第2章　面向对象编程

本章的主题

- 2.1　使用对象——第54页
- 2.2　实现类——第58页
- 2.3　构造对象——第63页
- 2.4　静态变量和方法——第69页
- 2.5　包——第73页
- 2.6　嵌套类——第80页
- 2.7　文档注释——第86页
- 练习——第91页

Chapter 2

在面向对象编程中，工作是通过对象间的协作完成的，这些对象的行为是由它们所属的类定义的。Java是第一个完全拥抱面向对象编程的主流编程语言。正如你已经看到的，在Java中，所有方法必须在类中声明；除了几个基本类型外，所有的值都是对象。在本章中，你将学习如何实现自己的类和方法。

本章要点：

1. 更改器方法改变对象的状态；存取器方法不改变对象的状态。
2. 在Java中，变量不持有对象；它们只有对象的引用。
3. 实例变量和方法实现是在类的内部声明的。
4. 实例方法是通过对象调用的，通过`this`引用可访问该对象。
5. 构造函数与类名相同。一个类可以有多个（重载）构造函数。
6. 静态变量不属于任何对象。静态方法不是通过对象调用的。
7. 类是按包的形式来组织的。使用`import`声明，这样程序中就不必使用包名。
8. 类可以嵌套在其他类中。
9. 内部类是非静态嵌套类。它的实例有个外部类对象的引用，这个外部类构建了内部类。
10. `javadoc`工具处理源代码文件，根据声明和程序员提供的注释产生HTML文件。

2.1 使用对象

早期，对象被发明前，通过调用函数写程序。当你调用一个函数时，它返回一个你将使用的结果，你无须关心该结果是如何计算的。函数有个重要好处：它们允许共享工作。你可以调用一个其他人写的函数，而无须知道该函数是如何执行的。

对象添加了另一个维度。每个对象可以有自己的状态。状态影响你调用方法时得到的结果。例如，如果in是个Scanner对象并且你调用了in.next()，则该对象会记得你之前读取的内容，并将下一个输入内容给你。

当你使用其他人实现的对象并在对象上调用方法时，你不需要知道底层是怎么做的。这个原则被称为封装，它是面向对象编程的一个关键概念。

某种情况下，你可能想给其他程序员提供可使用的对象，使得别人可以共享你的工作成果。在Java中，你提供类—— 一种创建和使用相同行为的机制。

考虑这种常见任务：日历日期的处理。日历有些凌乱，有长度不等的月份和润年，更不用提润秒了。让解决这些烦琐细节的专家提供一个其他程序员可以使用的实现类，是很有意义的。这种情况下，自然产生对象。日期就是一个对象，它的方法提供了这样的信息，例如，“今天是周几”和“明天是几号”。

在Java中，理解日期计算的专家提供日期类和与日期相关的其他类，例如工作日。如果你想计算日期，使用这些类中的某个类创建日期对象，并调用方法，例如，产生周几或明天日期的方法。

我们很少想了解日期计算的细节，但是你可能是其他领域的专家。要让其他程序员能共享你的知识，你可以提供类。即使不是为了让其他程序员使用你的知识，你也会发现在工作中使用类非常有用，这样你的程序结构比较清晰。

在学习如何声明自己的类之前，我们看几个使用对象的重要例子。

给定月份和年，UNIX程序cal按照如下类似的格式输出日历：

```
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
```

如何实现这样的程序？使用标准Java类库，你可以使用LocalDate类表示非特定地区的日期。我们需要该类的一个对象表示指定月的第一天。下面就是如何获得LocalDate对象的例子：

```
LocalDate date = LocalDate.of(year, month, 1);
```

要获取日期的下一天，你可以调用date.plusDays(1)。结果就是在原来对象的基础上加一后新构造的LocalDate对象。在我们的应用中，只是简单将结果重新赋值给date变量：

```
date = date.plusDays(1);
```

使用方法获取关于日期的信息，例如它所在的月份。我们需要那些信息，当我们还在同一个月份时，可以一直打印。

```
while (date.getMonthValue() == month) {
    System.out.printf("%4d", date.getDayOfMonth());
    date = date.plusDays(1);
    ...
}
```

另一个方法返回date是一周中的第几天。

```
DayOfWeek weekday = date.getDayOfWeek();
```

返回值是另一个类DayOfWeek的对象。计算出给定月份的第一天是周几，这样我们便可以在输出时留空白，因此我们需要知道星期几的数字值。getValue方法可以返回数字：

```
int value = weekday.getValue();
for (int i = 1; i < value; i++)
    System.out.print(" ");
```

getValue方法遵从国际习惯，周末在每周的尾部，1代表周一，2代表周二，依此类推。周日是7。

注意：你可以将方法调用连接起来，就像这样：

```
int value = date.getDayOfWeek().getValue();
```

第一个方法适用于date对象，它返回DayOfWeek对象。然后在返回的对象上调用getValue方法。

在本书配套的代码中你会发现完成的程序。解决打印日历的问题比较简单，因为LocalDate类的设计者给我们提供了一组有用的方法。在本章中，你会学习如何给自己的类添加实现方法。

2.1.1 Accessor（访问器）和Mutator（更改器）方法

重新思考一下方法调用date.plusDays(1)。LocalDate类的设计者们有两种方式实现plusDays方法。他们可以改变date对象的状态，不返回结果。或者，他们可以不改变date对象但是返回一个新构造的LocalDate对象。正如你看到的那样，他们选择了后者。

如果一个方法改变了调用它的对象，我们就说这是一个*更改器*方法。如果方法不改变调用自己的对象，它就是*访问器*方法。LocalDate类的plusDays方法就是*访问器*方法。

事实上，LocalDate类的所有方法都是访问器方法。这种情况越来越普遍，因为修改可能会有风险，特别是在同一个对象上同时计算时。现在，绝大多数计算机有多个处理单元，安全的并发访问成为严重问题。解决这种问题的一种方式就是只提供访问器方法，使得对象都是*不可变的*。

但是依然有很多情况下，修改对象更适合。ArrayList类的add方法就是更改器方法的典型例子。调用add后，数组列表对象被改变了。

```
ArrayList<String> friends = new ArrayList<>();
     // friends为空
friends.add("Peter");
     // friends现在有一个元素
```

2.1.2 对象引用

在一些编程语言（例如C++）中，变量实际上可以持有对象——也就是，组成对象状态的比特。在Java中，却并非如此。变量只能持有对象的*引用*。实际的对象在其他地方，引用是与实现相关的一种定位对象的方式（参看图2-1）。

