

第3章 Java 的基本程序设计结构

- ▲ 一个简单的 Java 应用程序
- ▲ 字符串
- ▲ 注释
- ▲ 输入输出
- ▲ 数据类型
- ▲ 控制流
- ▲ 变量
- ▲ 大数值
- ▲ 运算符
- ▲ 数组

现在，假定已经成功地安装了 JDK，并且能够运行第 2 章中给出的示例程序。我们从现在开始将介绍 Java 应用程序设计。本章主要介绍程序设计的基本概念（如数据类型、分支以及循环）在 Java 中的实现方式。

非常遗憾，需要告诫大家，使用 Java 编写 GUI 应用程序并不是一件很容易的事情，编程者需要掌握很多相关的知识才能够创建窗口、添加文本框以及能响应的按钮等。介绍基于 GUI 的 Java 应用程序设计技术与本章将要介绍的程序设计基本概念相差甚远，因此本章给出的所有示例都是为了说明一些相关概念而设计的“玩具式”程序，它们仅仅使用终端窗口提供输入输出。

最后需要说明，对于一个有 C++ 编程经验的程序员来说，本章的内容只需要浏览一下，应该重点阅读散布在正文中的 C/C++ 注释。对于具有使用 Visual Basic 等其他编程背景的程序员来说，可能会发现其中的绝大多数概念都很熟悉，但是在语法上有比较大的差异，因此，需要非常仔细地阅读本章的内容。

3.1 一个简单的 Java 应用程序

下面看一个最简单的 Java 应用程序，它只发送一条消息到控制台窗口中：

```
public class FirstSample
{
    public static void main(String[] args)
    {
        System.out.println("We will not use 'Hello, World!'");
    }
}
```

这个程序虽然很简单，但所有的 Java 应用程序都具有这种结构，还是值得花一些时间来研究。首先，Java 区分大小写。如果出现了大小写拼写错误（例如，将 main 拼写成 Main），程序将无法运行。

下面逐行地查看一下这段源代码。关键字 public 称为访问修饰符（access modifier），这些修饰符用于控制程序的其他部分对这段代码的访问级别。在第 5 章中将会更加详细地介绍

访问修饰符的具体内容。关键字 `class` 表明 Java 程序中的全部内容都包含在类中。这里，只需要将类作为一个加载程序逻辑的容器，程序逻辑定义了应用程序的行为。在第 4 章中将会用大量的篇幅介绍 Java 类。正如第 1 章所述，类是构建所有 Java 应用程序和 applet 的构建块。Java 应用程序中的全部内容都必须放置在类中。

关键字 `class` 后面紧跟类名。Java 中定义类名的规则很宽松。名字必须以字母开头，后面可以跟字母和数字的任意组合。长度基本上没有限制。但是不能使用 Java 保留字（例如，`public` 或 `class`）作为类名（保留字列表请参看附录 A）。

标准的命名规范为（类名 `FirstSample` 就遵循了这个规范）：类名是以大写字母开头的名词。如果名字由多个单词组成，每个单词的第一个字母都应该大写（这种在一个单词中间使用大写字母的方式称为骆驼命名法。以其自身为例，应该写成 `CamelCase`）。

源代码的文件名必须与公共类的名字相同，并用 `.java` 作为扩展名。因此，存储这段源代码的文件名必须为 `FirstSample.java`（再次提醒大家注意，大小写是非常重要的，千万不能写成 `firstsample.java`）。

如果已经正确地命名了这个文件，并且源代码中没有任何录入错误，在编译这段源代码之后就会得到一个包含这个类字节码的文件。Java 编译器将字节码文件自动地命名为 `FirstSample.class`，并与源文件存储在同一个目录下。最后，使用下面这行命令运行这个程序：

```
java FirstSample
```

（请记住，不要添加 `.class` 扩展名。）程序执行之后，控制台上将会显示“`We will not use 'Hello,World' !`”。

当使用

```
java ClassName
```

运行已编译的程序时，Java 虚拟机将从指定类中的 `main` 方法开始执行（这里的“方法”就是 Java 中所说的“函数”），因此为了代码能够执行，在类的源文件中必须包含一个 `main` 方法。当然，也可以将用户自定义的方法添加到类中，并且在 `main` 方法中调用它们（第 4 章将讲述如何自定义方法）。

 **注释：**根据 Java 语言规范，`main` 方法必须声明为 `public`（Java 语言规范是描述 Java 语言的官方文档。可以从网站 <http://docs.oracle.com/javase/specs> 上阅读或下载）。

不过，当 `main` 方法不是 `public` 时，有些版本的 Java 解释器也可以执行 Java 应用程序。有个程序员报告了这个 bug。如果感兴趣的话，可以在网站 <http://bugs.java.com/bugdatabase/index.jsp> 上输入 bug 号码 4252539 查看。这个 bug 被标明“关闭，不予修复。”Sun 公司的工程师解释说：Java 虚拟机规范（在 <http://docs.oracle.com/javase/specs/jvms/se8/html>）并没有要求 `main` 方法一定是 `public`，并且“修复这个 bug 有可能带来其他的隐患”。好在，这个问题最终得到了解决。在 Java SE 1.4 及以后的版本中强制 `main` 方法是 `public` 的。

从上面这段话可以发现一个问题的两个方面。一方面让质量保证工程师判断在 bug

报告中是否存在问题是件很头痛的事情，这是因为其工作量很大，并且工程师对 Java 的所有细节也未必了解得很清楚。另一方面，Sun 公司在 Java 开源很久以前就把 bug 报告及其解决方案放到网站上让所有人监督检查，这是一种非常了不起的举动。某些情况下，Sun 甚至允许程序员为他们最厌恶的 bug 投票，并用投票结果来决定发布的下一个 JDK 版本将修复哪些 bug。

需要注意源代码中的括号 {}。在 Java 中，像在 C/C++ 中一样，用大括号划分程序的各个部分（通常称为块）。Java 中任何方法的代码都用“{”开始，用“}”结束。

