 equals 方法。

```
class Employee
```

```

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

```

下面是一个典型的调用：

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

这个方法访问 `harry` 的私有字段，这并不让人奇怪，不过，它还访问了 `boss` 的私有字段。这是合法的，其原因是 `boss` 是 `Employee` 类型的对象，而 `Employee` 类的方法可以访问任何 `Employee` 类型对象的私有字段。

C++ C++ 注释：C++ 也有同样的规则。方法可以访问所属类任何对象的私有特性（feature），而不仅限于隐式参数。

4.3.10 私有方法

实现一个类时，我们会将所有实例字段都设置为私有字段，因为公共数据很危险。不过，方法又应该如何设置呢？尽管大多数方法都是公共的，但在某些情况下，私有方法可能很有用。有时，你可能希望将一个计算代码分解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共接口的一部分，这是因为它们往往与当前实现关系非常紧密，或者需要一个特殊协议或调用次序。最好将这样的方法实现为私有方法。

在 Java 中，要实现一个私有方法，只需将关键字 `public` 改为 `private` 即可。

如果将一个方法设置为私有，倘若你改变了方法的具体实现，并没有义务保证这个方法依然可用。如果数据的表示发生了变化，那么这个方法可能变得更难实现，或者不再需要；这并不重要。重点在于，只要方法是私有的，类的设计者就可以确信它不会在别处使用，所以可以将其删去。如果一个方法是公共的，就不能简单地将其删除，因为可能会有其他代码依赖这个方法。

4.3.11 final 实例字段

可以将实例字段定义为 `final`。这样的字段必须在构造对象时初始化。也就是说，必须确保在每一个构造器执行之后，这个字段的值已经设置，并且以后不能再修改这个字段。例如，可以将 `Employee` 类中的 `name` 字段声明为 `final`，因为在对象构造之后，这个值不会再改变，即没有 `setName` 方法。

```

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

```

`final` 修饰符对于类型为基本类型或者不可变类的字段尤其有用。（如果类中的所有方法都不会改变其对象，这样的类就是不可变类。例如，`String` 类就是不可变的。）

对于可变类，使用 final 修饰符可能会造成混乱。例如，考虑以下字段：

```
private final StringBuilder evaluations;
```

它在 Employee 构造器中初始化为

```
evaluations = new StringBuilder();
```

final 关键字只是表示存储在 evaluations 变量中的对象引用不会再指示另一个不同的 StringBuilder 对象。不过这个对象可以更改：

```
public void giveGoldStar()
{
    evaluations.append(LocalDate.now() + ": Gold star!\n");
}
```

4.4 静态字段与静态方法

在前面给出的示例程序中，main 方法都标记了 static 修饰符。下面来讨论这个修饰符的含义。

4.4.1 静态字段

如果将一个字段定义为 static，那么这个字段并不出现在每个类的对象中。每个静态字段只有一个副本。可以认为静态字段属于类，而不属于单个对象。例如，假设需要为每一个员工分配唯一的标识码，这里为 Employee 类添加一个实例字段 id 和一个静态字段 nextId：

```
class Employee
{
    private static int nextId = 1;

    private int id;
    ...
}
```

现在，每一个 Employee 对象都有自己的 id 字段，但这个类的所有实例将共享一个 nextId 字段。换句话说，如果有 1000 个 Employee 类对象，则有 1000 个实例字段 id，每个对象有一个实例字段 id。但是，只有一个静态字段 nextId。即使没有 Employee 对象，静态字段 nextId 也存在。它属于类，而不属于任何单个对象。

注释：在一些面向对象程序设计语言中，静态字段被称为类字段。术语“静态”只是沿用了 C++ 的叫法，并无实际意义。

在构造器中，我们为新 Employee 对象分配下一个可用的 ID，然后将其增 1：

```
id = nextId;
nextId++;
```

假设我们构造了对象 harry。harry 的 id 字段设置为静态字段 nextId 的当前值，并将静态字段 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，但静态常量却很常用。例如，Math类中定义了一个静态常量：

```

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

```

在你的程序中，可以用Math.PI来访问这个常量。

如果省略关键字static，那么PI就变成了Math类的一个实例字段。也就是说，需要通过Math类的一个对象来访问PI，并且每一个Math对象都有它自己的一个PI副本。

另一个你已经多次使用的静态常量是System.out。它在System类中声明如下：

```

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

```

前面曾经多次提到过，最好不要有公共字段，因为谁都可以修改公共字段。不过，公共常量（即final字段）却没问题。因为out被声明为final，所以，不允许再将它重新赋值为另一个打印流：

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

注释：如果查看System类，就会发现有一个setOut方法可以将System.out设置为不同的流。你可能会感到奇怪，为什么这个方法可以修改final变量的值。原因在于，setOut方法是一个原生方法，而不是在Java语言中实现的。原生方法可以绕过Java语言的访问控制机制。这是一种特殊的解决方法，你自己编写程序时不要模仿这种做法。

4.4.3 静态方法

静态方法是不操作对象的方法。例如，Math类的pow方法就是一个静态方法。以下表达式：

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

会计算幂 x^a 。它并不使用任何Math对象来完成这个任务。换句话说，它没有隐式参数。

可以认为静态方法是没有this参数的方法（在一个非静态方法中，this参数指示这个方法的隐式参数，参见4.3.7节）。

Employee类的静态方法不能访问id实例字段，因为它并不操作对象。但是，静态方法可以访问静态字段。下面是这样一个静态方法的示例：

```
public static int advanceId()
{
    int r = nextId; // obtain next available id
    nextId++;
    return r;
}
```

要调用这个方法，需要提供类名：

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

这个方法可以省略关键字 `static` 吗？答案是肯定的。但是，这样一来，你就需要通过 `Employee` 类型的对象引用来调用这个方法。

注释：可以使用对象调用静态方法，这是合法的。例如，如果 `harry` 是一个 `Employee` 对象，那么可以调用 `harry.advanceId()` 而不是 `Employee.advanceId()`。不过，我发现这种写法很容易造成混淆，其原因是 `advanceId` 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。

下面两种情况可以使用静态方法：

- 方法不需要访问对象状态，因为它需要的所有参数都通过显式参数提供（例如 `Math.pow()`）。
- 方法只需要访问类的静态字段（例如 `Employee.advanceId()`）。

C++ 注释：Java 中的静态字段与静态方法在功能上与 C++ 相同。但是，语法稍有所不同。在 C++ 中，要使用 `::` 操作符访问作用域之外的静态字段或静态方法，如 `Math::PI`。

术语“静态”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“静态”是有意义的：变量一直保留，当再次进入这个块时它仍然存在。随后，`static` 在 C 中有了第二种含义，表示不能从其他文件访问的全局变量和函数。重用关键字 `static` 只是为了避免引入一个新的关键字。最后，C++ 第三次重用了这个关键字，与之前赋予的含义完全无关，它指示属于类而不属于任何特定类对象的变量和函数。这与 Java 中这个关键字的含义相同。

4.4.4 工厂方法

静态方法还有另外一种常见的用途。类似 `LocalDate` 和 `NumberFormat` 的类使用静态工厂方法（factory method）来构造对象。你已经见过工厂方法 `LocalDate.now` 和 `LocalDate.of`。可以如下得到不同样式的格式化对象：

```
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` 类不使用构造器来创建对象呢？这有两个原因：

- 无法为构造器命名。构造器的名字总是要与类名相同。但是，这里希望有两个不同的名字，分别得到货币实例和百分比实例。

- 使用构造器时，无法改变所构造对象的类型。而工厂方法实际上将返回 `DecimalFormat` 类的对象，这是继承 `NumberFormat` 的一个子类（有关继承的更多详细内容请参见第 5 章）。

4.4.5 main 方法

需要指出，可以调用静态方法而不需要任何对象。例如，不需要构造 `Math` 类的任何对象就可以调用 `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，启动程序时还没有任何对象。将执行静态 `main` 方法，并构造程序所需要的对象。

 提示：每一个类都可以有一个 `main` 方法。这是为类增加演示代码的一个技巧。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;
        hireDay = LocalDate.of(year, month, day);
    }
    .
    .
    public static void main(String[] args) // unit test
    {
        var e = new Employee("Romeo", 50000, 2003, 3, 31);
        e.raiseSalary(10);
        System.out.println(e.getName() + " " + e.getSalary());
    }
    .
}
```

要看 `Employee` 类的演示，只需要执行

```
java Employee
```

如果 `Employee` 类是一个更大的应用的一部分，那么可以使用以下命令运行这个应用：

```
java Application
```

`Employee` 类的 `main` 方法永远不会执行。

程序清单 4-3 中的程序包含了 `Employee` 类的一个简单版本，其中有一个静态字段 `nextId` 和一个静态方法 `advanceId`。这里将三个 `Employee` 对象填入一个数组，然后打印员工信息。最后，

打印下一个可用的员工标识码来展示静态方法。

程序清单 4-3 StaticTest/StaticTest.java

```
1  /**
2   * This program demonstrates static methods.
3   * @version 1.02 2008-04-10
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          var 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              System.out.println("name=" + e.getName() + ",id=" + e.getId() + ",salary="
21                  + e.getSalary());
22          }
23
24          int n = Employee.getNextId(); // calls static method
25          System.out.println("Next available id=" + n);
26      }
27  }
28
29  class Employee
30  {
31      private static int nextId = 1;
32
33      private String name;
34      private double salary;
35      private int id;
36
37      public Employee(String n, double s)
38      {
39          name = n;
40          salary = s;
41          id = advanceId();
42      }
43
44      public String getName()
45      {
46          return name;
47      }
48
49      public double getSalary()
50      {
51          return salary;
```

```

52 }
53
54 public int getId()
55 {
56     return id;
57 }
58
59 public static int advanceId()
60 {
61     int r = nextId; // obtain next available id
62     nextId++;
63     return r;
64 }
65
66 public static void main(String[] args) // unit test
67 {
68     var e = new Employee("Harry", 50000);
69     System.out.println(e.getName() + " " + e.getSalary());
70 }
71 }