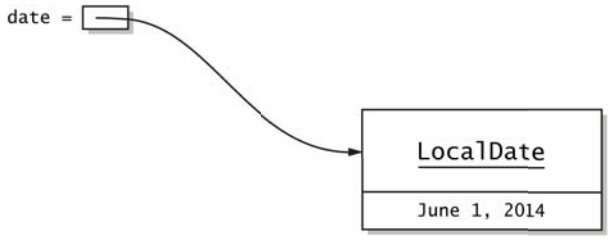

图2-1 一个对象引用

注意：除了非常安全外，引用就像C和C++中的指针。在C和C++中，你可以修改指针并使用它们重写任意位置内存。而Java引用，你只能访问特定对象。

当你用持有对象引用的变量给另一个变量赋值时，两个引用指向同一个对象。

```
ArrayList<String> people = friends;
    // 现在people和friends引用同一个对象
```

如果你修改了共享对象，则两个引用都可以看见修改效果。想一下如下调用：

```
people.add("Paul");
```

现在，数组列表people有两个元素，并且friends变量也一样有两个元素（参看图2-2）。当然，从技术上讲，不是people有两个元素。毕竟，people不是对象。它只是有两个元素的数组列表对象的引用。

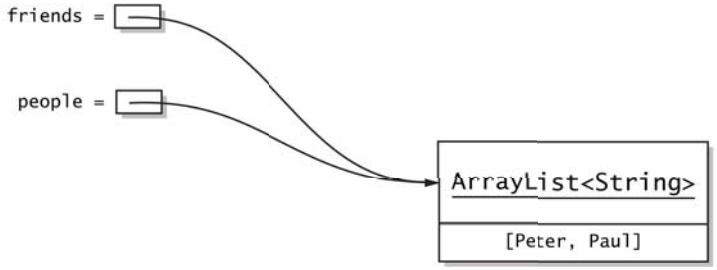

图2-2 同一个对象的两个引用

大部分时候，共享对象是高效和方便的，但是你必须意识到，通过任何一个引用，都可能修改共享对象。

但是，如果一个类没有更改器方法（例如String类或LocalDate类），你就无

须担心。因为没有人可以改变这样的对象，你可以大量分发对象的引用。

将对象变量设置为特殊值null，对象变量可能没有引用任何对象。

```
LocalDate date = null; // 现在date没有引用任何对象
```

如果你还没有对象供date引用，或者你想表明一种特殊情况，例如未知的日期，这样做是有用的。

警告：当null值出乎意料地出现时，它可能是危险的。在null上调用方法会导致NullPointerException（NullPointerException真应该称为NullReferenceException）。正因如此，不推荐使用null作为可选值。使用可选类型代替（参看第8章）。

最后，再看一下赋值：

```
date = LocalDate.of(year, month, 1);
date = date.plusDays(1);
```

第一个赋值后，date引用月的第一天。调用plusDays产生新的LocalDate对象，并且在第二次赋值后，date变量引用新的对象。第一个对象会发生什么？

第一个对象没有引用了，因此它也不再被需要了。最终，垃圾回收器（garbage collector）会回收该内存，并使之重用。在Java中，该过程是完全自动的，程序员从不需要担心释放内存。

2.2 实现类

现在，我们来实现自己的类。为展示各种语言规则，我使用经典的Employee（员工）类例子。Employee有name和salary。在这个例子中，name不能改变，但是有时员工salary可以上调。

2.2.1 实例变量

从员工对象的描述看，你可以看到这样一个对象的状态是由两个值描述的：name和salary。在Java中，你可以使用实例变量描述对象的状态。在类中声明实例变量的方式如下：

```
public class Employee {
    private String name;
    private double salary;
    ...
}
```

这意味着每个Employee类的对象或实例都有这两个变量。

在Java中，实例变量通常声明为private。这意味着只有同类的方法才可以访问该变量。为什么需要这种保护，这是有原因的：你能控制程序的哪个部分可以修改变量，你能决定在何时修改内部展现。例如，你想在数据库存储员工信息并且只留下主键在对象中。只要你重新实现的方法如同以前一样工作，类的使用者便不会关心内部实现。

2.2.2 方法头

现在，我们转到Employee类方法的实现上。当你声明一个方法时，你提供方法名、类型和参数名称以及返回类型，就像这样：

```
public void raiseSalary(double byPercent)
```

方法接收一个类型为double的参数并且不返回任何值，正如返回类型为void所表明的那样。

getName方法的签名不同：

```
public String getName()
```

该方法没有参数并且返回一个字符串。

注意：绝大多数方法均声明为public，这意味着任何人都可以调用该方法。有时，辅助方法声明为private，这样限制了该方法只能被同类中的其他方法使用。如果方法与类用户无关，则应该将方法声明为private（特别是如果方法依赖于实现细节）。如果实现改变了，你可以安全改变或删除private方法。

2.2.3 方法体

紧跟方法头之后是你提供的方法体：

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

如果方法会产生一个值，则使用关键字return：

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

将方法声明放在类声明中：

```
public class Employee {
   private String name;
   private double salary;

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

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

2.2.4 实例方法调用

考虑如下方法调用的例子：

```
fred.raiseSalary(5);
```

在该调用中，参数5用来初始化参数变量byPercent，相当于赋值：

```
double byPercent = 5;
```

然后发生下面的行为：

```
double raise = fred.salary * byPercent / 100;
fred.salary += raise;
```

注意，实例变量salary应用在被调方法的实例上。

与你在第1章末尾看到的方法不同，诸如raiseSalary这样的方法是在类的实例上运行。因此，这样的方法也被称为*实例方法*。在Java中，所有没有被声明为static的方法都是实例方法。

正如你看到的那样，两个值传递给raiseSalary方法：一个是调用该方法的对象引用，另一个是调用参数。从技术上讲，这两个都是方法参数，但是在Java中，和其他面向对象的编程语言一样，第一个参数扮演特殊角色。有时，也被称为方法调用的*接收者*（receiver）。

2.2.5 this引用

在对象上调用方法时，this引用指向该对象。如果你喜欢，你可以在方法的实现中使用this引用：

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

有些程序员偏好这种风格，因为它清晰地区分了局部变量和实例变量——现在很明显，raise是局部变量，而salary是实例变量。

当你不想给参数变量起不同的名称时，使用this引用非常普遍。例如：

```
public void setSalary(double salary) {
    this.salary = salary;
}
```

当实例变量和局部变量同名时，非限定名（例如salary）表示局部变量，而this.salary是实例变量。

注意：在某些编程语言中，实例变量以某种方式修饰，例如，_name和_salary。在Java中，这样也是合法的，但是不常用。

注意：如果喜欢，你甚至可以将this声明为方法（构造函数除外）的参数：

```
public void setSalary(Employee this, double salary) {
    this.salary = salary;
}
```

但是，这种语法非常少用。这种方式下， 你可以给方法的接收者做注解——参看第11章。

2.2.6 值调用

当你将对象传递给方法时，方法获得该对象引用的拷贝。通过this引用，它可以访问或修改参数对象，例如：