大括号的使用风格曾经引发过许多无意义的争论。我们的习惯是把匹配的大括号上下对齐。不过，由于空白符会被 Java 编译器忽略，所以可以选用自己喜欢的大括号风格。在下面讲述各种循环语句时，我们还会详细地介绍大括号的使用。

我们暂且不去理睬关键字 static void，而仅把它们当作编译 Java 应用程序必要的部分就行了。在学习完第 4 章后，这些内容的作用就会揭晓。现在需要记住：每个 Java 应用程序都必须有一个 main 方法，其声明格式如下所示：

```
public class ClassName
{
    public static void main(String[] args)
    {
        program statements
    }
}
```

C++ 注释：作为一名 C++ 程序员，一定知道类的概念。Java 的类与 C++ 的类很相似，但还是有些差异会使人感到困惑。例如，Java 中的所有函数都属于某个类的方法（标准术语将其称为方法，而不是成员函数）。因此，Java 中的 main 方法必须有一个外壳类。读者有可能对 C++ 中的静态成员函数（static member functions）十分熟悉。这些成员函数定义在类的内部，并且不对对象进行操作。Java 中的 main 方法必须是静态的。最后，与 C/C++ 一样，关键字 void 表示这个方法没有返回值，所不同的是 main 方法没有为操作系统返回“退出代码”。如果 main 方法正常退出，那么 Java 应用程序的退出代码为 0，表示成功地运行了程序。如果希望在终止程序时返回其他的代码，那就需要调用 System.exit 方法。

接下来，研究一下这段代码：

```
{
    System.out.println("We will not use 'Hello, World!'");
}
```

一对大括号表示方法体的开始与结束，在这个方法中只包含一条语句。与大多数程序设计语言一样，可以将 Java 语句看成是这种语言的句子。在 Java 中，每个句子必须用分号结束。特别需要说明，回车不是语句的结束标志，因此，如果需要可以将一条语句写在多行上。

在上面这个 main 方法体中只包含了一条语句，其功能是：将一个文本行输出到控制台上。

在这里，使用了 System.out 对象并调用了它的 println 方法。注意，点号（·）用于调用方法。Java 使用的通用语法是

object.method(parameters)

这等价于函数调用。

在这个示例中，调用了 println 方法并传递给它一个字符串参数。这个方法将传递给它的字符串参数显示在控制台上。然后，终止这个输出行，使得每次调用 println 都会在新的一行上显示输出。需要注意一点，Java 与 C/C++ 一样，都采用双引号分隔字符串。（本章稍后将会详细地讲解有关字符串的知识）。

与其他程序设计语言中的函数一样，在 Java 的方法中，可以没有参数，也可以有一个或多个参数（有的程序员把参数叫做实参）。对于一个方法，即使没有参数也需要使用空括号。例如，不带参数的 println 方法只打印一个空行。使用下面的语句来调用：

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

注释：System.out 还有一个 print 方法，它在输出之后不换行。例如，System.out.print (“Hello”) 打印 “Hello” 之后不换行，后面的输出紧跟在字母 “o” 之后。

3.2 注释

与大多数程序设计语言一样，Java 中的注释也不会出现在可执行程序中。因此，可以在源程序中根据需要添加任意多的注释，而不必担心可执行代码会膨胀。在 Java 中，有 3 种标记注释的方式。最常用的方式是使用 //，其注释内容从 // 开始到本行结尾。

```
System.out.println("We will not use 'Hello, World!'"); // is this too cute?
```

当需要长篇的注释时，既可以在每行的注释前面标记 //，也可以使用 /* 和 */ 将一段比较长的注释括起来。

最后，第 3 种注释可以用来自动地生成文档。这种注释以 /** 开始，以 */ 结束。请参见程序清单 3-1。有关这种注释的详细内容和自动生成文档的具体方法请参见第 4 章。

程序清单 3-1 FirstSample/FirstSample.java

```

1 /**
2  * This is the first sample program in Core Java Chapter 3
3  * @version 1.01 1997-03-22
4  * @author Gary Cornell
5 */
6 public class FirstSample
7 {
8     public static void main(String[] args)
9     {
10         System.out.println("We will not use 'Hello, World!'");
11     }
12 }
```

! 警告：在 Java 中，`/* */`注释不能嵌套。也就是说，不能简单地把代码用`/*`和`*/`括起来作为注释，因为这段代码本身可能也包含一个`*`。

3.3 数据类型

Java 是一种强类型语言。这就意味着必须为每一个变量声明一种类型。在 Java 中，一共有 8 种基本类型（primitive type），其中有 4 种整型、2 种浮点类型、1 种用于表示 Unicode 编码的字符单元的字符类型 `char`（请参见论述 `char` 类型的章节）和 1 种用于表示真值的 `boolean` 类型。

■ 注释：Java 有一个能够表示任意精度的算术包，通常称为“大数值”（big number）。虽然被称为大数值，但它并不是一种新的 Java 类型，而是一个 Java 对象。本章稍后将会详细地介绍它的用法。

3.3.1 整型

整型用于表示没有小数部分的数值，它允许是负数。Java 提供了 4 种整型，具体内容如表 3-1 所示。

表 3-1 Java 整型

类型	存储需求	取值范围
<code>int</code>	4 字节	-2 147 483 648 ~ 2 147 483 647 (正好超过 20 亿)
<code>short</code>	2 字节	-32 768 ~ 32 767
<code>long</code>	8 字节	-9 223 372 036 854 775 808 ~ 9 223 372 036 854 775 807
<code>byte</code>	1 字节	-128 ~ 127

在通常情况下，`int` 类型最常用。但如果表示星球上的居住人数，就需要使用 `long` 类型了。`byte` 和 `short` 类型主要用于特定的应用场合，例如，底层的文件处理或者需要控制占用存储空间量的大数组。

在 Java 中，整型的范围与运行 Java 代码的机器无关。这就解决了软件从一个平台移植到另一个平台，或者在同一个平台中的不同操作系统之间进行移植给程序员带来的诸多问题。与此相反，C 和 C++ 程序需要针对不同的处理器选择最为高效的整型，这样就有可能造成一个在 32 位处理器上运行很好的 C 程序在 16 位系统上运行却发生整数溢出。由于 Java 程序必须保证在所有机器上都能够得到相同的运行结果，所以各种数据类型的取值范围必须固定。