```

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

`java Employee`

和

`java StaticTest`

执行两个 main 方法。

API `java.util.Objects` 7

- `static <T> void requireNonNull(T obj)`
- `static <T> void requireNonNull(T obj, String message)`
- `static <T> void requireNonNull(T obj, Supplier<String> messageSupplier)` 8

如果 obj 为 null，这些方法会抛出一个 `NullPointerException` 异常而没有任何消息，或者有给定的消息。(第 6 章会解释如何利用供应者以懒方式得到一个值。第 8 章会解释 `<T>` 语法。)

- `static <T> T requireNonNullElse(T obj, T defaultObj)` 9
- `static <T> T requireNonNullElseGet(T obj, Supplier<T> defaultSupplier)` 9

如果 obj 不为 null 则返回 obj，或者如果 obj 为 null 则返回默认对象。

4.5 方法参数

首先来回顾在程序设计语言中关于如何将参数传递到方法（或函数）的一些专业术语。按值调用（call by value）表示方法接收的是调用者提供的值。而按引用调用（call by reference）表示方法接收的是调用者提供的变量位置（location）。所以，方法可以修改按引用传递的变量的值，而不能修改按值传递的变量的值。“按……调用”（call by）是一个标准的计算机科学

术语，用来描述各种程序设计语言（不只是 Java）中方法参数的行为（事实上，以前还有一种按名调用（call by name），Algol 程序设计语言是最古老的高级语言之一，它就采用了按名调用方式。不过，这种传递方式已经成为历史）。

Java 程序设计语言总是采用按值调用。也就是说，方法会得到所有参数值的一个副本。具体来讲，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);
```

不论这个方法具体如何实现，我们知道，在这个方法调用之后，percent 的值还是 10。

下面再仔细研究一下这种情况。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);
```

不过，这并不起作用。调用这个方法之后，percent 的值还是 10。下面来看发生了什么：

1. x 初始化为 percent 值的一个副本（也就是 10）。
2. x 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。

不过，有两种不同类型的方法参数：

- 基本数据类型（数字、布尔值）。
- 对象引用。

你已经看到，一个方法不可能修改基本数据类型的参数，而对象参数就不同了，可以很容易地实现一个方法将一个员工的工资增至 3 倍：

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

如下调用这个方法时，

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

具体的执行过程为：

1. x 初始化为 harry 值的一个副本，这里就是一个对象引用。

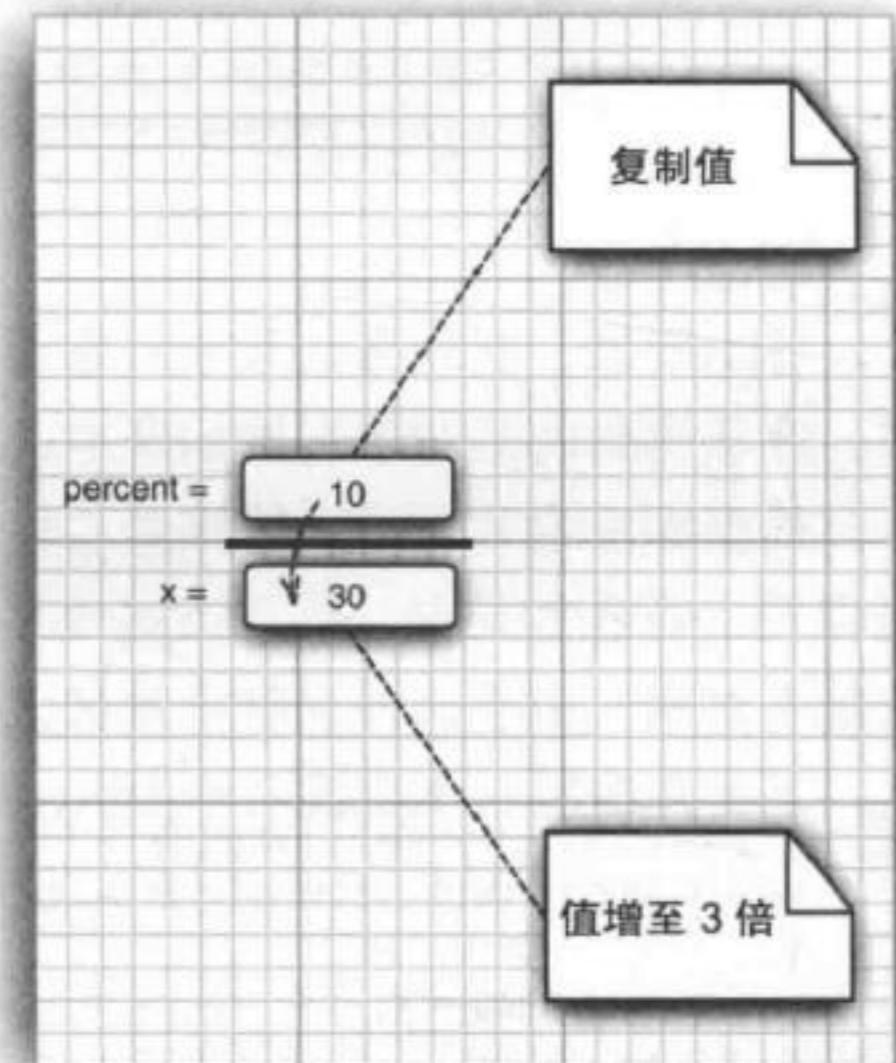


图 4-6 修改参数变量没有持续效果

2. `raiseSalary` 方法应用于这个对象引用。`x` 和 `harry` 同时引用的那个 `Employee` 对象的工资提高了 200%。

3. 方法结束后，参数变量 `x` 不再使用。当然，对象变量 `harry` 继续引用那个工资增至 3 倍的员工对象（如图 4-7 所示）。

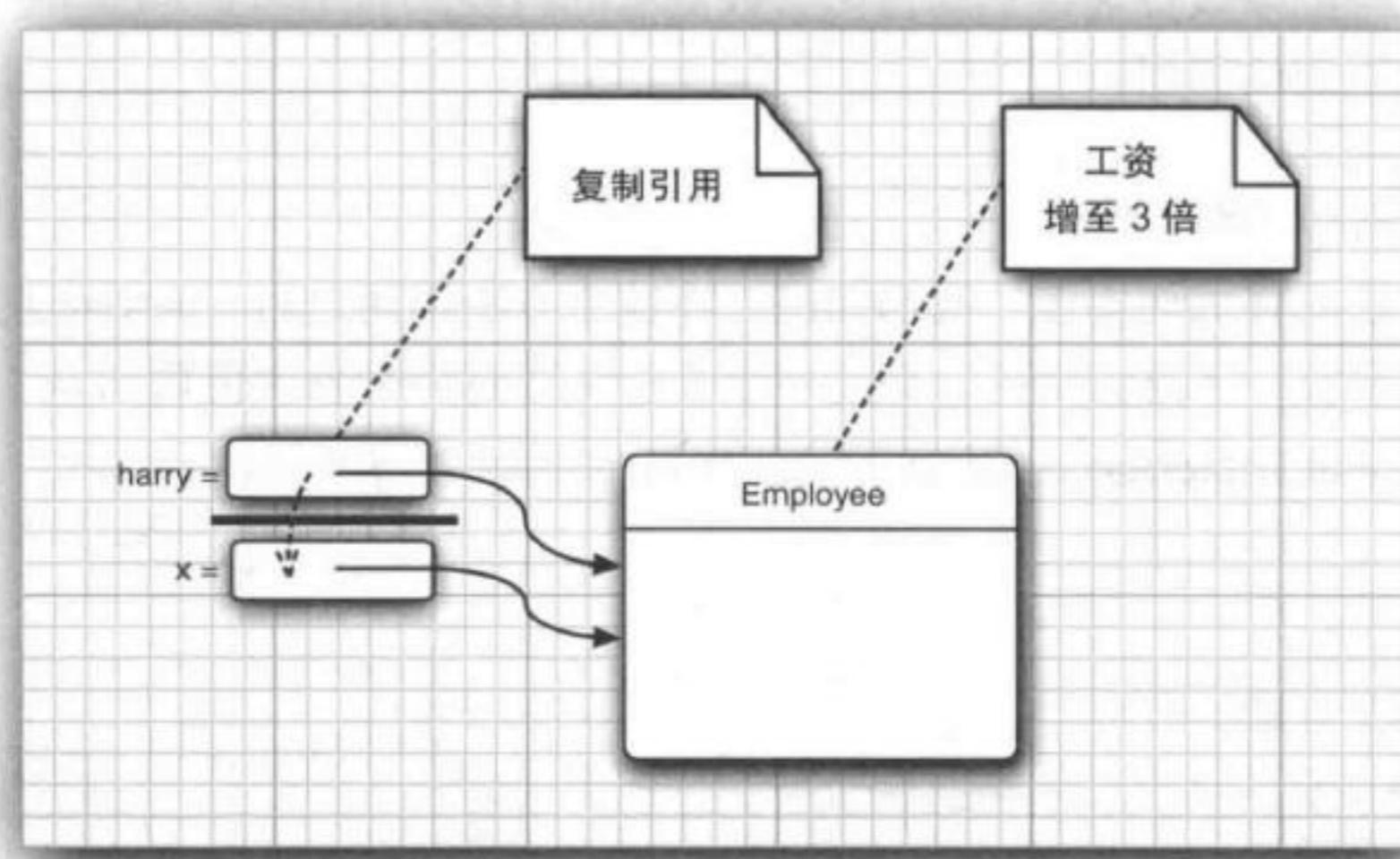


图 4-7 修改参数引用的对象有持续效果

可以看到，实现方法改变对象参数的状态是完全可以的，实际上也相当常见。理由很简单，方法得到的是对象引用的副本，原来的对象引用和这个副本都引用同一个对象。

很多程序设计语言（特别是 C++ 和 Pascal）提供了两种参数传递方式：按值调用和按引用调用。有些程序员（甚至有些书的作者）声称 Java 对对象采用的是按引用调用，实际上，这是不对的。由于这种误解很常见，所以很有必要给出一个反例来详细地说明这个问题。