```
public class EvilManager {
   private Random generator;
   ...
   public void giveRandomRaise(Employee e) {
       double percentage = 10 * generator.nextGaussian();
       e.raiseSalary(percentage);
   }
}
```

考虑这个调用：

```
boss.giveRandomRaise(fred);
```

引用fred被复制到参数变量e（参看图2-3）。该方法修改了被两个引用共享的对象。

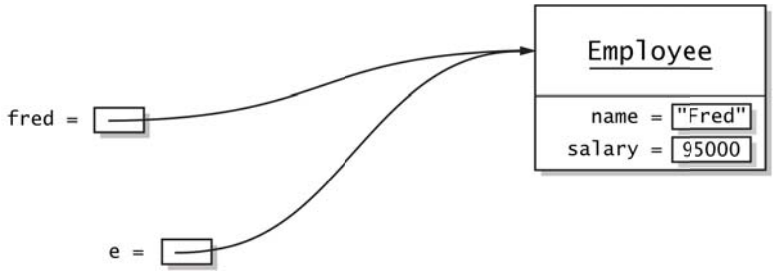

图 2-3 持有对象引用拷贝的参数变量

在Java中，你不能写出更新基本类型参数的方法。试图增加double类型值的方法将无法生效：

```
public void increaseRandomly(double x) { // 无法工作
    double amount = x * generator.nextDouble();
    x += amount;
}
```

如果你调用

```
boss.increaseRandomly(sales);
```

sales被复制进x。然后x增加了，但是这并不改变sales。然后，参数变量离开它的作用域，increaseRandomly方法就没有留下有效作用。

正因如此，不可能写出一个方法，该方法可以将对象引用修改为其他东西。例如，如下方法不会像它假设的那样工作：

```
public class EvilManager {
    ...
    public void replaceWithZombie(Employee e) {
        e = new Employee("", 0);
    }
}
```

如下调用中

```
boss.replaceWithZombie(fred);
```

引用fred被复制进变量e，然后变量e被设置为不同的引用。当方法退出时，e离开它的作用域。fred一点都没改变。

注意：有些人说Java使用“引用调用”。正如你在第二个例子中看到的，事实并非如此。在支持引用调用的语言中，方法能替换传递给它的变量的内容。在Java中，所有参数——对象引用以及基本类型值——都是值传递。

2.3 构造对象

距离完成Employee类还有一步：我们需要提供一个构造函数，细节见如下章节。

2.3.1 实现构造函数

声明构造函数与声明方法类似。但是，构造函数的名称与类名称相同，并且不返回任何类型。

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

注意：该构造函数是公有的。有个私有的构造函数也是有用的。例如，LocalDate类没有公有构造函数。但是，该类的用户可以通过“工厂方法”获得对象，例如，now和of。这些方法调用私有构造函数。

警告：如果你意外指定返回void类型

```
public void Employee(String name, double salary)
```

那么，你声明的是名称为Employee的方法，不是构造函数！

当你使用new运算符时，会执行构造函数。例如，如下表达式

```
new Employee("James Bond", 500000)
```

分配一个Employee对象并调用构造函数的方法体，构造函数将实例变量设置为提供的参数。

new运算符返回一个构造好的对象的引用。通常你会将该引用保存到变量中：

```
Employee james = new Employee("James Bond", 500000);
```

或者传递给方法：

```
ArrayList<Employee> staff = new ArrayList<>();
staff.add(new Employee("James Bond", 500000));
```

2.3.2 重载

你可以提供多个版本的构造函数。例如，如果你想使得构造无名字的员工变得更

容易，则提供第二个构造函数，它只接受salary参数。

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

现在Employee类有两个构造函数。调用哪个构造函数取决于参数。

```
Employee james = new Employee("James Bond", 500000);
    // 调用Employee(String, double)构造函数
Employee anonymous = new Employee(40000);
    // 调用Employee(double)构造函数
```

这种情况下，我们说构造函数是重载的。

注意：如果有多个名称相同但是参数不同的方法版本，那么这个方法就是重载的。例如，println方法有参数为int、double、Sting等的多个重载版本。因为你无法选择构造函数的名称，所以重载构造函数很常见。

2.3.3 调用另一个构造函数

当有多个构造函数时，它们通常有些共有的工作，而且最好不要重复那段共有的工作代码。常常可能将共有初始化代码放在一个构造函数中。

你可以在构造函数中调用另一个构造函数，但是只能作为构造函数方法体的第一条语句。令人奇怪的是，调用不使用构造函数的名称，而是使用关键字this：

```
public Employee(double salary) {
    this("", salary); // 调用构造函数Employee(String, double)
    // 后面是其他声明
}
```

注意：这里的this不是正被构造对象的引用。相反，它是一种特殊语法，只用在同类的另一个构造函数调用中。

2.3.4 默认初始化

如果你在构造函数中没有显式地设置实例变量，那么系统会自动给实例变量赋个默认值：数字为0，boolean为假，对象引用为null。

例如，你可以为一个没有薪水的实习生提供一个构造函数。

```
public Employee(String name) {
    //自动设置salary为0
    this.name = name;
}
```

注意：就这点而言，实例变量与局部变量非常不同。回想一下，你必须总是显式地初始化局部变量。

对于数字，使用0初始化通常较方便。但是对于对象引用，这样做是常见的错误源头。假设我们没有在构造函数Employee(double)中将变量name设置为空字符串：

```
public Employee(double salary) {
    // 自动设置name为null
    this.salary = salary;
}
```

如果任何人调用了getName方法，他们会得到一个可能不是他们期待的null引用。诸如这样的情况

```
if (e.getName().equals("James Bond"))
```

就会导致空指针异常。

2.3.5 实例变量初始化

你可以为任何实例变量指定初始值，像这样：

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

这种初始化发生在对象分配之后，构造函数运行之前。因此，初始值在所有构造

函数都是可见的。当然，某些构造函数可能选择重写实例变量。

除了能在声明时初始化实例变量外，你也可以在类的声明中包含任何*初始化块*。

```
public class Employee() {
    private String name = "";
    private int id;
    private double salary;

    { // 初始化块
        Random generator = new Random();
        id = 1 + generator.nextInt(1_000_000);
    }

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

注意：这不是重用的特性。大多数程序员将较长的初始化块放在辅助方法中，并在构造函数中调用该方法。

实例变量初始化和初始化块以它们在类中出现的先后顺序执行，并且在构造函数方法体之前执行。

2.3.6 final实例变量

你可以将实例变量声明为`final`。这样的变量必须在所有的构造函数末尾初始化。之后，该变量也许不会再次被修改。例如，`Employee`类的`name`变量可能被声明为`final`，因为对象构造之后，`name`再也不会改变——没有`setName`方法。