长整型数值有一个后缀 `L` 或 `l`（如 `4000000000L`）。十六进制数值有一个前缀 `0x` 或 `0X`（如 `0xCAFE`）。八进制有一个前缀 `0`，例如，`010` 对应八进制中的 `8`。很显然，八进制表示法比较容易混淆，所以建议最好不要使用八进制常数。

从 Java 7 开始，加上前缀 `0b` 或 `0B` 就可以写二进制数。例如，`0b1001` 就是 `9`。另外，同样是从 Java 7 开始，还可以为数字字面量加下划线，如用 `1_000_000`（或 `0b1111_0100_0010_0100_0000`）表示一百万。这些下划线只是为了让人更易读。Java 编译器会去除这些下划线。

C++ 注释：在 C 和 C++ 中，int 和 long 等类型的大小与目标平台相关。在 8086 这样的 16 位处理器上整型数值占 2 字节；不过，在 32 位处理器（比如 Pentium 或 SPARC）上，整型数值则为 4 字节。类似地，在 32 位处理器上 long 值为 4 字节，在 64 位处理器上则为 8 字节。由于存在这些差别，这对编写跨平台程序带来了很大难度。在 Java 中，所有的数值类型所占据的字节数量与平台无关。

注意，Java 没有任何无符号（unsigned）形式的 int、long、short 或 byte 类型。

3.3.2 浮点类型

浮点类型用于表示有小数部分的数值。在 Java 中有两种浮点类型，具体内容如表 3-2 所示。

表 3-2 浮点类型

类型	存储需求	取值范围
float	4 字节	大约 $\pm 3.402\ 823\ 47E+38F$ (有效位数为 6 ~ 7 位)
double	8 字节	大约 $\pm 1.797\ 693\ 134\ 862\ 315\ 70E+308$ (有效位数为 15 位)

double 表示这种类型的数值精度是 float 类型的两倍（有人称之为双精度数值）。绝大部分应用程序都采用 double 类型。在很多情况下，float 类型的精度很难满足需求。实际上，只有很少的情况适合使用 float 类型，例如，需要单精度数据的库，或者需要存储大量数据。

float 类型的数值有一个后缀 F 或 f（例如，3.14F）。没有后缀 F 的浮点数值（如 3.14）默认为 double 类型。当然，也可以在浮点数值后面添加后缀 D 或 d（例如，3.14D）。

注释：可以使用十六进制表示浮点数值。例如， $0.125=2^{-3}$ 可以表示成 0x1.0p-3。在十六进制表示法中，使用 p 表示指数，而不是 e。注意，尾数采用十六进制，指数采用十进制。指数的基数是 2，而不是 10。

所有的浮点数值计算都遵循 IEEE 754 规范。具体来说，下面是用于表示溢出和出错情况的三个特殊的浮点数值：

- 正无穷大
- 负无穷大
- NaN（不是一个数字）

例如，一个正整数除以 0 的结果为正无穷大。计算 0/0 或者负数的平方根结果为 NaN。

注释：常量 Double.POSITIVE_INFINITY、Double.NEGATIVE_INFINITY 和 Double.NaN（以及相应的 Float 类型的常量）分别表示这三个特殊的值，但在实际应用中很少遇到。特别要说明的是，不能这样检测一个特定值是否等于 Double.NaN：

```
if (x == Double.NaN) // is never true
```

所有“非数值”的值都认为是不相同的。然而，可以使用 Double.isNaN 方法：

```
if (Double.isNaN(x)) // check whether x is "not a number"
```

！ 警告：浮点数值不适用于无法接受舍入误差的金融计算中。例如，命令 System.out.println(2.0-1.1) 将打印出 0.8999999999999999，而不是人们想象的 0.9。这种舍入误差的主要原因是浮点数值采用二进制系统表示，而在二进制系统中无法精确地表示分数 1/10。这就好像十进制无法精确地表示分数 1/3 一样。如果在数值计算中不允许有任何舍入误差，就应该使用 BigDecimal 类，本章稍后将介绍这个类。

3.3.3 char 类型

char 类型原本用于表示单个字符。不过，现在情况已经有所变化。如今，有些 Unicode 字符可以用一个 char 值描述，另外一些 Unicode 字符则需要两个 char 值。有关的详细信息请阅读下一节。

char 类型的字面量值要用单引号括起来。例如：'A' 是编码值为 65 所对应的字符常量。它与 "A" 不同，"A" 是包含一个字符 A 的字符串。char 类型的值可以表示为十六进制值，其范围从 \u0000 到 \Uffff。例如：\u2122 表示注册符号 (™)，\u03c0 表示希腊字母 π。

除了转义序列 \u 之外，还有一些用于表示特殊字符的转义序列，请参看表 3-3。所有这些转义序列都可以出现在加引号的字符字面量或字符串中。例如，"\u2122" 或 "Hello\n"。转义序列 \u 还可以出现在加引号的字符常量或字符串之外（而其他所有转义序列不可以）。例如：

```
public static void main(String\u005B\u005D args)
```

就完全符合语法规则，\u005B 和 \u005D 是 [和] 的编码。

表 3-3 特殊字符的转义序列

转义序列	名称	Unicode 值	转义序列	名称	Unicode 值
\b	退格	\u0008	\"	双引号	\u0022
\t	制表	\u0009	\'	单引号	\u0027
\n	换行	\u000a	\\\	反斜杠	\u005c
\r	回车	\u000d			

！ 警告：Unicode 转义序列会在解析代码之前得到处理。例如，"\u0022+\u0022" 并不是一个由引号 (U+0022) 包围加号构成的字符串。实际上，\u0022 会在解析之前转换为 "，这会得到 ""+""，也就是一个空串。