下面来编写一个交换两个 `Employee` 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 对对象采用的是按引用调用，那么这个方法就应该能够实现交换：

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

但是，这个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` 初始化为两个对象引用的副本，这个方法交换的是这两个副本。

```
// 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。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）。

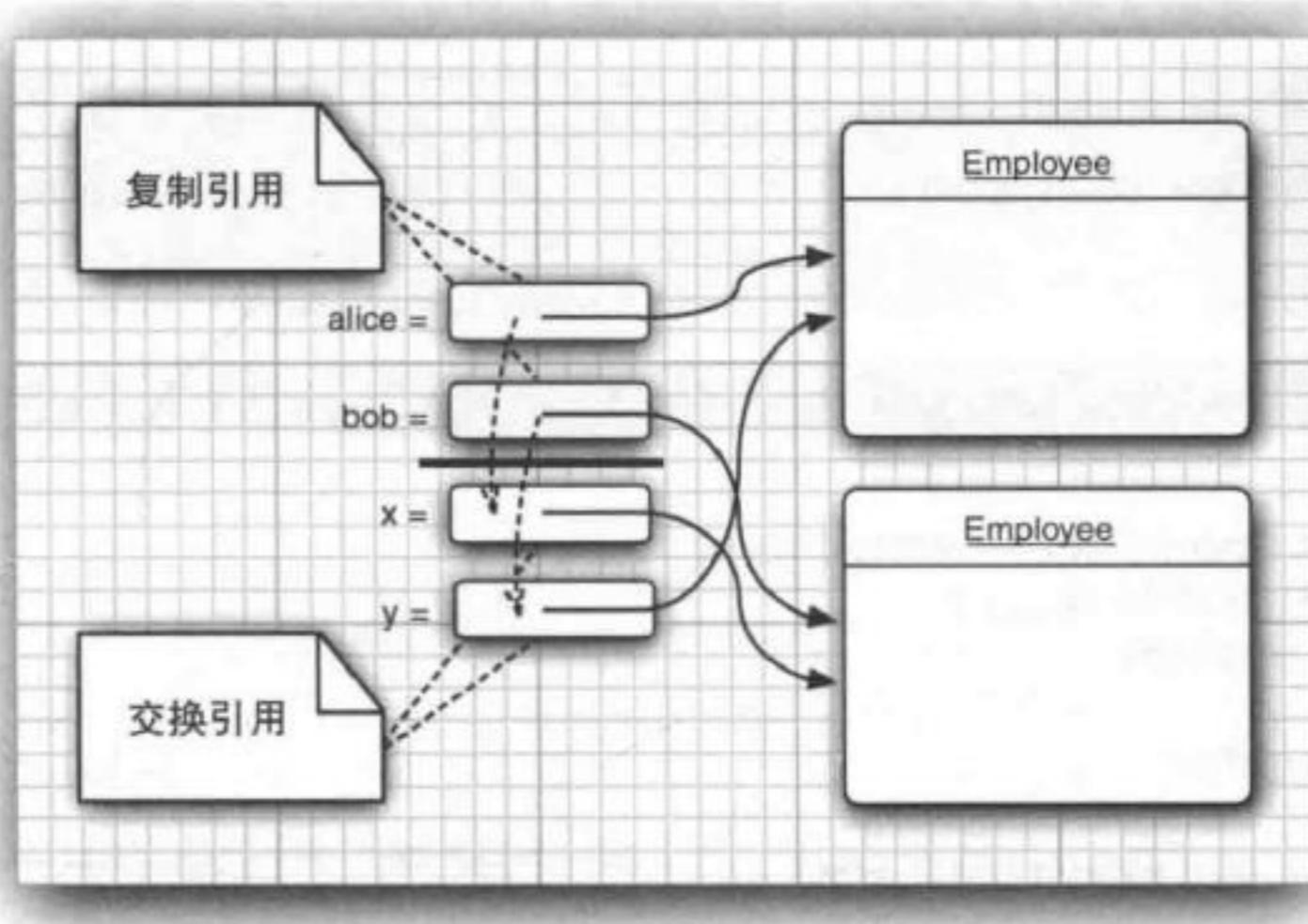


图 4-8 交换参数变量没有持续效果

这说明：Java 程序设计语言对对象采用的不是按引用调用。实际上，对象引用（object reference）是按值传递的。

下面来总结在 Java 中对方法参数能做什么和不能做什么：

- 方法不能修改基本数据类型的参数（即数值型或布尔型）。
- 方法可以改变对象参数的状态。
- 方法不能让一个对象参数引用一个新对象。

程序清单 4-4 中的程序展示了这几点。在这个程序中，首先试图将一个数值参数的值增至 3 倍，但没有成功：

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

随后，成功地将一个员工的工资增至 3 倍：

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

方法结束之后，`harry` 引用的对象的状态发生了改变。这是因为这个方法可以通过对象引用的副本修改所引用对象的状态。

最后，程序演示了 `swap` 方法的失败结果：

```
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 交换了，但是变量 a 和 b 没有受到影响。

C++ 注释：C++ 中有按值调用和按引用调用。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) 方法来修改它们的引用参数。

程序清单 4-4 ParamTest/ParamTest.java

```
1 /**
2  * This program demonstrates parameter passing in Java.
3  * @version 1.01 2018-04-10
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         var 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         var a = new Employee("Alice", 70000);
33         var 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     private String name;
66     private double salary;
67
68     public Employee(String n, double s)
69     {
70         name = n;
71         salary = s;
72     }
73
74     public String getName()
75     {
76         return name;
77     }
78
79     public double getSalary()
80     {
81         return salary;
82     }
83
84     public void raiseSalary(double byPercent)
85     {
86         double raise = salary * byPercent / 100;
87         salary += raise;
88     }
89 }
```

4.6 对象构造

前面已经学习了如何编写简单的构造器来定义对象的初始状态。不过，由于对象构造非

常重要，所以 Java 提供了多种编写构造器的机制。下面几节将详细介绍这些机制。

4.6.1 重载

有些类有多个构造器。例如，可以如下构造一个空的 `StringBuilder` 对象：

```
var messages = new StringBuilder();
```

或者，可以指定一个初始字符串：

```
var todoList = new StringBuilder("To do:\n");
```

这种功能叫作重载（overloading）。如果多个方法（比如，`StringBuilder` 构造器方法）有相同的方法名但有不同的参数，便出现了重载。编译器必须挑选出具体调用哪个方法。它用各个方法首部中的参数类型与特定方法调用中所使用的值类型进行匹配，来选出正确的方法。如果编译器无法匹配参数，就会产生编译时错误，这可能因为根本不存在匹配，或者所有重载方法中没有一个相对更好的方法（这个查找匹配的过程称为重载解析（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，需要指定方法名以及参数类型，这叫作方法的签名（signature）。例如，`String` 类有 4 个名为 `indexOf` 的公共方法。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、参数类型也相同却有不同返回类型的方法。

4.6.2 默认字段初始化

如果在构造器中没有显式地为一个字段设置初始值，就会将它自动设置为默认值：数值将设置为 0，布尔值为 `false`，对象引用为 `null`。有些人认为依赖默认值的做法是一种不好的编程实践。确实，如果不明确地对字段进行初始化，就会影响程序代码的可读性。

注释：这是字段与局部变量的一个重要区别。方法中的局部变量必须明确地初始化。但是在类中，如果没有初始化类中的字段，将会自动初始化为默认值（0、`false` 或 `null`）。

例如，考虑 `Employee` 类。假定没有在构造器中指定如何初始化某些字段，默认情况下，就会将 `salary` 字段初始化为 0，将 `name` 和 `hireDay` 字段初始化为 `null`。

但是，这并不是一个好主意。如果有人调用 `getName` 方法或 `getHireDay` 方法，就会得到一个 `null` 引用，这可能不是他们想要的结果：

```
LocalDate h = harry.getHireDay();
int year = h.getYear(); // throws exception if h is null
```

4.6.3 无参数的构造器

很多类都包含无参数的构造器，由无参数构造器创建对象时，对象的状态会设置为适当

的默认值。例如，以下是 Employee 类的一个无参数构造器：

```
public Employee()
{
    name = "";
    salary = 0;
    hireDay = LocalDate.now();
}
```

如果你写的类没有构造器，就会为你提供一个无参数构造器。这个构造器将所有的实例字段设置为相应的默认值。所以，实例字段中的所有数值型数据会设置为 0，所有布尔值设置为 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，但是没有提供无参数构造器，那么构造对象时就必须提供参数，否则就是不合法的。例如，程序清单 4-2 中的 Employee 类提供了一个构造器：

```
public Employee(String n, double s, int year, int month, int day)
```