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

注意：当使用可修改对象的引用时，final修饰符只是声明该引用永不改变。修改对象自身是完全合法的。

```
public class Person {
    private final ArrayList<Person> friends = new ArrayList<>();
        // 可以给该数组列表添加元素
    ...
}
```

方法可能修改friends引用的数组列表，但是它们不能用其他对象替代。特别是，friends不能变为null。

2.3.7 无参构造函数

许多类包含一个没有参数的构造函数，该构造函数创建一个状态为适当默认值的对象。例如，下面是Employee类的一个无参构造函数：

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

如同给每个贫穷的被告指定一名公共辩护人一样，系统会自动给没有构造函数的类指定一个什么也不做的无参构造函数。所有的实例变量都是它们的默认值(0、false或者null)，除非它们被明确初始化。

因此，所有的类都至少有一个构造函数。

注意：如果一个类已经有一个构造函数，则系统不会自动再给它一个无参的构造函数。如果你提供一个构造函数并且还想要无参的构造函数，则你必须自己写一个构造函数。

注意：在前面的小节中，你看到当构造一个对象时，发生了什么。在一些编程语言中，尤其是C++，通常要指定对象析构时发生什么。当垃圾回收器回收对象时，Java也有一种“终结”对象的机制。但是它发生的时间不可预测，所以你

不应该使用。但是，正如你将会在第5章看到的，有一种关闭资源（例如文件资源）的机制。

2.4 静态变量和方法

在你看到的所有示例程序中，main方法有static修饰符标签。接下来你将会学到static修饰符意味着什么。

2.4.1 静态变量

如果你在类中将变量声明为static，那么该变量是属于类的，而不是对象，每个类只有一个。相比之下，每个对象都有一份实例变量的拷贝。例如，假设我们想给每个员工一个不同的员工ID号。那么，我们可以共享最后一个给出的ID。

```
public class Employee {
    private static int lastId = 0;
    private int id;
    ...
    public Employee() {
        lastId++;
        id = lastId;
    }
}
```

每个Employee对象都有自己的实例变量id，但是只有一个lastId变量，它属于该类，而不属于该类的任何特殊实例。

当构造一个新的Employee对象时，共享的lastId变量自增并且将id设置为lastId的最新值。因此，每个Employee对象都得到不同的id值。

警告：如果Employee对象在多线程中并发构造对象，则该代码无法达到预期效果，第10章会展示如何解决并发中的问题。

注意：你可能在想为什么属于类而不是属于单个实例的变量被称为“静态的”。该词来自C++中一个无意义的词，C++没有想出更适合的词，因而从C语言中借来了一个不相关的关键词。更合适的描述性术语应该是“class variable”。

2.4.2 静态常量

可修改的静态常量很少，但是静态常量（也就是static final变量）相当普遍。例如，Math类声明了一个静态常量：

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

你可以在自己的程序中以Math.PI访问该常量。

没有static关键字，PI就成为Math类的实例变量。也就是，你需要一个该类的对象才能访问PI，并且每个Math对象都有一份自己的PI拷贝。

下面就是一个static final变量的例子，该变量是个对象，而不是数字。每次当你想要一个随机数时构造一个新的随机数产生器，这样做既浪费资源同时也不安全。最好在类的实例中共享一个随机数产生器。

```
public class Employee {
    private static final Random generator = new Random();
    private int id;
    ...
    public Employee() {
        id = 1 + generator.nextInt(1_000_000);
    }
}
```

另一个静态常量的例子是System.out。它在System类中像这样声明：

```
public class System {
    public static final PrintStream out;
```

```
    ...
}
```

警告：虽然在System类中out被声明为final，但还是有个setOut方法将System.out设置到不同的流上。该方法是“本地”方法，不是在Java中实现的，它能绕过Java语言的访问控制机制。这种罕见情况来自早期Java，并且你不会在其他地方遇到。

2.4.3 静态初始块

在前面，静态变量是在其声明时初始化的。有时，你需要额外的初始化工作。你可以将其放入*静态初始化块*。

```
public class CreditCardForm {
    private static final ArrayList<Integer> expirationYear = new ArrayList<>();
    static {
        //将下一个20年添加到数组列表
        int year = LocalDate.now().getYear();
        for (int i = year; i <= year + 20; i++) {
            expirationYear.add(i);
        }
    }
    ...
}
```

当类第一次加载时，执行静态初始化。就像实例变量那样，静态变量默认值为0、false或者null（除非你明确地设置了变量值）。所有的静态变量初始化和静态初始化块以它们在类声明中出现的顺序执行。

2.4.4 静态方法

静态方法是指可以不用运行在对象上的方法。例如，Math类的pow方法就是静态方法，表达式

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

计算幂x^a。它不使用任何Math对象来执行任务。

正如你在第1章中看到的，静态方法以static修饰符声明：

```
public class Math {
    public static double pow(double base, double exponent) {
        ...
    }
}
```

为什么不把pow放到实例方法？它不能成为double的实例方法。因为在Java中，基本类型不是类。可以使它成为Math类的实例方法，但是为了调用该方法，你将需要构造Math对象。

给不属于你的类提供额外功能是静态方法存在的另一个普遍原因。例如，有个方法能产生给定范围内的随机数，这样不是很好吗？虽然你不能给标准类库中的Random类添加方法，但是你能提供静态方法：

```
public class RandomNumbers {
    public static int nextInt(Random generator, int low, int high) {
        return low + generator.nextInt(high - low + 1);
    }
}
```

调用该方法：

```
int dieToss = RandomNumbers.nextInt(gen, 1, 6);
```

注意：可以在对象上调用静态方法。例如，代替调用LocalDate.now()来获取今天的日期，你可以在LocalDate类的对象date上调用date.now()。但是这样做没多大意义。now方法不是根据date对象计算结果。大多数Java程序员会认为这不是一种好的编程风格。

因为静态方法并不运行在对象上，所以你不能在静态方法中访问实例变量。但是，静态方法可以访问本类的静态变量。例如，在RandomNumbers.nextInt方法中，我们可以将产生的随机数放到静态变量中：

```
public class RandomNumbers {
    private static Random generator = new Random();
```

```
    public static int nextInt(int low, int high) {
        return low + generator.nextInt(high - low + 1);
        //这里可以访问静态generator变量
    }
}
```

2.4.5 工厂方法

静态方法常见的使用就是*工厂方法*，也就是返回一个类的新实例的静态方法。例如，NumberFormat类使用工厂方法产生各种格式的formatter对象。

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

为什么不用构造函数代替？区分两个构造函数的唯一方法是它们的参数类型。因此，你不能有两个无参数的构造函数。

此外，构造函数new NumberFormat(...)产生一个NumberFormat。工厂方法能返回一个子类的对象。事实上，这些工厂方法可以返回DecimalFormat类的实例（参看第4章，了解更多关于子类的信息）。