更隐秘地，一定要当心注释中的 \u。注释

```
// \u00A0 is a newline
```

会产生一个语法错误，因为读程序时 \u00A0 会替换为一个换行符。类似地，下面这个注释

```
// Look inside c:\users
```

也会产生一个语法错误，因为 \u 后面并未跟着 4 个十六进制数。

3.3.4 Unicode 和 char 类型

要想弄清 char 类型，就必须了解 Unicode 编码机制。Unicode 打破了传统字符编码机制的限制。在 Unicode 出现之前，已经有许多种不同的标准：美国的 ASCII、西欧语言中的 ISO 8859-1、俄罗斯的 KOI-8、中国的 GB 18030 和 BIG-5 等。这样就产生了下面两个问题：一个是对任意给定的代码值，在不同的编码方案下有可能对应不同的字母；二是采用大字符集的语言其编码长度有可能不同。例如，有些常用的字符采用单字节编码，而另一些字符则需要两个或更多个字节。

设计 Unicode 编码的目的就是要解决这些问题。在 20 世纪 80 年代开始启动设计工作时，人们认为两个字节的代码宽度足以对世界上各种语言的所有字符进行编码，并有足够的空间留给未来的扩展。在 1991 年发布了 Unicode 1.0，当时仅占用 65 536 个代码值中不到一半的部分。在设计 Java 时决定采用 16 位的 Unicode 字符集，这样会比使用 8 位字符集的程序设计语言有很大的改进。

十分遗憾，经过一段时间，不可避免的事情发生了。Unicode 字符超过了 65 536 个，其主要原因是增加了大量的汉语、日语和韩语中的表意文字。现在，16 位的 char 类型已经不能满足描述所有 Unicode 字符的需要了。

下面利用一些专用术语解释一下 Java 语言解决这个问题的基本方法。从 Java SE 5.0 开始。码点（code point）是指与一个编码表中的某个字符对应的代码值。在 Unicode 标准中，码点采用十六进制书写，并加上前缀 U+，例如 U+0041 就是拉丁字母 A 的码点。Unicode 的码点可以分成 17 个代码级别（code plane）。第一个代码级别称为基本的多语言级别（basic multilingual plane），码点从 U+0000 到 U+FFFF，其中包括经典的 Unicode 代码；其余的 16 个级别码点从 U+10000 到 U+10FFFF，其中包括一些辅助字符（supplementary character）。

UTF-16 编码采用不同长度的编码表示所有 Unicode 码点。在基本的多语言级别中，每个字符用 16 位表示，通常被称为代码单元（code unit）；而辅助字符采用一对连续的代码单元进行编码。这样构成的编码值落入基本的多语言级别中空闲的 2048 字节内，通常被称为替代区域（surrogate area）[U+D800 ~ U+DBFF 用于第一个代码单元，U+DC00 ~ U+DFFF 用于第二个代码单元]。这样设计十分巧妙，我们可以从中迅速地知道一个代码单元是一个字符的编码，还是一个辅助字符的第一或第二部分。例如，∅ 是八元数集 (<http://math.ucr.edu/home/baez/octonions>) 的一个数学符号，码点为 U+1D546，编码为两个代码单元 U+D835 和 U+DD46。（关于编码算法的具体描述见 <http://en.wikipedia.org/wiki/UTF-16>）。

在 Java 中，char 类型描述了 UTF-16 编码中的一个代码单元。

我们强烈建议不要在程序中使用 char 类型，除非确实需要处理 UTF-16 代码单元。最好将字符串作为抽象数据类型处理（有关这方面的内容将在 3.6 节讨论）。

3.3.5 boolean 类型

boolean（布尔）类型有两个值：false 和 true，用来判定逻辑条件。整型值和布尔值之间不能进行相互转换。

C++ C++注释：在C++中，数值甚至指针可以代替boolean值。值0相当于布尔值false，非0值相当于布尔值true。在Java中则不是这样。因此，Java程序员不会遇到下述麻烦：

```
if (x = 0) // oops... meant x == 0
```

在C++中这个测试可以编译运行，其结果总是false。而在Java中，这个测试将不能通过编译，其原因是整数表达式x=0不能转换为布尔值。

3.4 变量

在Java中，每个变量都有一个类型(type)。在声明变量时，变量的类型位于变量名之前。这里列举一些声明变量的示例：

```
double salary;
int vacationDays;
long earthPopulation;
boolean done;
```

可以看到，每个声明以分号结束。由于声明是一条完整的Java语句，所以必须以分号结束。

变量名必须是一个以字母开头并由字母或数字构成的序列。需要注意，与大多数程序设计语言相比，Java中“字母”和“数字”的范围更大。字母包括'A' ~ 'Z'、'a' ~ 'z'、'_'、'\$'或在某种语言中表示字母的任何Unicode字符。例如，德国的用户可以在变量名中使用字母‘ä’；希腊人可以用π。同样，数字包括'0' ~ '9'和在某种语言中表示数字的任何Unicode字符。但'+'和'©'这样的符号不能出现在变量名中，空格也不行。变量名中所有的字符都是有意义的，并且大小写敏感。变量名的长度基本上没有限制。

提示：如果想要知道哪些Unicode字符属于Java中的“字母”，可以使用Character类的isJavaIdentifierStart和isJavaIdentifierPart方法来检查。

提示：尽管\$是一个合法的Java字符，但不要在你自己的代码中使用这个字符。它只用在Java编译器或其他工具生成的名字中。

另外，不能使用Java保留字作为变量名（请参看附录A中的保留字列表）。

可以在一行中声明多个变量：

```
int i, j; // both are integers
```

不过，不提倡使用这种风格。逐一声明每一个变量可以提高程序的可读性。

注释：如前所述，变量名对大小写敏感，例如，hireday和hireDay是两个不同的变量名。在对两个不同的变量进行命名时，最好不要只存在大小写上的差异。不过，在有些时候，确实很难给变量取一个好的名字。于是，许多程序员将变量名命名为类型名，例如：

```
Box box; // "Box" is the type and "box" is the variable name
```