对于这个类，构造默认的员工就是不合法的。也就是说，以下调用

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

将产生错误。

! **警告：**请记住，仅当类没有任何其他构造器的时候，你才会得到一个默认的无参数构造器。编写类的时候，如果写了一个你自己的构造器，要想让这个类的使用者能够通过以下调用创建一个实例：

```
new ClassName()
```

你就必须提供一个无参数的构造器。当然，如果接受所有字段设置为默认值，则只需要提供以下代码：

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

C++ 注释：C++ 对于构造字段有一种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++ 使用这种特殊语法来避免不必要的调用无参数构造器。在 Java 中，不需要这种语法，因为对象没有子对象，只有其他对象的引用。

4.6.4 显式字段初始化

通过重载类的构造器方法，可以采用多种形式设置类实例字段的初始状态。不论调用哪个构造器，每个实例字段都要设置为一个有意义的初始值，确保这一点总是一个好主意。

可以在类定义中直接为任何字段赋值。例如：

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

在执行构造器之前完成这个赋值。如果一个类的所有构造器都需要把某个特定的实例字段设置为同一个值，那么这个语法尤其有用。

初始值不一定是常量值。在下面的例子中，就是利用方法调用初始化一个字段。考虑以下 Employee 类，其中每个员工有一个 id 字段。可以使用以下方式进行初始化：

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

4.6.5 参数名

在编写很小的构造器时（这十分常见），在为参数命名时可能有些困惑。

我们通常喜欢用单个字母作为参数名：

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

但这样做有一个缺点：只有阅读代码才能够了解参数 n 和参数 s 的含义。

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;
}
```

这样更好一些。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：参数变量会遮蔽同名的实例字段。例如，如果将参数命名为 salary，那么 salary 将指示这个参数，而不是实例字段。但是，还是可以用 this.salary 访问实例字段。回想一下，this 指示隐式参数，也就是所构造的对象。下面来看一个示例：

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

C++ 注释：在 C++ 中，经常用下画线或某个固定的字母（一般选用 m 或 x）作为实例字段的前缀。例如，salary 字段可能被命名为 _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 指示一个方法的隐式参数。不过，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，这样只需要写一次公共构造代码。

C++ 注释：在 Java 中，this 引用等价于 C++ 中的 this 指针。但是，在 C++ 中，一个构造器不能调用另一个构造器。如果在 C++ 中想抽取出公共的初始化代码，则必须编写一个单独的方法。

4.6.7 初始化块

前面已经介绍过两种初始化实例字段的方法：

- 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，可以包含任意的代码块。构造这个类的对象时，这些块就会执行。例如，

```
class Employee
{
    private static int nextId;

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

    // object initialization block
```

```

{
    id = nextId;
    nextId++;
}

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

public Employee()
{
    name = "";
    salary = 0;
}

...
}

```

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

这种机制不是必需的，也不常见。通常会直接将初始化代码放在构造器中。

注释：可以在初始化块中设置字段，即使这些字段在类后面才定义，这是合法的。但是，为了避免循环定义，不允许读取在后面初始化的字段。具体规则请参见 Java 语言规范的 8.3.3 节 (<http://docs.oracle.com/javase/specs>)。这些规则太过复杂，让编译器的实现者都很头疼，所以较早的 Java 版本中这些规则的实现存在一些小错误。因此，建议总是将初始化块放在字段定义之后。

由于初始化实例字段有多种途径，所以列出构造过程的所有路径可能让人很费解。下面是调用构造器时的具体处理步骤：

1. 如果构造器的第一行调用了另一个构造器，则基于所提供的参数执行第二个构造器。
2. 否则，
 - a) 所有实例字段初始化为其默认值（`0`、`false` 或 `null`）。
 - b) 按照在类声明中出现的顺序，执行所有字段初始化方法和初始化块。
3. 执行构造器主体代码。

当然，最好精心地组织初始化代码，以便其他程序员轻松理解，而不要求他们都是语言专家。例如，如果让类的构造器依赖于实例字段声明的顺序，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值，或者使用一个静态的初始化块来初始化静态字段。前面已经介绍过第一种机制：

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

如果类的静态字段需要很复杂的初始化代码，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。我们希望将员工 ID 的起始值赋为一个小于 10 000 的随机整数。

```

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

```

在类第一次加载的时候，会完成静态字段的初始化。与实例字段一样，除非将静态字段显式地设置成其他值，否则默认的初始值为 0、false 或 null。所有的静态字段初始化方法以及静态初始化块都将依照类声明中出现的顺序执行。

注释：让人惊讶的是，在 JDK 6 之前，完全可以用 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，就会加载这个类，静态初始化块将会打印“Hello, World”。在此之后才会显示一个消息指出 `main` 未定义。从 Java 7 以后，`java` 程序会首先检查是否有一个 `main` 方法。

这个例子使用了 `Random` 类来生成随机数。从 JDK 17 开始，`java.util.random` 包提供了考虑多种因素的强算法的实现。阅读 `java.util.random` 包的 API 文档，其中对如何选择算法给出了建议。然后通过提供算法名来得到一个实例，如下所示：

```
RandomGenerator generator = RandomGenerator.of("L64X128MixRandom");
```

调用 `generator.nextInt(n)` 或其他 `RandomGenerator` 方法来生成随机数。（`RandomGenerator` 是一个接口，第 6 章将介绍接口概念。`Random` 类的对象可以使用所有 `RandomGenerator` 方法。）

程序清单 4-5 中的程序展示了本节讨论的很多特性：

- 重载构造器；
- 用 `this(...)` 调用另一个构造器；
- 无参数构造器；
- 对象初始化块；
- 静态初始化块；
- 实例字段初始化。

程序清单 4-5 ConstructorTest/ConstructorTest.java

```

1 import java.util.*;
2
3 /**
4  * This program demonstrates object construction.
5  * @version 1.02 2018-04-10

```

```
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        var 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     private static int nextId;
29
30     private int id;
31     private String name = ""; // instance field initialization
32     private double salary;
33
34     private static Random generator = new Random();
35
36     // static initialization block
37     static
38     {
39         // set nextId to a random number between 0 and 9999
40         nextId = generator.nextInt(10000);
41     }
42
43     // object initialization block
44     {
45         id = nextId;
46         nextId++;
47     }
48
49     // three overloaded constructors
50     public Employee(String n, double s)
51     {
52         name = n;
53         salary = s;
54     }
55
56     public Employee(double s)
57     {
58         // calls the Employee(String, double) constructor
59         this("Employee #" + nextId, s);
60     }
}
```

```

61
62     // the default constructor
63     public Employee()
64     {
65         // name initialized to ""--see above
66         // salary not explicitly set--initialized to 0
67         // id initialized in initialization block
68     }
69
70     public String getName()
71     {
72         return name;
73     }
74
75     public double getSalary()
76     {
77         return salary;
78     }
79
80     public int getId()
81     {
82         return id;
83     }
84 }
```

API `java.util.Random 1.0`

- `Random()`

构造一个新的随机数生成器。

API `java.util.random.RandomGenerator 1.7`

- `int nextInt(int n)`

返回一个 $0 \sim n-1$ 之间的随机数。

- `static RandomGenerator of(String name)`

由给定算法名生成一个随机数生成器。算法 “`L64X128MixRandom`” 对大多数应用都适用。

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，其中放置一些清理代码，当对象不再使用可能需要执行这些清理代码。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 会完成自动的垃圾回收，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，将其回收和再利用就十分重要。

如果一个资源一旦使用完就需要立即关闭，那么应当提供一个 `close` 方法来完成必要的清理工作。可以在对象使用完时调用这个 `close` 方法。第 7 章将介绍如何确保自动调用这个方法。

如果可以等到虚拟机退出，那么可以用方法 `Runtime.addShutdownHook` 增加一个“关闭钩”（shutdown hook）。在 Java 9 中，可以使用 `Cleaner` 类注册一个动作，当对象不再可达时（除了清洁器还能访问，其他对象都无法访问这个对象），就会完成这个动作。在实际中这些情况很少见。可以参见 API 文档来了解这两种方法的详细内容。

！ 警告：不要使用 `finalize` 方法来完成清理。这个方法原本要在垃圾回收器清理对象之前调用。不过，你并不能知道这个方法到底什么时候调用，而且该方法已经被废弃。

4.7 记录

有时，数据就只是数据，而面向对象程序设计提供的数据隐藏有些碍事。考虑一个类 `Point`，这个类描述平面上的一个点，有 `x` 和 `y` 坐标。

当然，可以如下创建一个类：

```
class Point
{
    private final double x;
    private final double y;
    public Point(double x, double y) { this.x = x; this.y = y; }
    public getX() { return x; }
    public getY() { return y; }
    public String toString() { return "Point[x=%d, y=%d]".formatted(x, y); }
    // More methods ...
}
```

这里隐藏了 `x` 和 `y`，然后通过获取方法来获得这些值，不过，这种做法对我们确实有好处吗？

我们将来想改变 `Point` 的实现吗？当然，还有极坐标，不过对于图形 API，你可能不会使用极坐标。在实际中，平面上的一个点就用 `x` 和 `y` 坐标来描述。

为了更简洁地定义这些类，JDK 14 引入了一个预览特性：“记录”。最终版本在 JDK 16 中发布。

4.7.1 记录概念

记录（record）是一种特殊形式的类，其状态不可变，而且公共可读。可以如下将 `Point` 定义为一个记录：

```
record Point(double x, double y) {}
```

其结果是有以下实例字段的类：

```
private final double x;
private final double y;
```

在 Java 语言规范中，一个记录的实例字段称为组件（component）。

这个类有一个构造器：

```
Point(double x, double y)
```

和以下访问器方法：

```
public double x()
public double y()
```

注意，访问器方法名为 `x` 和 `y`，而不是 `getX` 和 `getY`。（Java 中实例字段可以与方法同名，这是合法的。）