工厂方法也能返回共享对象，无须每次构建一个新对象。例如，调用Collections.emptyList()返回一个共享的不可修改的空列表。

2.5 包

在Java中，你将相关的类放到一个包中，包便于组织你的工作并且可以将其他人提供的代码类库和自己的代码分隔开。正如你看到的，标准的Java类库分布在很多包中，包括java.lang、java.util、java.math等。

使用包的主要原因是保证类名的唯一性。假设两个程序都提出好点子提供一个Element类（事实上，仅在Java API就至少有5位开发者有那样的想法）。只要这些开发者将他们所开发的类放到不同的包中，就没有冲突。

下面你将学习如何使用包。

2.5.1 包的声明

包的名称是以点号分隔的标识符，例如java.util.regex。

为了确保包名的唯一性，使用倒着写的因特网域名是个好主意（已知因特网域名是唯一的）。例如，我拥有域名horstmann.com。我的工程，我使用诸如这样的包名：com.horstmann.corejava。该规则的主要例外是标准的Java类库，标准类库的包名以java或者javax开始。

注意：在Java中，包不能嵌套。例如，包java.util和java.util.regex彼此之间没有关系。两个包都是各自独立类的集合。

要将一个类放到一个包中，你需要将包的声明作为源文件的第一个声明：

```
package com.horstmann.corejava;

public class Employee {
    ...
}
```

现在Employee类在包com.horstmann.corejava中，它的全限定名为com.horstmann.corejava.Employee。

也存在一个没有名称的默认包，你可以在简单程序中使用。将类添加到默认包中，不用提供包声明。但是，不推荐使用默认包。

当从文件系统读取类文件时，路径名称必须匹配包名称。例如文件Employee.class必须在子目录com/horstmann/corejava。

如果你以同样的方式组织源文件并且从包含最初包名的目录编译，那么class（类）文件自动放入正确的位置。假设EmployeeDemo类使用Employee对象，并且你这样编译EmployeeDemo：

```
javac com/horstmann/corejava/EmployeeDemo.java
```

编译器产生class文件com/horstmann/corejava/EmployeeDemo.class和com/horstmann/corejava/Employee.class。通过指定全限定的类名运行程序：

```
java com.horstmann.corejava.EmployeeDemo
```

警告：如果源文件不在匹配包名的子目录中，javac编译器会产生class文件，但不会报错。但是你需要将class文件放到恰当的位置。这样做相当混乱——参看练习12。

提示：运行javac时带上-d选项是个好主意。这样class文件会产生在单独的目录中，不会搞乱源代码树，并且class文件也有正确的子目录结构。

2.5.2 类路径

你可以将class文件放到一个或多个被称为JAR文件的归档文件中，而不是将class文件存储在文件系统。你可以使用jar工具制作归档文件，jar工具是JDK的一部分。它的命令行参数与UNIX上的tar程序类似。

```
jar cvf library.jar com/mycompany/*.class
```

包类库常常这样做。

注意：默认情况下，JAR文件使用ZIP格式。还有个选项用来指定另外的压缩模式，被称为“pack200”，目的是更高效地压缩class文件。

提示：你可以使用JAR文件将程序打包，而不仅仅是类库。以如下方式产生JAR文件：

```
jar cvfe program.jar com.mycompany.MainClass com/mycompany/*.class
```

然后这样运行程序：

```
java -jar program.jar
```

当你在项目中使用类库的JAR文件时，你需要通过指定class path告诉编译器和虚拟机这些JAR文件在哪里。class path可以包含：

- 包含class文件的目录（包含匹配包名的子目录）

- JAR文件
- 包含JAR文件的目录

javac和java命令都有-classpath选项，可以缩写为-cp。例如：

```
java -classpath .:../libs/lib1.jar:../libs/lib2.jar com.mycompany.MainClass
```

上面例子中的类路径有三个元素：当前目录（.），以及目录../libs中的两个JAR文件。

注意：在Windows中，使用分号而不是冒号分隔路径元素：

```
java -classpath .;..\libs\lib1.jar;..\libs\lib2.jar com.mycompany.MainClass
```

如果你有许多JAR文件，则将它们全部放在一个目录，然后使用通配符包含所有文件：

```
java -classpath .:../libs/\* com.mycompany.MainClass
```

注意：在UNIX中，禁止使用*以防止shell命令进一步扩展。

警告：javac编译器总是在当前目录寻找文件，但是如果“.”目录在类路径上，则java程序只查看当前目录。如果你没有设置类路径，这不是个问题——默认的类路径由“.”目录组成。如果你设置了类路径并且忘记包含“.”目录，你的程序可以正常编译，但是将不能运行。

设置类路径首选使用-classpath选项。另外一个替代方法是CLASSPATH环境变量。细节取决于你的shell。如果你使用bash，则使用诸如这样的命令：

```
export CLASSPATH=.:/home/username/project/libs/\*
```

在Windows中，这样做：

```
SET CLASSPATH=.;C:\Users\username\project\libs\*
```

警告：你可以设置全局CLASSPATH环境变量（例如，在.bashrc或者Windows控制面板）。但是，当许多程序员忘记了曾经设置过CLASSPATH环境变量并惊讶地发现他们的类无法找到时，就会后悔之前设置过全局CLASSPATH环境变量。

警告：一些人推荐将所有的JAR文件放到jre/lib/ext目录，完全绕开类路径。/jre/lib/ext是个特殊的目录，虚拟机会查阅“已经安装的扩展”。这是极坏的主意，原因有两个。JAR文件放在扩展目录中，手动加载类的代码无法正确工作。此外，相比CLASSPATH，程序员不太可能记得隐蔽的jre/lib/ext目录——当类加载器忽视程序员精心制作的类路径，而从扩展目录中加载了早就被忘记的类时，程序员会抓破脑袋也想不出原因。

2.5.3 包作用域

你已经见过访问修饰符public和private。打上public标签的功能部分可以被任何类使用。private功能部分只能被声明它的类使用。如果你既不指定public也不指定private，该功能部分（也就是类、方法或者变量）就可以被同一个包中的所有方法访问。

包的作用域对于工具类和方法是有用的，这些类和方法被另外一个包中的方法需要但是另外一个包的用户又对其不感兴趣。另一个常见的用例就是测试。你可以将测试类放在同一个包中，同时它们能访问被测试类的内容。

注意：一个源文件可以包含多个类，但是最多只有一个可以声明为public。如果一个源文件有个public类，那么文件名和类名必须匹配。

对于变量，很不幸，包的作用域是默认的。常见的错误是忘记了private修饰符，因此意外使得实例变量可以被整个包访问。这个例子来自java.awt包中的Window类：