还有一些程序员更加喜欢在变量名前加上前缀“a”：

```
Box aBox;
```

3.4.1 变量初始化

声明一个变量之后，必须用赋值语句对变量进行显式初始化，千万不要使用未初始化的变量。例如，Java 编译器认为下面的语句序列是错误的：

```
int vacationDays;
System.out.println(vacationDays); // ERROR--variable not initialized
```

要想对一个已经声明过的变量进行赋值，就需要将变量名放在等号（=）左侧，相应取值的 Java 表达式放在等号的右侧。

```
int vacationDays;
vacationDays = 12;
```

也可以将变量的声明和初始化放在同一行中。例如：

```
int vacationDays = 12;
```

最后，在 Java 中可以将声明放在代码中的任何地方。例如，下列代码的书写形式在 Java 中是完全合法的：

```
double salary = 65000.0;
System.out.println(salary);
int vacationDays = 12; // OK to declare a variable here
```

在 Java 中，变量的声明尽可能地靠近变量第一次使用的地方，这是一种良好的程序编写风格。

 **C++ 注释：**C 和 C++ 区分变量的声明与定义。例如：

```
int i = 10;
```

是一个定义，而

```
extern int i;
```

是一个声明。在 Java 中，不区分变量的声明与定义。

3.4.2 常量

在 Java 中，利用关键字 final 指示常量。例如：

```
public class Constants
{
    public static void main(String[] args)
    {
        final double CM_PER_INCH = 2.54;
        double paperWidth = 8.5;
        double paperHeight = 11;
        System.out.println("Paper size in centimeters: "
            + paperWidth * CM_PER_INCH + " by " + paperHeight * CM_PER_INCH);
    }
}
```

关键字 final 表示这个变量只能被赋值一次。一旦被赋值之后，就不能够再更改了。习惯上，常量名使用全大写。

在 Java 中，经常希望某个常量可以在一个类中的多个方法中使用，通常将这些常量称为类常量。可以使用关键字 `static final` 设置一个类常量。下面是使用类常量的示例：

```
public class Constants2
{
    public static final double CM_PER_INCH = 2.54;

    public static void main(String[] args)
    {
        double paperWidth = 8.5;
        double paperHeight = 11;
        System.out.println("Paper size in centimeters: "
            + paperWidth * CM_PER_INCH + " by " + paperHeight * CM_PER_INCH);
    }
}
```

需要注意，类常量的定义位于 `main` 方法的外部。因此，在同一个类的其他方法中也可以使用这个常量。而且，如果一个常量被声明为 `public`，那么其他类的方法也可以使用这个常量。在这个示例中，`Constants2.CM_PER_INCH` 就是这样一个常量。

C++ 注释：`const` 是 Java 保留的关键字，但目前并没有使用。在 Java 中，必须使用 `final` 定义常量。

3.5 运算符

在 Java 中，使用算术运算符 `+`、`-`、`*`、`/` 表示加、减、乘、除运算。当参与 `/` 运算的两个操作数都是整数时，表示整数除法；否则，表示浮点除法。整数的求余操作（有时称为取模）用 `%` 表示。例如，`15/2` 等于 `7`，`15%2` 等于 `1`，`15.0/2` 等于 `7.5`。

需要注意，整数被 `0` 除将会产生一个异常，而浮点数被 `0` 除将会得到无穷大或 `NaN` 结果。

注释：可移植性是 Java 语言的设计目标之一。无论在哪个虚拟机上运行，同一运算应该得到同样的结果。对于浮点数的算术运算，实现这样的可移植性是相当困难的。`double` 类型使用 64 位存储一个数值，而有些处理器使用 80 位浮点寄存器。这些寄存器增加了中间过程的计算精度。例如，以下运算：

```
double w = x * y / z;
```

很多 Intel 处理器计算 `x * y`，并且将结果存储在 80 位的寄存器中，再除以 `z` 并将结果截断为 64 位。这样可以得到一个更加精确的计算结果，并且还能够避免产生指数溢出。但是，这个结果可能与始终在 64 位机器上计算的结果不一样。因此，Java 虚拟机的最初规范规定所有的中间计算都必须进行截断。这种行为遭到了数值计算团体的反对。截断计算不仅可能导致溢出，而且由于截断操作需要消耗时间，所以在计算速度上实际上要比精确计算慢。为此，Java 程序设计语言承认了最优性能与理想结果之间存在的冲突，并给予了改进。在默认情况下，虚拟机设计者允许对中间计算结果采用扩展的精度。但是，对于使用 `strictfp` 关键字标记的方法必须使用严格的浮点计算来生成可再生的结

果。例如，可以把 main 方法标记为

```
public static strictfp void main(String[] args)
```

于是，在 main 方法中的所有指令都将使用严格的浮点计算。如果将一个类标记为 strictfp，这个类中的所有方法都要使用严格的浮点计算。

实际的计算方式将取决于 Intel 处理器的行为。在默认情况下，中间结果允许使用扩展的指数，但不允许使用扩展的尾数（Intel 芯片在截断尾数时并不损失性能）。因此，这两种方式的区别仅仅在于采用默认的方式不会产生溢出，而采用严格的计算有可能产生溢出。

如果没有仔细阅读这个注释，也没有什么关系。对大多数程序来说，浮点溢出不属于大问题。在本书中，将不使用 strictfp 关键字。

3.5.1 数学函数与常量

在 Math 类中，包含了各种各样的数学函数。在编写不同类别的程序时，可能需要的函数也不同。

要想计算一个数值的平方根，可以使用 sqrt 方法：

```
double x = 4;
double y = Math.sqrt(x);
System.out.println(y); // prints 2.0
```

 **注释：** println 方法和 sqrt 方法存在微小的差异。println 方法处理 System.out 对象。但是，Math 类中的 sqrt 方法处理的不是对象，这样的方法被称为静态方法。有关静态方法的详细内容请参看第 4 章。