```
var p = new Point(3, 4);
System.out.println(p.x() + " " + p.y());
```

注释：Java 没有遵循 `get` 约定，因为那有些麻烦。对于布尔字段，通常使用 `is` 而不是 `get`。而且首字母大写可能有问题。如果一个类既有 `x` 字段又有 `X` 字段，会发生什么？有些程序员不太满意，因为他们原先的类不能轻松地变为记录。不过实际上，那些遗留类中，很多都是可变的，所以并不适合转换为记录。

除了字段访问器方法，每个记录有 3 个自动定义的方法：`toString`、`equals` 和 `hashCode`。下一章会更多地了解这些方法。

警告：对于这些自动提供的方法，也可以定义你自己的版本，只要它们有相同的参数和返回类型。例如，下面的定义就是合法的：

```
record Point(double x, double y)
{
    public double x() { return y; } // BAD
}
```

不过，这并不是一个好主意。

可以为一个记录增加你自己的方法：

```
record Point(double x, double y)
{
    public double distanceFromOrigin() { return Math.hypot(x, y); }
}
```

与所有其他类一样，记录可以有静态字段和方法：

```
record Point(double x, double y)
{
    public static Point ORIGIN = new Point(0, 0);
    public static double distance(Point p, Point q)
    {
        return Math.hypot(p.x - q.x, p.y - q.y);
    }
    ...
}
```

不过，不能为记录增加实例字段：

```
record Point(double x, double y)
{
    private double r; // ERROR
    ...
}
```

◆ 警告：记录的实例字段自动为 final 字段。不过，它们可能是可变对象的引用。

```
record PointInTime(double x, double y, Date when) {}
```

这样记录实例将是可变的：

```
var pt = new PointInTime(0, 0, new Date());
pt.when().setTime(0);
```

如果希望记录实例是不可变的，那么字段就不能使用可变的类型。

✓ 提示：对于完全由一组变量表示的不可变数据，要使用记录而不是类。如果数据是可变的，或者数据表示可能随时间改变，则使用类。记录更易读、更高效，而且在并发程序中更安全。

4.7.2 构造器：标准、自定义和简洁

自动定义地设置所有实例字段的构造器称为标准构造器（canonical constructor）。

还可以定义另外的自定义构造器（custom constructor）。这种构造器的第一个语句必须调用另一个构造器，所以最终会调用标准构造器。下面来看一个例子：

```
record Point(double x, double y)
{
    public Point() { this(0, 0); }
}
```

这个记录有两个构造器：标准构造器和一个生成原点的无参数构造器。

如果标准构造器需要完成额外的工作，那么可以提供你自己的实现：

```
record Range(int from, int to)
{
    public Range(int from, int to)
    {
        if (from <= to)
        {
            this.from = from;
            this.to = to;
        }
        else
        {
            this.from = to;
            this.to = from;
        }
    }
}
```

不过，实现标准构造器时，建议使用一种简洁（compact）形式（见程序清单 4-6）。不用指定参数列表：

```
record Range(int from, int to)
{
    public Range // Compact form
    {
        if (from > to) // Swap the bounds
        {
            int temp = from;
```

```
        from = to;  
        to = temp;  
    }  
}
```

简洁形式的主体是标准构造器的“前奏”。它只是在为实例字段 `this.from` 和 `this.to` 赋值之前修改参数变量 `from` 和 `to`。不能在简洁构造器的主体中读取或修改实例字段。

程序清单 4-6 RecordTest/RecordTest.java

```
1 import java.util.*;
2
3 /**
4 * This program demonstrates records.
5 * @version 1.0 2021-05-13
6 * @author Cay Horstmann
7 */
8 public class RecordTest
9 {
10     public static void main(String[] args)
11     {
12         var p = new Point(3, 4);
13         System.out.println("Coordinates of p: " + p.x() + " " + p.y());
14         System.out.println("Distance from origin: " + p.distanceFromOrigin());
15         // Same computation with static field and method
16         System.out.println("Distance from origin: " + Point.distance(Point.ORIGIN, p));
17
18         // A mutable record
19         var pt = new PointInTime(3, 4, new Date());
20         System.out.println("Before: " + pt);
21         pt.when().setTime(0);
22         System.out.println("After: " + pt);
23
24         // Invoking a compact constructor
25
26         var r = new Range(4, 3);
27         System.out.println("r: " + r);
28     }
29 }
30
31 record Point(double x, double y)
32 {
33     // A custom constructor
34     public Point() { this(0, 0); }
35     // A method
36     public double distanceFromOrigin()
37     {
38         return Math.hypot(x, y);
39     }
40     // A static field and method
41     public static Point ORIGIN = new Point();
42     public static double distance(Point p, Point q)
43     {
44         return Math.hypot(p.x - q.x, p.y - q.y);
45     }
46 }
```

```

45     }
46 }
47
48 record PointInTime(double x, double y, Date when) { }
49
50 record Range(int from, int to)
51 {
52     // A compact constructor
53     public Range
54     {
55         if (from > to) // Swap the bounds
56         {
57             int temp = from;
58             from = to;
59             to = temp;
60         }
61     }
62 }

```

4.8 包

Java 允许使用包（package）将类组织在一个集合中。借助包可以方便地组织你的代码，并将你自己的代码与其他人提供的代码库分开。下面我们将介绍如何使用和创建包。

4.8.1 包名

使用包的主要原因是确保类名的唯一性。假如两个程序员不约而同地提供了 Employee 类，只要他们将自己的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唯一性，可以使用一个因特网域名（这显然是唯一的）以逆序的形式作为包名，然后对于不同的项目使用不同的子包。例如，考虑域名 horstmann.com。如果逆序来写，就得到了包名 com.horstmann。然后可以追加一个项目名，如 com.horstmann.corejava。如果再把 Employee 类放在这个包里，那么这个类的“完全限定”名就是 com.horstmann.corejava.Employee。

注释：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。每一个包都是独立的类集合。

4.8.2 类的导入

一个类可以使用所属包（这个类所在的包）中的所有类，以及其他包中的公共类（public class）。

我们可以采用两种方式访问另一个包中的公共类。第一种方式是使用完全限定名（fully qualified name），也就是包名后面跟着类名。例如：

```
java.time.LocalDate today = java.time.LocalDate.now();
```

这显然很烦琐。更简单且更常用的方式是使用 import 语句。import 语句的关键是可以提

供一种简写方式来引用包中各个类。一旦增加了 import 语句，在使用类时，就不必写出类的全名了。

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.time 包中的所有类。

```
import java.time.*;
```

然后，就可以使用

```
LocalDate today = LocalDate.now();
```

而不需要在前面加上包前缀。还可以导入一个包中的特定类：

```
import java.time.LocalDate;
```

java.time.* 的语法比较简单，对代码的规模也没有任何负面影响。不过，如果能够明确地指出所导入的类，那么代码的读者就能更加准确地知道你使用了哪些类。

 提示：在 Eclipse 中，可以使用菜单选项 Source → Organize Imports。诸如 import java.util.*; 等包语句将会自动扩展为一组特定的导入语句，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，可以只导入你需要的包，并无须过多考虑。但在发生命名冲突的时候，就要注意包了。例如，java.util 和 java.sql 包都有 Date 类。假设在程序中导入了这两个包：

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

在程序中使用 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，编译器无法确定你想使用的是哪一个 Date 类。可以增加一个特定的 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。

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

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

C++ 注释：C++ 程序员有时会将 `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` 来包含外部特性的声明，这是因为，除了正在编译的文件以及显式包含的头文件，C++ 编译器不会查看任何其他文件。Java 编译器则不同，只要你告诉它文件在哪里，它很乐于查看其他文件。

在 Java 中，通过显式地给出完整的类名，如 `java.util.Date`，可以完全避免使用 `import` 机制；而在 C++ 中，则无法避免使用 `#include` 指令。

`import` 语句唯一的好处是简捷。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，可以只用 `Date` 来引用 `java.util.Date` 类。

在 C++ 中，与包机制类似的是命名空间（namespace）特性。可以认为 Java 中的 `package` 和 `import` 语句类似于 C++ 中的 `namespace` 和 `using` 指令。

4.8.3 静态导入

有一种 `import` 语句允许导入静态方法和静态字段，而不只是类。

例如，如果在源文件最上面添加一条指令：

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

就可以使用 `System` 类的静态方法和静态字段，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字段：

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

实际上，是否有很多程序员想要用简写 `System.out` 或 `System.exit`，这一点很让人怀疑。这样写出的代码看起来不太清晰。不过，

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

看起来则比

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

简洁得多。

4.8.4 在包中增加类

要想将类放入包中，就必须将包名放在源文件的开头，即放在定义这个包中各个类的代码之前。例如，程序清单 4-8 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，那么这个源文件中的类就属于无名包（unnamed package）。无名包没有包名。到目前为止，我们定义的所有类都在无名包中。

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

程序清单 4-7 和程序清单 4-8 中的程序分别放在两个包中：PackageTest 类属于无名包；Employee 类属于 com.horstmann.corejava 包。因此，Employee.java 文件必须在子目录 com/horstmann/corejava 中。换句话说，目录结构如下所示：

```
. (base directory)
└── PackageTest.java
└── PackageTest.class
└── com/
    └── horstmann/
        └── corejava/
            └── Employee.java
            └── Employee.class
```

要想编译这个程序，只需切换到基目录，并运行以下命令