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

因为变量warningString不是private的，所以java.awt包中所有类的方法都可以访问它。实际上，除了Window类自己的方法外，没有其他方法访问该变量。因此，似乎可能只是程序员忘记了private修饰符。

因为包是开放的，这就有安全问题了。任何类都可通过提供恰当的包声明将自己添加到包中。

Java实现者通过操控类加载器保护自己免受这样的攻击，因此它不会加载任何全限定名以java开始的类。

如果你想让自己的包也有类似的保护机制，则需要将它们放到封闭的JAR文件中。提供一个manifest，也就是包含条目的文本文件：

```
Name: com/mycompany/util/
Sealed: true
Name: com/mycompany/misc/
Sealed: true
```

然后，像这样运行jar命令：

```
jar cvfm library.jar manifest.txt com/mycompany/*/*.class
```

2.5.4 导入包

import声明让你无须全限定名就可以使用类。例如，当你使用

```
import java.util.Random;
```

之后，可以在代码中用Random代替java.util.Random。

注意：import声明比较方便，但不是必需的。你可以选择放弃所有的import声明，然后在所有地方使用全限定的类名。

```
java.util.Random generator = new java.util.Random();
```

import声明要放在源文件第一个类声明之前，包声明之后。

你可以使用通配符导入一个包的所有类：

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

通配符只能导入类，不能导入包。你不能使用import java.*获得以java开始的所有包。

当你导入多个包时，可能有名称冲突。例如，包java.util和java.sql都包含Date类。假设你导入两个包：

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

如果你的程序不使用Date类，就没有问题。但是如果你涉及Date，并且没有加上包名，编译器就会报错。

这种情况下，你可以导入自己想要的那个具体类：

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

如果两个类你都确实需要，则必须使用其中至少一个类的全限定名。

注意：import声明对程序员很方便。在类文件内部，所有的名称都是全限定名的。

注意：import声明与C和C++中的#include指令完全不同。#include指令为编译器包含了头文件。import不会导致文件重编译。它们只是缩短了名称，就像C++中的using声明。

2.5.5 静态导入

一种import声明形式允许导入静态方法和变量。例如，如果你在源文件的头部添加如下指令：

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

无须类名称作为前缀，你就可以使用Math类的静态方法和静态变量：

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

你也可以导入具体的静态方法或变量：

```
import static java.lang.Math.sqrt;
import static java.lang.Math.PI;
```

注意：正如你将在第3章和第8章看到的那样，对java.util.Comparator和java.util.stream.Collectors，使用静态导入声明很常见。java.util.Comparator和java.util.stream.Collectors提供了大量静态方法。

警告：你不能从默认包中的类导入静态方法或域。

2.6 嵌套类

在上一节中，你看到了如何将类组织成包。还有一种选择，你可以将一个类放到另一个类的内部。这样的类被称为*嵌套类*。这样对于限制可见性或者避免因为导入了一些使用了通用名称（诸如Element、Node或Item之类）的包而产生混乱，是有用的。Java有两种行为稍微有些不同的嵌套类。下面我们介绍这两个嵌套类。

2.6.1 静态嵌套类

考虑一个拥有Item（发货项）的Invoice（发货单）类，每个Item都有description（描述）、quantity（数量）和unit price（单价）。我们可以将Item变成嵌套类：

```
public class Invoice {
    private static class Item { // Item类嵌套在Invoice内
        String description;
        int quantity;
        double unitPrice;

        double price() { return quantity * unitPrice; }
    }

    private ArrayList<Item> items = new ArrayList<>();
```

```
        ...
    }
```

直到2.6.2节我们才会清楚为什么内部类要声明为static。现在，我们先接受这种方式。

Item类没有什么特别的（除了访问控制）。在Invoice中该类是private的，因此，只有Invoice方法能访问它。正因如此，我们不用操心要将内部类的实例变量改为私有变量。

下面是个构建内部类对象方法的例子：

```
public class Invoice {
    ...
    public void addItem(String description, int quantity, double unitPrice) {
        Item newItem = new Item();
        newItem.description = description;
        newItem.quantity = quantity;
        newItem.unitPrice = unitPrice;
        items.add(newItem);
    }
}
```

类可以使得嵌套类成为public类。在那种情况下，就需要使用通常的封装机制。

```
public class Invoice {
    public static class Item { // 一个公有的嵌套类
        private String description;
        private int quantity;
        private double unitPrice;

        public Item(String description, int quantity, double unitPrice) {
            this.description = description;
            this.quantity = quantity;
            this.unitPrice = unitPrice;
        }
        public double price() { return quantity * unitPrice; }
        ...
```

```
    }

    private ArrayList<Item> items = new ArrayList<>();

    public void add(Item item) { items.add(item); }
    ...
}
```

现在任何人通过使用限定名Invoice.Item就能构建Item对象：

```
Invoice.Item newItem = new Invoice.Item("Blackwell Toaster", 2, 19.95);
myInvoice.add(newItem);
```

这样的Invoice.Item类和在其他任何类外部声明一个InvoiceItem类没有任何本质上的区别。嵌套类只是使得Item明显代表的是Invoice中的Item。

2.6.2 内部类

前面你看到了我们将嵌套类声明为static。在本节中，你会看到如果丢弃static修饰符会发生什么。这样的类被称为内部类。

考虑这样一个社会网络，社会中的每个成员都有朋友，而他们的朋友也是社会成员。

```
public class Network {
    public class Member { // Member类是Network类的内部类
        private String name;
        private ArrayList<Member> friends;

        public Member(String name) {
            this.name = name;
            friends = new ArrayList<>();
        }
        ...
    }

    private ArrayList<Member> members;
    ...
}
```

去掉了static修饰符，就会有本质区别。Member对象知道自己属于哪个网络。我们看看这是如何工作的。

首先，这里是向网络添加社会成员的方法：

```
public class Network {
    ...
    public Member enroll(String name) {
        Member newMember = new Member(name);
        members.add(newMember);
        return newMember;
    }
}
```

到目前为止，似乎什么也没发生。我们可以添加一个成员并获取它的引用。

```
Network myFace = new Network();
Network.Member fred = myFace.enroll("fred");
```

现在，假设Fred认为这个网络不再是最热的社交网络，并且他想离开这个网络。

```
fred.leave();
```

这里是leave方法的实现：

```
public class Network {
    public class Member {
        ...
        public void leave() {
            members.remove(this);
        }
    }

    private ArrayList<Member> members;
    ...
}
```