在 Java 中，没有幂运算，因此需要借助于 Math 类的 pow 方法。语句：

```
double y = Math.pow(x, a);
```

将 y 的值设置为 x 的 a 次幂 (x^a)。pow 方法有两个 double 类型的参数，其返回结果也为 double 类型。

floorMod 方法的目的是解决一个长期存在的有关整数余数的问题。考虑表达式 $n \% 2$ 。所有人都知道，如果 n 是偶数，这个表达式为 0；如果 n 是奇数，表达式则为 1。当然，除非 n 是负数。如果 n 为负，这个表达式则为 -1。为什么呢？设计最早的计算机时，必须有人制定规则，明确整数除法和求余对负数操作数该如何处理。数学家们几百年来都知道这样一个最优（或“欧几里德”）规则：余数总是要 ≥ 0 。不过，最早制定规则的人并没有翻开数学书好好研究，而是提出了一些看似合理但实际上很不方便的规则。

下面考虑这样一个问题：计算一个时钟时针的位置。这里要做一个时间调整，而且要归一化为一个 0 ~ 11 之间的数。这很简单： $(position + adjustment) \% 12$ 。不过，如果这个调整为负会怎么样呢？你可能会得到一个负数。所以要引入一个分支，或者使用 $((position + adjustment) \% 12 + 12) \% 12$ 。不管怎样，总之都很麻烦。

floorMod 方法就让这个问题变得容易了： $\text{floorMod}(position + adjustment, 12)$ 总会得到一

个 0 ~ 11 之间的数。(遗憾的是，对于负除数，`floorMod` 会得到负数结果，不过这种情况在实际中很少出现。)

`Math` 类提供了一些常用的三角函数：

```
Math.sin  
Math.cos  
Math.tan  
Math.atan  
Math.atan2
```

还有指数函数以及它的反函数——自然对数以及以 10 为底的对数：

```
Math.exp  
Math.log  
Math.log10
```

最后，Java 还提供了两个用于表示 π 和 e 常量的近似值：

```
Math.PI  
Math.E
```

 **提示：**不必在数学方法名和常量名前添加前缀“`Math`”，只要在源文件的顶部加上下面这行代码就可以了。

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

例如：

```
System.out.println("The square root of \u03c0 is " + sqrt(PI));
```

在第 4 章中将讨论静态导入。

 **注释：**在 `Math` 类中，为了达到最快的性能，所有的方法都使用计算机浮点单元中的例程。如果得到一个完全可预测的结果比运行速度更重要的话，那么就应该使用 `StrictMath` 类。它使用“自由发布的 Math 库”(fdlibm) 实现算法，以确保在所有平台上得到相同的结果。有关这些算法的源代码请参看 www.netlib.org/fdlibm (当 fdlibm 为一个函数提供了多个定义时，`StrictMath` 类就会遵循 IEEE 754 版本，它的名字将以“e”开头)。

3.5.2 数值类型之间的转换

经常需要将一种数值类型转换为另一种数值类型。图 3-1 给出了数值类型之间的合法转换。

在图 3-1 中有 6 个实心箭头，表示无信息丢失的转换；有 3 个虚箭头，表示可能有精度损失的转换。例如，123 456 789 是一个大整数，它所包含的位数比 `float` 类型所能够表达的位数多。当将这个整型数值转换为 `float` 类型时，将会得到同样大小的结果，但却失去了一定的精度。

```
int n = 123456789;  
float f = n; // f is 1.2345679E8
```

当使用上面两个数值进行二元操作时（例如 `n + f`，`n` 是整数，`f` 是浮点数），先要将两个

操作数转换为同一种类型，然后再进行计算。

- 如果两个操作数中有一个是 double 类型，另一个操作数就会转换为 double 类型。
- 否则，如果其中一个操作数是 float 类型，另一个操作数将会转换为 float 类型。
- 否则，如果其中一个操作数是 long 类型，另一个操作数将会转换为 long 类型。
- 否则，两个操作数都将被转换为 int 类型。

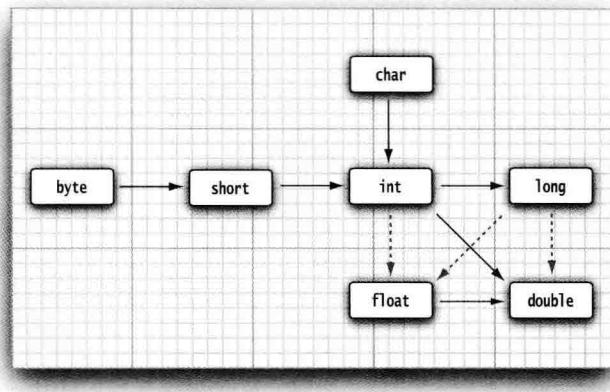


图 3-1 数值类型之间的合法转换

3.5.3 强制类型转换

在上一小节中看到，在必要的时候，int 类型的值将会自动地转换为 double 类型。但另一方面，有时也需要将 double 转换成 int。在 Java 中，允许进行这种数值之间的类型转换。当然，有可能会丢失一些信息。在这种情况下，需要通过强制类型转换 (cast) 实现这个操作。强制类型转换的语法格式是在圆括号中给出想要转换的目标类型，后面紧跟待转换的变量名。例如：

```
double x = 9.997;
int nx = (int) x;
```

这样，变量 nx 的值为 9。强制类型转换通过截断小数部分将浮点值转换为整型。

如果想对浮点数进行舍入运算，以便得到最接近的整数（在很多情况下，这种操作更有用），那就需要使用 Math.round 方法：

```
double x = 9.997;
int nx = (int) Math.round(x);
```

现在，变量 nx 的值为 10。当调用 round 的时候，仍然需要使用强制类型转换 (int)。其原因是 round 方法返回的结果为 long 类型，由于存在信息丢失的可能性，所以只有使用显式的强制类型转换才能够将 long 类型转换成 int 类型。

◆ 警告：如果试图将一个数值从一种类型强制转换为另一种类型，而又超出了目标类型的表示范围，结果就会截断成一个完全不同的值。例如，(byte) 300 的实际值为 44。

C++ 注释：不要在 boolean 类型与任何数值类型之间进行强制类型转换，这样可以防止发生错误。只有极少数的情况才需要将布尔类型转换为数值类型，这时可以使用条件表达式 `b ? 1:0`。