```
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。在这里没有使用无名包，而是将类分别放在不同的包中（com.horstmann.corejava 和 com.mycompany）。

```
. (base directory)
└── 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 目录的目录：

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

再次强调，编译器处理文件（带有文件分隔符和扩展名 .java 的文件），而 Java 解释器加载类（带有 . 分隔符）。

 **提示：**从下一章开始，我们将对源代码使用包。这样一来，就可以为各章建立一个 IDE 项目，而不是各小节分别建立项目。

 **警告：**编译器在编译源文件的时候不检查目录结构。例如，假设一个源文件开头有以下指令：

```
package com.mycompany;
```

即使这个源文件不在子目录 com/mycompany 下，这个文件也可以编译。如果它不依赖于其他包，就可以通过编译而不会出现编译错误。但是，最终的程序将无法运行，除非先将所有类文件移到正确的位置上。如果包与目录不匹配，虚拟机就找不到这些类。

程序清单 4-7 PackageTest/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  * @version 1.11 2004-02-19
9  * @author Cay Horstmann
10 */
11 public class PackageTest
12 {
13     public static void main(String[] args)
14     {
15         // because of the import statement, we don't have to use
16         // com.horstmann.corejava.Employee here
17         var 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 }
```

程序清单 4-8 PackageTest/com/horstmann/corejava/Employee.java

```

1 package com.horstmann.corejava;
2
3 // the classes in this file are part of this package
4
5 import java.time.*;
6
7 // import statements come after the package statement
8
9 /**
10  * @version 1.11 2015-05-08
11  * @author Cay Horstmann
12 */
13 public class Employee
14 {
15     private String name;
16     private double salary;
17     private LocalDate hireDay;
18
19     public Employee(String name, double salary, int year, int month, int day)
```

```

28  {
29      this.name = name;
30      this.salary = salary;
31      hireDay = LocalDate.of(year, month, day);
32  }
33
34  public String getName()
35  {
36      return name;
37  }
38
39  public double getSalary()
40  {
41      return salary;
42  }
43
44  public LocalDate getHireDay()
45  {
46      return hireDay;
47  }
48
49  public void raiseSalary(double byPercent)
50  {
51      double raise = salary * byPercent / 100;
52      salary += raise;
53  }
54 }
```

4.8.5 包访问

前面已经见过访问修饰符 `public` 和 `private`。标记为 `public` 的部分可以由任意类使用；标记为 `private` 的部分只能由定义它们的类使用。如果没有指定 `public` 或 `private`，这个部分（类、方法或变量）可以由同一个包中的所有方法访问。

下面再来考虑程序清单 4-2。在这个程序中，没有将 `Employee` 类定义为公共类，因此只有在同一个包（在此是无名包）中的其他类（例如 `EmployeeTest`）可以访问这个类。对于类来说，这种默认方式是合乎情理的。但是，对于变量来说就有些不适宜了，变量必须显式地标记为 `private`，不然的话将默认为包可访问。显然，这样会破坏封装性。问题是人们经常忘记键入关键字 `private`。以 `java.awt` 包中的 `Window` 类为例（`java.awt` 包是 JDK 提供的源代码的一部分）：

```

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` 类的方法访问这个变量，因此本应该将它设置为私有变量才合适。可能是程序员敲代码时匆忙之中忘记 `private` 修饰符了？也可能没有人关心这个问题？已经 20 多年了，这个变量仍然不是私

有变量。不仅如此，这个类还陆续增加了一些新的字段，而其中大约有一半也不是私有的。

这可能会成为一个问题。在默认情况下，包不是封闭的实体。也就是说，任何人都可以向包中添加更多的类。当然，有恶意或糟糕的程序员很可能利用包访问添加一些能修改变量的代码。例如，在 Java 程序设计语言的早期版本中，只需要将以下这条语句放在类文件的开头，就可以很容易地在 `java.awt` 包中混入其他类：

```
package java.awt;
```

然后，把得到的类文件放置在类路径上某处的 `java.awt` 子目录下，这样就可以访问 `java.awt` 包的内部了。使用这一手段，完全可以修改警告字符串（如图 4-9 所示）。

从 1.2 版开始，JDK 的实现者修改了类加载器，明确地禁止加载包名以 "java." 开头的用户自定义的类！当然，用户自定义的类无法从这种保护中受益。另一种机制是让 JAR 文件声明包为密封的（sealed），以防止第三方修改，但这种机制已经过时。现在应当使用模块封装包。我们会在卷 II 的第 9 章详细讨论模块。



图 4-9 在一个 applet 窗口中修改警告字符串

4.8.6 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，可以包含多个压缩格式的类文件和子目录，这样既可以节省空间又可以改善性能。在程序中用到第三方的库时，你通常会得到一个或多个需要包含的 JAR 文件。第 11 章将介绍如何创建你自己的 JAR 文件。



提示：JAR 文件使用 ZIP 格式组织文件和子目录。可以使用任何 ZIP 工具查看 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. 把类文件放到一个目录中，例如 `/home/user/classdir`。需要注意，这个目录是包树状结构的基目录。如果希望增加 `com.horstmann.corejava.Employee` 类，那么 `Employee.class` 类文件就必须位于子目录 `/home/user/classdir/com/horstmann/corejava` 中。
2. 将 JAR 文件放在一个目录中，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，类路径中的各项之间用冒号（`:`）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
```

不论是 UNIX 还是 Windows，都用句点（`.`）表示当前目录。

类路径包括：

- 基目录 `/home/user/classdir` 或 `c:\classdir`；

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

从 Java 6 开始，可以为 JAR 文件目录指定一个通配符，如下：

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

或者

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

在 UNIX 中，* 必须转义以防止 shell 扩展。

archives 目录中的所有 JAR 文件（但不包括 .class 文件）都包含在这个类路径中。

由于总是会搜索 Java API 的类，所以不必显式地包含在类路径中。

！ 警告：javac 编译器总是在当前目录中查找文件，但只有当类路径中包含“.”目录时，java 虚拟机才会查看当前目录。如果你没有设置类路径，那么没有什么问题，因为默认的类路径会包含“.”目录。但是如果你设置了类路径却忘记包含“.”目录，那么尽管你的程序可以没有错误地通过编译，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 类的类文件。它首先要查看 Java API 类。显然，在那里找不到相应的类文件，所以转而查看类路径。它会查找以下文件：

- /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（/home/user/archives/archive.jar 中）

编译器查找文件要比虚拟机复杂得多。如果引用了一个类，而没有指定这个类的包，那么编译器将首先查找包含这个类的包。它会查看所有的 import 指令，确定其中是否包含这个类。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。编译器将尝试查找 java.lang.Employee（因为总是会默认导入 java.lang 包）、java.util.Employee、com.horstmann.corejava.Employee 和当前包中的 Employee。它会在类路径所有位置中搜索以上各个类。如果找到了一个以上的类，就会产生编译时错误（因为完全限定类名必须是唯一的，所以 import 语句的次序并不重要）。

编译器的任务不止这些，它还要查看源文件是否比类文件新。如果是这样的话，那么源文件就会自动地重新编译。在前面已经知道，只可以导入其他包中的公共类。一个源文件只能包含一个公共类，并且文件名与公共类名必须匹配。因此，编译器很容易找到公共类的源文件。不过，还可以从当前包中导入非公共类。这些类有可能在与类名不同的源文件中定义。如果从当前包中导入一个类，那么编译器就要搜索当前包中的所有源文件，查看哪个源文件定义了这个类。

4.8.7 设置类路径

最好使用 `-classpath` (或 `-cp`, 或者 Java 9 中的 `--class-path`) 选项指定类路径:

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

或者

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

整个指令必须写在一行中。将这样一个很长的命令行放在一个 shell 脚本或一个批处理文件中是个不错的主意。

利用 `-classpath` 选项设置类路径是首选的方法，另一种方法是通过设置 `CLASSPATH` 环境变量来指定类路径。具体细节依赖于所使用的 shell。在 Bourne Again shell (bash) 中，命令如下：

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

在 Windows shell 中，命令如下：

```
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：有人建议永久地设置 `CLASSPATH` 环境变量。一般来说这是一个糟糕的想法。人们有可能会忘记全局设置，因此，当他们的类没有正确地加载时，就会感到很奇怪。一个颇受诟病的示例是 Windows 中 Apple QuickTime 安装程序。很多年来，它都将 `CLASSPATH` 全局设置为指向它需要的一个 JAR 文件，而没有在类路径中包含当前目录。因此，当程序编译后却不能运行时，无数 Java 程序员不得不花费很多精力去解决这个问题。

◆ 警告：过去，有人建议完全绕过类路径，将所有的 JAR 文件都放在 `jre/lib/ext` 目录中。这种机制在 Java 9 中已经过时，不过不管怎样这都是一个不好的建议。从扩展目录加载一些已经遗忘很久的类时，这会让人非常困惑。

注释：在 Java 9 中，还可以从模块路径加载类。本书卷 II 的第 9 章将讨论模块和模块路径。

4.9 JAR 文件

在将应用程序打包时，你希望只向用户提供一个单独的文件，而不是一个包含大量类文件的目录结构，Java 归档 (JAR) 文件就是为此目的而设计的。JAR 文件既可以包含类文件，也可以包含诸如图像和声音等其他类型的文件。此外，JAR 文件是压缩的，它使用了我们熟悉的 ZIP 压缩格式。

4.9.1 创建 JAR 文件

可以使用 `jar` 工具制作 JAR 文件 (在默认的 JDK 安装中，这个工具位于 `jdk/bin` 目录

下)。创建一个新 JAR 文件最常用的命令使用以下语法:

```
jar cvf jarFileName file1 file2 . . .
```

例如:

```
jar cvf CalculatorClasses.jar *.class icon.gif
```

通常, jar 命令的格式如下:

```
jar options file1 file2 . . .
```

表 4-2 列出了 jar 程序的所有选项。它们类似于 UNIX tar 命令的选项。

表 4-2 jar 程序选项

选项	说 明
c	创建一个新的或者空的存档文件并加入文件。如果指定的文件名是目录, jar 程序将会对它们进行递归处理
C	临时改变目录, 例如:
jar cvf jarFileName.jar -C classes *.class	切换到 classes 子目录以便增加类文件
e	在清单文件中创建一个入口点(请参见 4.9.3 节)
f	指定 JAR 文件名作为第二个命令行参数。如果没有这个参数, jar 命令会将结果写至标准输出(在创建 JAR 文件时)或者从标准输入读取输入(在解压或者列出 JAR 文件内容时)
i	创建索引文件(用于加快大型归档中的查找)
m	将一个清单文件添加到 JAR 文件中。清单文件是对归档内容和来源的一个说明。每个归档有一个默认的清单文件。但是, 如果想验证归档文件的内容, 可以提供你自己的清单文件
M	不为条目创建清单文件
t	显示内容表
u	更新一个已有的 JAR 文件
v	生成详细的输出
x	解压文件。如果提供一个或多个文件名, 只解压这些文件; 否则, 解压所有文件
0	存储, 但不进行 ZIP 压缩

可以将应用程序和代码库打包在 JAR 文件中。例如, 如果想在一个 Java 程序中发送邮件, 可以使用打包在文件 javax.mail.jar 中的一个库。

4.9.2 清单文件

除了类文件、图像和其他资源外, 每个 JAR 文件还包含一个清单文件(manifest), 用于描述归档文件的特殊特性。

清单文件被命名为 MANIFEST.MF, 它位于 JAR 文件的一个特殊的 META-INF 子目录中。合法的最小清单文件极其简单:

```
Manifest-Version: 1.0
```

复杂的清单文件可能包含更多条目。这些清单条目被分组为多个节。第一节被称为主节

(main section)。它作用于整个 JAR 文件。随后的条目可以指定命名实体的属性，如单个文件、包或者 URL。它们都必须以一个 Name 条目开始。节与节之间用空行分开。例如：

```
Manifest-Version: 1.0
lines describing this archive

Name: Woozle.class
lines describing this file
Name: com/mycompany/mypkg/
lines describing this package
```

要想编辑清单文件，需要将希望添加到清单文件中的行放到文本文件中，然后运行

```
jar cfm jarFileName manifestFileName . . .
```

例如，要创建一个包含清单文件的 JAR 文件，应该运行

```
jar cfm MyArchive.jar manifest.mf com/mycompany/mypkg/*.class
```

要想更新一个已有的 JAR 文件的清单，则需要将增加的部分放置到一个文本文件中，然后执行以下命令：

```
jar ufm MyArchive.jar manifest-additions.mf
```

注释：请参见 <https://docs.oracle.com/javase/10/docs/specs/jar/jar.html> 获得有关 JAR 文件和清单文件格式的更多信息。

4.9.3 可执行 JAR 文件

可以使用 jar 命令中的 e 选项指定程序的入口点，即通常调用 java 执行程序时指定的类：

```
jar cvfe MyProgram.jar com.mycompany.mypkg.MainAppClass files to add
```

或者，可以在清单文件中指定程序的主类，包括以下形式的语句：

```
Main-Class: com.mycompany.mypkg.MainAppClass
```

不要为主类名加扩展名 .class。

警告： 清单文件的最后一行必须以换行符结束。否则，将无法正确地读取清单文件。常见的一个错误是创建了一个只包含 Main-Class 行而没有行结束符的文本文件。

不论使用哪一种方法，用户都可以简单地通过下面的命令来启动程序：

```
java -jar MyProgram.jar
```

取决于操作系统的配置，用户甚至可以通过双击 JAR 文件图标来启动应用程序。下面是各种操作系统的操作方式：

- 在 Windows 平台中，Java 运行时安装程序将为 ".jar" 扩展名创建一个文件关联，会用 javaw -jar 命令启动文件（与 java 命令不同，javaw 命令不打开 shell 窗口）。
- 在 Mac OS X 平台中，操作系统能够识别 ".jar" 扩展名文件。双击 JAR 文件时就会执行 Java 程序。

不过，人们对 JAR 文件中的 Java 程序与原生应用还是感觉不同。在 Windows 平台上，可以使用第三方的包装器工具将 JAR 文件转换成 Windows 可执行文件。包装器是一个 Windows 程序，有大家熟悉的扩展名 .exe，它可以查找和加载 Java 虚拟机（JVM），或者在没有找到 JVM 时会告诉用户应该做些什么。有许多商业的和开源的产品，例如，Launch4J (<http://launch4j.sourceforge.net>) 和 IzPack (<http://izpack.org>)。

4.9.4 多版本 JAR 文件

随着模块和包强封装的引入，之前可以访问的一些内部 API 不再可用。这可能要求库提供商为不同 Java 版本发布不同的代码。为此，Java 9 引入了多版本 JAR (multi-release JAR)。

为了保证向后兼容，特定于版本的类文件放在 META-INF VERSIONS 目录中：

```

Application.class
BuildingBlocks.class
Util.class
META-INF
└ MANIFEST.MF (with line Multi-Release: true)
  versions
    9
      Application.class
      BuildingBlocks.class
    10
      BuildingBlocks.class

```

假设 Application 类使用了 CssParser 类，那么遗留版本的 Application.class 文件可以使用 com.sun.javafx.css.CssParser，而 Java 9 版本可以使用 javafx.css.CssParser。

Java 8 完全不知道 META-INF VERSIONS 目录，它只会加载遗留的类。Java 9 读取这个 JAR 文件时，则会使用新版本。

要增加不同版本的类文件，可以使用 --release 标志：

```
jar uf MyProgram.jar --release 9 Application.class
```

要从头构建一个多版本 JAR 文件，可以使用 -C 选项，对应每个版本要切换到一个不同的类文件目录：

```
jar cf MyProgram.jar -C bin/8 . --release 9 -C bin/9 Application.class
```

面向不同版本编译时，要使用 --release 标志和 -d 标志来指定输出目录：

```
javac -d bin/8 --release 8 . . .
```

在 Java 9 中，-d 选项会创建这个目录（如果原先该目录不存在）。

--release 标志也是 Java 9 新增的。在较早的版本中，需要使用 -source、-target 和 -bootclasspath 标志。JDK 现在为之前的两个 API 版本提供了符号文件。在 Java 9 中，编译时可以将 --release 设置为 9、8 或 7。

多版本 JAR 并不适用于不同版本的程序或库。对于不同的版本，所有类的公共 API 都应当是一样的。多版本 JAR 的唯一作用是使你的某个特定版本的程序或库能够使用多个不同的 JDK 版本。如果你增加了功能或者改变了一个 API，就应当提供一个新版本的 JAR。

注释：javap之类的工具并没有改造为可以处理多版本 JAR 文件。如果调用

```
javap -classpath MyProgram.jar Application.class
```

你会得到类的基本版本（毕竟，它与更新的版本应该有相同的公共 API）。如果必须查看更新的版本，则可以调用：

```
javap -classpath MyProgram.jar\!/META-INF VERSIONS/9/Application.class
```

4.9.5 关于命令行选项的说明

Java 开发包（JDK）的命令行选项一直以来都使用单个短横线加多字母选项名的形式，如：

```
java -jar . .
javac -Xlint:unchecked -classpath . . .
```

但 jar 命令是个例外，这个命令遵循经典的 tar 命令选项格式，而没有短横线：

```
jar cvf . . .
```

从 Java 9 开始，Java 工具开始转向一种更常用的选项格式，多字母选项名前面加两个短横线，另外对于常用的选项可以使用单字母快捷方式。例如，调用 Linux ls 命令时可以提供一个“human-readable”选项：

```
ls --human-readable
```

或者

```
ls -h
```

在 Java 9 中，可以使用 --version 而不是 -version，另外可以使用 --class-path 而不是 -classpath。在本书卷 II 的第 9 章中可以看到，--module-path 选项有一个快捷方式 -p。

详细内容可以参见 JEP 293 增强请求 (<http://openjdk.java.net/jeps/293>)。在所有清理工作中，作者还提出要标准化选项参数。带 -- 和多字母选项的参数用空格或者一个等号 (=) 分隔：

```
javac --class-path /home/user/classdir . . .
```

或

```
javac --class-path=/home/user/classdir . . .
```

单字母选项的参数可以用空格分隔，或者直接跟在选项后面：

```
javac -p moduledir . . .
```

或

```
javac -pmoduledir . . .
```

警告：后一种方式现在不能使用，而且一般来讲这也不是一个好主意。如果模块目录恰好是 parameters 或 processor，这就很容易与遗留的选项 (parameters 或 processor) 发生冲突，这又何必呢？

无参数的单字母选项可以组合在一起：

```
jar -cvf MyProgram.jar -e mypackage.MyProgram /*.class
```

！ 警告：目前不能使用这种方式，这肯定会带来混淆。假设 javac 有一个 -c 选项，那么 javac -cp 是指 javac -c -p 还是 -cp？

这就会带来一些混乱，希望过段时间能够解决这个问题。尽管我们想要远离这些古老的 jar 选项，但最好还是等到尘埃落定为妙。不过，如果你想做到最现代化，那么可以安全地使用 jar 命令的长选项：

```
jar --create --verbose --file jarFileName file1 file2 . . .
```

对于单字母选项，如果不组合，也是可以使用的：