正如你看到的，内部类的方法可以访问它的外部类的实例变量。在这种情况下，它们是外部类对象的实例变量。不流行的myFace网络创建了Member对象fred。

这就是内部类和静态嵌套类的不同之处。每个内部类对象都有自己外部类对象的引用。例如，方法

```
members.remove(this);
```

实际含义为

```
outer.members.remove(this);
```

这里我们用*outer*表示被隐藏的对外部类的引用。

静态嵌套类没有这样的引用（就像静态方法没有this引用）。当嵌套类的实例不需要知道它属于外部类的哪个实例时，使用静态嵌套类。只有当这种信息重要时，才使用内部类。

内部类通过它的外部类实例也能调用外部类的方法。例如，假设外部类有个方法能取消注册成员的方法，那么leave方法可以调用它：

```
public class Network {
    public class Member {
        ...
        public void leave() {
            unenroll(this);
        }
    }

    private ArrayList<Member> members;

    public Member enroll(String name) { ... }
    public void unenroll(Member m) { ... }
    ...
}
```

这种情况下，

```
unenroll(this);
```

实际含义为

```
outer.unenroll(this);
```

2.6.3 内部类的特殊语法

在上一节中，我解释内部类对象的外部类引用时，将其称为*outer*。外部类引用的实际语法有点复杂。表达式

```
OuterClass.this
```

表示外部类引用。例如，你可以将内部类Member的leave方法写成：

```
public void leave() {
    Network.this.members.remove(this);
}
```

这种情况下，Network.this语法不是必需的。仅用members已经内含使用外部类引用。但是有时，你需要明确的外部类引用。这里有个方法，它会检查一个Member对象是否属于某个特定网络：

```
public class Network {
    public class Member {
        ...
        public boolean belongsTo(Network n) {
            return Network.this == n;
        }
    }
}
```

当你构造一个内部类对象时，它记住构造它的外部类对象。在上一节中，通过这样的方法创建一个新的Member对象：

```
public class Network {
    ...
    Member enroll(String name) {
        Member newMember = new Member(name);
        ...
    }
}
```

那是如下声明的快捷方式：

```
Member newMember = this.new Member(name);
```

你可以在外部类的任何实例上调用内部类构造函数：

```
Network.Member wilma = myFace.new Member("Wilma");
```

注意：除了编译时常量外，内部类不能声明静态成员。如果内部类有静态成员，那关于“static.”的含义将会是模糊不清的。它是意味着虚拟机中只有一个实例，还是意味着每个外部对象只有一个实例？Java语言的设计者决定不解决这样的问题。

注意：由于历史偶然，在大家认为虚拟机规范已经完成时，Java语言添加了内部类，因此它们都被翻译成普通类，该类有一个指向外部实例的隐藏实例变量。练习14邀请你探索这种转换。

注意：局部类是内部类的另一种变种，我们会在第3章讨论。

2.7 文档注释

JDK包含一个非常有用的工具javadoc，它能从源代码文件产生HTML文档。事实上，我们在第1章提到的在线API文档就是在标准Java类库源代码上运行javadoc的结果。

如果在源代码中添加以特殊分隔符/**开始的注释，你也可以很容易地生成看起来专业的文档。这是一种非常好的方法，因为它让代码和文档在一个地方。在之前的艰难时代，程序员常常将文档放在一个单独的文件；代码和注释迟早会产生偏离，这只是时间问题。当文档注释和源代码在同一个文件时，很容易更新两者并再次运行javadoc。

2.7.1 插入注释

javadoc工具导出以下项目信息：

- 包

- 公有类和接口
- 公有和保护类型的变量
- 公有和保护类型的构造函数和方法

接口会在第3章介绍，保护类型特性会在第4章介绍。

你可以（并且应该）为每个功能提供注释。每个注释放在它所描述功能的紧上面。注释以/**开始，以*/结束。

所有/** … */文档注释包含自由格式的文本，后面紧跟标签。标签以@开始，例如@author或@param。

自由格式文本的第一句应该是总结声明。javadoc工具自动提取这些语句，产生总结页。

在自由格式文本中，你可以使用HTML修饰符，例如，用于强调的...，用于单一空格“打字机”字体的<code>...</code>，用于粗黑的...，甚至可以使用包含图片的<img...>。但是你要远离用于标题的<h*n*>和用于规则的<hr>，因为它们会干扰文档的格式化。

注意：如果你的注释包含对其他文件，例如图片（例如，图表或者用户接口组件的图片）的链接，则将这些文件放在包含源文件的目录的子目录下面，命名为doc-files。javadoc工具会将doc-files目录和它们的内容从源代码目录复制到文档目录。你需要在链接中指定doc-files目录，例如<img src="doc-files/uml.png" alt="UML diagram">。

2.7.2 类注释

类注释必须直接放在类声明的前面。你可能想以@author和@version标签注释类的作者和版本。

这里是类注释的例子：

```
/**
 * 一个<code>Invoice</code> 对象代表发货单
 * 发货单中有订单每个部分的发货信息。
 * @author Fred Flintstone
 * @author Barney Rubble
```

```
 * @version 1.1
 */
public class Invoice {
    ...
}
```

注意：没有必要将*放在每一行的前面。但是，大多数IDE自动提供星号，并且当换行符变化时一些IDE甚至能重新排列。

2.7.3 方法注释

将方法的注释放在方法的紧前面。注释一些特性：

- 以@param变量描述注释每个参数
- 如果不是void，以@return描述注释返回值
- 以@throws异常类描述注释任何抛出的异常（参考第5章）

下面是方法注释的例子：

```
/**
 *提高员工的薪水。
 * @param byPercent 薪水提高的百分比（例如，10意味着10%）
 * @return 涨薪后的薪水
*/
public double raiseSalary(double byPercent) {
    double raise = salary * byPercent / 100;
    salary += raise;
    return raise;
}
```

2.7.4 变量注释

你只需要注释公有变量——通常是指静态常量。例如：

```
/**
 * 地球上每年的天数（闰年除外）
 */
public static final int DAYS_PER_YEAR = 365;
```

2.7.5 通用注释

在所有文档注释中，你可以使用@since标签来描述从哪个版本开始有这个功能：

```
@since version 1.7.1
```

@deprecated标签添加注释，表示类、方法或变量不应该再使用了。文本应该提供一个替代者。例如：

```
@deprecated使用<code>setVisible(true)</code>代替
```

注意：还有一个@Deprecated注解，当使用了不建议使用的内容时，编译器用来发出警告——参看第11章。注解没有提供建议替代者的机制，因此你应该为不建议使用的内容同时提供注解和Javadoc注释。

2.7.6 链接

你可以使用@see和@link标签给javadoc文档的相关部分或者外部文档添加超链接。

标签@see引用在"see also"部分添加超链接。它可以用在类和方法。引用可以是以下内容之一：

- *package.class#feature label*
- *label*
- "*text*"