3.5.4 结合赋值和运算符

可以在赋值中使用二元运算符，这是一种很方便的简写形式。例如，

```
x += 4;
```

等价于：

```
x = x + 4;
```

(一般地，要把运算符放在 = 号左边，如 *= 或 %=)。

注释：如果运算符得到一个值，其类型与左侧操作数的类型不同，就会发生强制类型转换。例如，如果 x 是一个 int，则以下语句

```
x += 3.5;
```

是合法的，将把 x 设置为 `(int)(x + 3.5)`。

3.5.5 自增与自减运算符

当然，程序员都知道加 1、减 1 是数值变量最常见的操作。在 Java 中，借鉴了 C 和 C++ 的做法，也提供了自增、自减运算符：`n++` 将变量 n 的当前值加 1，`n--` 则将 n 的值减 1。例如，以下代码：

```
int n = 12;
n++;
```

将 n 的值改为 13。由于这些运算符会改变变量的值，所以它们的操作数不能是数值。例如，`4++` 就不是一个合法的语句。

实际上，这些运算符有两种形式；上面介绍的是运算符放在操作数后面的“后缀”形式。还有一种“前缀”形式：`++n`。后缀和前缀形式都会使变量值加 1 或减 1。但用在表达式中时，二者就有区别了。前缀形式会先完成加 1；而后缀形式会使用变量原来的值。

```
int m = 7;
int n = 7;
int a = 2 * ++m; // now a is 16, m is 8
int b = 2 * n++; // now b is 14, n is 8
```

建议不要在表达式中使用 `++`，因为这样的代码很容易让人困惑，而且会带来烦人的 bug。

3.5.6 关系和 boolean 运算符

Java 包含丰富的关系运算符。要检测相等性，可以使用两个等号 `==`。例如，

```
3 == 7
```

的值为 `false`。

另外可以使用 != 检测不相等。例如，

`3 != 7`

的值为 true。

最后，还有经常使用的 < (小于)、> (大于)、<= (小于等于) 和 >= (大于等于) 运算符。

Java 沿用了 C++ 的做法，使用 `&&` 表示逻辑“与”运算符，使用 `||` 表示逻辑“或”运算符。从 `!=` 运算符可以想到，感叹号 `!` 就是逻辑非运算符。`&&` 和 `||` 运算符是按照“短路”方式来求值的：如果第一个操作数已经能够确定表达式的值，第二个操作数就不必计算了。如果用 `&&` 运算符合并两个表达式，

`expression1 && expression2`

而且已经计算得到第一个表达式的真值为 `false`，那么结果就不可能为 `true`。因此，第二个表达式就不必计算了。可以利用这一点来避免错误。例如，在下面的表达式中：

`x != 0 && 1 / x > x + y // no division by 0`

如果 `x` 等于 0，那么第二部分就不会计算。因此，如果 `x` 为 0，也就不会计算 `1 / x`，除以 0 的错误就不会出现。

类似地，如果第一个表达式为 `true`，`expression1 || expression2` 的值就自动为 `true`，而无需计算第二个表达式。

最后一点，Java 支持三元操作符 `? :`，这个操作符有时很有用。如果条件为 `true`，下面的表达式

`condition ? expression1 : expression2`

就为第一个表达式的值，否则计算为第二个表达式的值。例如，

`x < y ? x : y`

会返回 `x` 和 `y` 中较小的一个。

3.5.7 位运算符

处理整型类型时，可以直接对组成整型数值的各个位完成操作。这意味着可以使用掩码技术得到整数中的各个位。位运算符包括：

`& ("and") | ("or") ^ ("xor") ~ ("not")`

这些运算符按位模式处理。例如，如果 `n` 是一个整数变量，而且用二进制表示的 `n` 从右边数第 4 位为 1，则

`int fourthBitFromRight = (n & 0b1000) / 0b1000;`

会返回 1，否则返回 0。利用 `&` 并结合使用适当的 2 的幂，可以把其他位掩掉，而只保留其中的某一位。

 **注释：**应用在布尔值上时，`&` 和 `|` 运算符也会得到一个布尔值。这些运算符与 `&&` 和 `||` 运算符很类似，不过 `&` 和 `|` 运算符不采用“短路”方式来求值，也就是说，得到计算结果

之前两个操作数都需要计算。

另外，还有`>>`和`<<`运算符将位模式左移或右移。需要建立位模式来完成位掩码时，这两个运算符会很方便：

```
int fourthBitFromRight = (n & (1 << 3)) >> 3;
```

最后，`>>>`运算符会用0填充高位，这与`>>`不同，它会用符号位填充高位。不存在`<<<`运算符。

- ◆ 警告：移位运算符的右操作数要完成模32的运算（除非左操作数是long类型，在这种情况下需要对右操作数模64）。例如，`1 << 35`的值等同于`1 << 3`或8。
- ◆ C++注释：在C/C++中，不能保证`>>`是完成算术移位（扩展符号位）还是逻辑移位（填充0）。实现者可以选择其中更高效地任何一种做法。这意味着C/C++`>>`运算符对于负数生成的结果可能会依赖于具体的实现。Java则消除了这种不确定性。

3.5.8 括号与运算符级别

表3-4给出了运算符的优先级。如果不使用圆括号，就按照给出的运算符优先级次序进行计算。同一个级别的运算符按照从左到右的次序进行计算（除了表中给出的右结合运算符外。）例如，由于`&&`的优先级比`||`的优先级高，所以表达式

```
a && b || c
```

等价于

```
(a && b) || c
```

又因为`+=`是右结合运算符，所以表达式

```
a += b += c
```

等价于

```
a += (b += c)
```

也就是将`b += c`的结果（加上`c`之后的`b`）加到`a`上。

- ◆ C++注释：与C或C++不同，Java不使用逗号运算符。不过，可以在for语句的第一部分中使用逗号分隔表达式列表。

表3-4 运算符优先级

运 算 符	结合性
[].()(方法调用)	从左向右
<code>! ~ ++ -- + (一元运算) - (一元运算) () (强制类型转换) new</code>	从右向左
<code>* / %</code>	从左向右
<code>+ -</code>	从左向右
<code><< >> >>></code>	从左向右
<code>< <= > >= instanceof</code>	从左向右