```
jar -c -v -f jarFileName file1 file2 . . .
```

4.10 文档注释

JDK 包含一个很有用的工具，叫作 javadoc，它可以由源文件生成一个 HTML 文档。事实上，在第 3 章介绍的联机 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特殊定界符 `/**` 开始的注释，那么你也可以很容易地生成一个看上去具有专业水准的文档。这是一种很好的方法，因为这样可以将代码与注释放在一个地方。应该知道，如果将文档存放在一个单独的文件中，随着时间的推移，代码和注释很可能出现不一致。不过，如果文档注释与源代码在同一个文件中，就可以很容易地同时修改源代码和注释，然后重新运行 javadoc。

4.10.1 注释的插入

javadoc 实用工具从下面几项中抽取信息：

- 模块；
- 包；
- 公共类与接口；
- 公共的和受保护的字段；
- 公共的和受保护的构造器及方法。

第 5 章中将介绍受保护特性，第 6 章中将介绍接口，模块在卷Ⅱ的第 9 章介绍。

可以（而且应该）为以上各个特性编写注释。各个注释放置在所描述特性的前面。注释以 `/**` 开始，并以 `*/` 结束。

每个 `/** . . . */` 文档注释包含标记以及之后紧跟着的自由格式文本（free-form text）。标记以 @ 开始，如 @since 或 @param。

自由格式文本的第一个句子应该是一个概要陈述。javadoc 工具自动地将这些句子抽取出生成概要页。

在自由格式文本中，可以使用 HTML 修饰符，例如，用于强调的 `...`、用于着重强调的 `...`、用于项目符号列表的 `/` 以及用于包含图像的 `` 等。要键入等宽代码，需要使用 `{@code ... }` 而不是 `<code>...</code>`——这样一来，就不用操心对代码中的 `<` 字符转义了。

注释：如果文档中有到其他文件的链接，如图像文件（例如，图表或用户界面组件的图像），就应该将这些文件放到包含源文件的目录下的一个子目录 `doc-files` 中。javadoc 工具将从源目录将 `doc-files` 目录及其内容复制到文档目录中。在链接中需要使用 `doc-files` 目录，例如 ``。

4.10.2 类注释

类注释必须放在 `import` 语句之后，`class` 定义之前。

下面是一个类注释的例子：

```
/**
 * A {@code 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都添加 `*`，例如，以下注释同样是合法的：

```
/*
 * A <code>Card</code> 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).
 */
```

不过，大部分 IDE 会自动提供星号，而且换行改变时，还会重新放置星号。

4.10.3 方法注释

每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`

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

- `@return description`

这个标记将给当前方法添加“returns”（返回）部分。这个描述可以跨多行，并且可以使用 HTML 标记。

- `@throws class description`

这个标记将添加一个注释，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7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 means 10%)
 * @return the amount of the raise
 */
public double raiseSalary(double byPercent)
{
    double raise = salary * byPercent / 100;
    salary += raise;
    return raise;
}
```

4.10.4 字段注释

只需要对公共字段（通常指的是静态常量）增加文档注释。例如，

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

4.10.5 通用注释

标记 `@since text` 会建立一个“since”（始于）条目。`text`（文本）可以是对引入这个特性的版本的描述。例如，`@since 1.7.1`。

类文档注释中可以使用下面的标记：

- `@author name`

这个标记将建立一个“author”（作者）条目。可以有多个 `@author` 标记，每个 `@author` 标记对应一个作者。并不是非得使用这个标记，你的版本控制系统能够更好地跟踪作者。

- `@version text`

这个标记将建立一个“version”（版本）条目。这里的 `text` 可以是对当前版本的任何描述。

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

标记 `@see reference` 将在“see also”（参见）部分增加一个超链接。它可以用于类中，也可以用于方法中。这里的 `reference`（引用）可以有以下选择：

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

第一种情况是最有用的。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)
```

会建立一个超链接，链接到 com.horstmann.corejava.Employee 类的 raiseSalary(double) 方法。可以省略包名，甚至把包名和类名都省去，这样一来，这会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 编译器自身可以熟练地确定句点在分隔包、子包、类、内部类以及方法和变量时的不同含义。但是 javadoc 工具就没有这么聪明了，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)，这会显示为链接锚 (link anchor)。如果省略了标签，则用户看到的锚就是目标代码名或 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，可以在任何文档注释中放置指向其他类或方法的超链接。可以在注释中的任何位置插入一个形式如下的特殊标记：

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

这里的特性描述规则与 @see 标记的规则相同。

最后，在 Java 9 中，还可以使用 {@index entry} 标记为搜索框增加一个条目。

4.10.6 包注释

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. 提供一个名为 package-info.java 的 Java 文件。这个文件必须包含一个初始的 Javadoc 注释，以 /** 和 */ 界定，后面是一个 package 语句。它不能包含更多的代码或注释。
2. 提供一个名为 package.html 的 HTML 文件，抽取标记 <body>...</body> 之间的所有文本。

4.10.7 注释提取

在这里，假设你希望 HTML 文件将放在名为 docDirectory 的目录下。执行以下步骤：

1. 切换到源文件目录，其中包含想要生成文档的源文件。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（如果提供 overview.html 文件的话，这就是这个文件所在的目录）。

2. 如果是一个包，应该运行命令：

```
javadoc -d docDirectory nameOfPackage
```

或者，如果要为多个包生成文档，运行：

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

如果你的文件在无名包中，则应该运行：

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

如果省略了 -d *docDirectory* 选项，HTML 文件就会提取到当前目录下。这样可能很混乱，因此我不提倡这种做法。

可以使用很多命令行选项对 javadoc 程序进行微调。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://docs.oracle.com/javase/9/docs/api *.java
```

那么，所有的标准类库的类都会自动地链接到 Oracle 网站的文档。

如果使用 -linksouce 选项，那么每个源文件将会转换为 HTML（不对代码着色，但包含行号），并且每个类和方法名将变为指向源代码的超链接。

还可以为所有源文件提供一个概要注释。把它放在一个类似 overview.html 的文件中，运行 javadoc 工具，并提供命令行选项 -overview *filename*。将抽取标记 <body>... </body> 之间的所有文本。当用户从导航栏中选择“Overview”时，就会显示这些内容。

有关其他的选项，请查阅 javadoc 工具的联机文档 <https://docs.oracle.com/javase/9/javadoc/javadoc.htm>。

4.11 类设计技巧

我们不会面面俱到，也不希望过于沉闷，所以在这一章结束之前再简单地介绍几点技巧。应用这些技巧可以使你设计的类更能得到专业 OOP 圈子的认可。

1. 一定要保证数据私有。

这是最重要的；绝对不要破坏封装性。有时候，可能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字段的私有性。很多惨痛的教训告诉我们，数据的表示形式很可能改变，但它们的使用方式却不会经常变化。当数据保持私有时，表示形式的变化不会对类的使用者产生影响，而且也更容易检测 bug。

2. 一定要初始化数据。

Java 不会为你初始化局部变量，但是会对对象的实例字段进行初始化。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变量，可以提供默认值，也可以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类型。

其想法是要用其他的类，而不是使用多个相关的基本类型。这样会使类更易于理解，也更易于修改。例如，可以用一个名为 Address 的新类替换一个 Customer 类中的以下实例字段：

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

这样一来，可以很容易地处理地址的变化，例如，可能需要处理国际地址。

4. 不是所有的字段都需要单独的字段访问器和更改器。

你可能需要获得或设置员工的工资。而一旦构造了员工对象，肯定不需要更改雇用日期。另外，在对象中，常常包含一些不希望别人获得或设置的实例字段，例如，Address类中的州缩写数组。

5. 分解有过多职责的类。

这样说似乎有点含糊，究竟多少算是“过多”？每个人的看法都不同。但是，如果明显地可以将一个复杂的类分解成两个概念上更为简单的类，就应该进行分解。（但另一方面，也不要走极端。如果设计 10 个类，每个类只有一个方法，显然就有些矫枉过正了。）

下面是一个反面的设计示例。

```
public class CardDeck // bad design
{
    private int[] value;
    private int[] suit;

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

实际上，这个类实现了两个独立的概念：一副牌（包含 shuffle 方法和 draw 方法）和一张牌（包含查看面值和花色的方法）。最好引入一个表示一张牌的 Card 类。现在有两个类，每个类分别完成自己的职责：

```
public class CardDeck
{
    private Card[] cards;

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

public class Card
{
    private int value;
    private int suit;

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

6. 类名和方法名要能够体现它们的职责。

变量应该有一个能够反映其含义的名字，类似地，类也应该如此（在标准类库中，确实存在着一些含义不明确的例子，如 `Date` 类实际上是一个描述时间的类）。

对此有一个很好的惯例：类名应当是一个名词（`Order`），或者是前面有形容词修饰的名词（`RushOrder`），或者是有动名词（有“-ing”后缀）修饰的名词（例如，`BillingAddress`）。对于方法来说，要遵循标准惯例：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`）。

7. 优先使用不可变的类。

`LocalDate` 类以及 `java.time` 包中的其他类是不可变的——没有方法能修改对象的状态。类似 `plusDays` 的方法并不会更改对象，而是会返回状态已修改的新对象。

更改对象的问题在于，如果多个线程试图同时更新一个对象，就会发生并发更改，其结果是不可预料的。如果类是不可变的，就可以安全地在多个线程间共享其对象。

因此，要尽可能让类是不可变的，这是一个很好的想法。对于表示值的类，如一个字符串或一个时间点，这尤其容易。计算会生成新值，而不是更新原来的值。

当然，并不是所有类都应当是不可变的。如果员工加薪时让 `raiseSalary` 方法返回一个新的 `Employee` 对象，这会很奇怪。

本章介绍了有关对象和类的基础知识，这使得 Java 可以作为一种“基于对象”的语言。要真正做到面向对象，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。