第一种情况最有用。只要提供类、方法或者变量的名称，javadoc就在文档中插入超链接。例如：

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

会建立一个链接到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，链接会在当前包或者当前类中定位。

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推测出句号字符在分隔包、子包、类、内部类与方法和变量时的不同含义。但是javadoc工具就没有那么聪明了，因此必须对它提供帮助。

如果@see标签后面有一个<字符，则你正在指定一个超链接。你可以链接到你喜欢的任何URL，例如：@see Leap years。

在上述各种情况下，都可以指定一个可选的标识（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称或URL。

如果@see标签后面紧跟一个"字符，引号中的文本会在"see also"部分显示。例如：

```
@see "Core Java for the Impatient"
```

你可以给一个功能添加多个@see标签，但是必须将它们放在一起。

如果你喜欢，你可以在文档注释的任何地方添加链接到其他类或方法的超链接。在注释的任何地方插入这样的标签格式：

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

特性描述规则与@see标签规则相同。

2.7.7 包和概述注释

类、方法和变量注释都直接放在Java源文件中，以/**...*/分隔。但是，要想产生包注释，你需要在每个包目录添加单独的文件。

提供一个名称为package-info.java的Java文件。该文件必须包含以/**...*/分隔的初始javadoc注释，后面紧跟包声明。它不应该包含更多的代码或注释。

你也可以为所有源文件提供一个概述注释。放在一个名称为overview.html的文件中，该文件位于包含所有源文件的父目录中。所有在标签<body>...</body>之间的文本都会被提取。当用户从导航栏选择“Overview”时，会显示这些注释。

2.7.8 注释的提取

这里，假设你想将HTML文件存放到目录*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（如果你提供了overview.html文件，这也是它所在的目录）。

2. 运行命令：

```
javadoc -d docDirectory package1 package2 ...
```

如果省略了-d *docDirectory*选项，那HTML文件就会被提取到当前目录下。这样会带来混乱，因此不推荐这种做法。

可以使用许多命令行选项对javadoc程序进行微调整。例如，可以使用-author和-version选项在文档中包含@author和@version标签（默认情况下，这些标记会被省略）。另一个很有用的选项是-link，用来添加到标准类的超链接。例如，如果运行命令

```
javadoc -link http://docs.oracle.com/javase/8/docs/api *.java
```

那么，所有的标准类库类都会自动地链接到Oracle公司网站上的文档。

如果使用-linksource选项，则每个源文件被转换为HTML，并且每个类和方法名将转变为指向源代码的超链接。

练习

1. 改变日历打印程序，让每周从周日开始。同时让它在结尾新打印一行（并且只新打印一行）。
2. 思考一下Scanner类的nextInt方法。它是访问器方法还是修改器方法？为什么？那Random类的nextInt方法呢？
3. 你曾经写过返回值不为void的修改器方法吗？你曾有过返回值为void的访问器方法吗？可能的话给个例子。
4. 为什么不能实现一个可以交换两个int变量内容的Java方法，但是却可以编写一个交换两个IntHolder对象内容的方法？（在API文档中查询这个比较晦涩的IntHolder类。）你能交换两个Integer对象的内容吗？
5. 实现一个描述平面上点的不可修改的Point类。提供一个设置具体点的构造函数、一个设置为原点的无参数构造函数，以及方法getX、getY、translate和scale。translate方法根据给定量在*x*和*y*方向上移动该点。scale方法根据给定因子在坐标系按比例变化。实现这些方法以便它们将新的Point对象作为结果返回。例如：

```
Point p = new Point(3, 4).translate(1, 3).scale(0.5);
```

应该设置p为坐标系上的点(2,3.5)。

6. 重复前面的练习5，但是使得translate和scale方法变成修改器方法.
7. 给前面两个版本的Point类添加javadoc注释。
8. 在前面的练习中，提供的Point类的这些构造函数和getter方法有些重复。大多数IDE为编写样板代码都提供了快捷方式。你的IDE都提供了哪些?
9. 实现一个Car类，模拟汽车沿着*x*轴移动，随着移动消耗燃油。提供一个按照给定英里数驱动汽车的方法、给汽车燃料箱添加给定加仑的方法，以及获取原点到油位的当前距离的方法。在构造函数中指定燃料效率（单位为每加仑多少英里）。Car类应该是不可修改类吗？为什么或者为什么不应该?
10. 在RandomNumbers类中，提供两个静态方法randomElement，从数组或者整数数组列表中获得随机元素。（如果数组或者数组列表为空，则返回0。）为什么不能使这两个方法成为int[]或ArrayList<Integer>的实例方法?
11. 对System和LocalDate类使用静态导入，重写Cal类。
12. 创建一个HelloWorld.java文件，它在包ch01.sec01中声明了HelloWorld类。把它放在某个目录，但不在ch01/sec01子目录。从存放文件的那个目录运行javac HelloWorld.java。你会得到类文件吗？类文件存放在哪里？接着运行java HelloWorld。会发生什么？为什么？（提示：运行javap HelloWorld 并研究告警信息。）最后，为什么javac -d . HelloWorld.java更好?
13. 从http://opencsv.sourceforge.net下载OpenCSV的JAR文件。写个有main方法的类，读取你选择的CSV文件并打印部分内容。OpenCSV网站上有示例代码。你还未学习处理异常。先用下面的内容作为main方法头：

```
public static void main(String[] args) throws Exception
```

这个练习的目的不是让你用CSV文件做些有用的事情，而是练习如何使用交付为JAR文件格式的类库。

14. 编译Network类。注意内部类文件被命名为Network$Member.class。使用javap程序检查生成的代码。命令

```
javap -private Clasname
```

显示方法和实例变量。你在哪里看到外部类的引用？（在Linux/Mac OS X系统上，运行javap时，在$符号前你需要输入“\”。）

15. 完整实现2.6.1节中的Invoice类。提供一个打印invoice的方法，以及构造并打印示例invoice的演示程序。
16. 实现一个Queue类——一个无边界的字符串队列，分别提供在末尾添加字符串的add方法和从队列头部进行删除的remove方法。将元素存储成链表节点。创建一个嵌套类Node。Node类应该是静态类或不应该是静态类吗？
17. 提供一个iterator——依次产生队列元素的对象——就是上一道练习题中的队列。将Iterator作为拥有next和hasNext方法的嵌套类。给Queue类提供iterator()方法，该方法产生Queue.Iterator。Iterator应该是静态类或不应该是静态类吗？

第3章　接口和lambda表达式

本章的主题

- 3.1　接口——第96页
- 3.2　静态方法和默认方法——第101页
- 3.3　接口示例——第105页
- 3.4　lambda表达式——第110页
- 3.5　方法引用和构造函数引用——第112页
- 3.6　使用lambda表达式——第115页
- 3.7　lambda表达式和变量作用域——第119页
- 3.8　高阶函数——第123页
- 3.9　局部内部类——第125页
- 练习——第127页