(续)

运 算 符	结合性
<code>== !=</code>	从左向右
<code>&</code>	从左向右
<code>^</code>	从左向右
<code> </code>	从左向右
<code>&&</code>	从左向右
<code> </code>	从左向右
<code>?:</code>	从右向左
<code>= += -= *= /= %= &= = ^= <<= >>= >>>=</code>	从右向左

3.5.9 枚举类型

有时候，变量的取值只在一个有限的集合内。例如：销售的服装或比萨饼只有小、中、大和超大这四种尺寸。当然，可以将这些尺寸分别编码为 1、2、3、4 或 S、M、L、X。但这样存在着一定的隐患。在变量中很可能保存的是一个错误的值（如 0 或 m）。

针对这种情况，可以自定义枚举类型。枚举类型包括有限个命名的值。例如，

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

现在，可以声明这种类型的变量：

```
Size s = Size.MEDIUM;
```

Size 类型的变量只能存储这个类型声明中给定的某个枚举值，或者 null 值，null 表示这个变量没有设置任何值。

有关枚举类型的详细内容将在第 5 章介绍。

3.6 字符串

从概念上讲，Java 字符串就是 Unicode 字符序列。例如，串 “Java\u2122” 由 5 个 Unicode 字符 J、a、v、a 和 ™。Java 没有内置的字符串类型，而是在标准 Java 类库中提供了一个预定义类，很自然地叫做 String。每个用双引号括起来的字符串都是 String 类的一个实例：

```
String e = ""; // an empty string
String greeting = "Hello";
```

3.6.1 子串

String 类的 substring 方法可以从一个较大的字符串提取出一个子串。例如：

```
String greeting = "Hello";
String s = greeting.substring(0, 3);
```

创建了一个由字符 “Hel” 组成的字符串。

`substring` 方法的第二个参数是不想复制的第一个位置。这里要复制位置为 0、1 和 2（从 0 到 2，包括 0 和 2）的字符。在 `substring` 中从 0 开始计数，直到 3 为止，但不包含 3。

`substring` 的工作方式有一个优点：容易计算子串的长度。字符串 `s.substring(a, b)` 的长度为 `b-a`。例如，子串“Hel”的长度为 `3-0=3`。

3.6.2 拼接

与绝大多数的程序设计语言一样，Java 语言允许使用 + 号连接（拼接）两个字符串。

```
String expletive = "Expletive";
String PG13 = "deleted";
String message = expletive + PG13;
```

上述代码将“`Expletive deleted`”赋给变量 `message`（注意，单词之间没有空格，+ 号按照给定的次序将两个字符串拼接起来）。

当将一个字符串与一个非字符串的值进行拼接时，后者被转换成字符串（在第 5 章中可以看到，任何一个 Java 对象都可以转换成字符串）。例如：

```
int age = 13;
String rating = "PG" + age;
```

`rating` 设置为“`PG13`”。

这种特性通常用在输出语句中。例如：

```
System.out.println("The answer is " + answer);
```

这是一条合法的语句，并且将会打印出所希望的结果（因为单词 `is` 后面加了一个空格，输出时也会加上这个空格）。

如果需要把多个字符串放在一起，用一个定界符分隔，可以使用静态 `join` 方法：

```
String all = String.join(" / ", "S", "M", "L", "XL");
// all is the string "S / M / L / XL"
```

3.6.3 不可变字符串

`String` 类没有提供用于修改字符串的方法。如果希望将 `greeting` 的内容修改为“`Help!`”，不能直接地将 `greeting` 的最后两个位置的字符修改为‘`p`’和‘`!`’。这对于 C 程序员来说，将会感到无从下手。如何修改这个字符串呢？在 Java 中实现这项操作非常容易。首先提取需要的字符，然后再拼接上替换的字符串：

```
greeting = greeting.substring(0, 3) + "p!";
```

上面这条语句将 `greeting` 当前值修改为“`Help!`”。

由于不能修改 Java 字符串中的字符，所以在 Java 文档中将 `String` 类对象称为不可变字符串，如同数字 3 永远是数字 3 一样，字符串“`Hello`”永远包含字符 H、e、l、l 和 o 的代码单元序列，而不能修改其中的任何一个字符。当然，可以修改字符串变量 `greeting`，让它引用另外一个字符串，这就如同可以将存放 3 的数值变量改成存放 4 一样。

这样做是否会降低运行效率呢？看起来好像修改一个代码单元要比创建一个新字符串更加简洁。答案是：也对，也不对。的确，通过拼接“Hel”和“p!”来创建一个新字符串的效率确实不高。但是，不可变字符串却有一个优点：编译器可以让字符串共享。

为了弄清具体的工作方式，可以想象将各种字符串存放在公共的存储池中。字符串变量指向存储池中相应的位置。如果复制一个字符串变量，原始字符串与复制的字符串共享相同的字符。

总而言之，Java 的设计者认为共享带来的高效率远胜过于提取、拼接字符串所带来的低效率。查看一下程序会发现：很少需要修改字符串，而是往往需要对字符串进行比较（有一种例外情况，将来自于文件或键盘的单个字符或较短的字符串汇集成字符串。为此，Java 提供了一个独立的类，在 3.6.9 节中将详细介绍）。

C++ 注释：在 C 程序员第一次接触 Java 字符串的时候，常常会感到迷惑，因为他们总将字符串认为是字符型数组：

```
char greeting[] = "Hello";
```

这种认识是错误的，Java 字符串大致类似于 char* 指针，

```
char* greeting = "Hello";
```

当采用另一个字符串替换 greeting 的时候，Java 代码大致进行下列操作：

```
char* temp = malloc(6);
strncpy(temp, greeting, 3);
strncpy(temp + 3, "p!", 3);
greeting = temp;
```

的确，现在 greeting 指向字符串“Help!”。即使一名最顽固的 C 程序员也得承认 Java 语法要比一连串的 strncpy 调用舒适得多。然而，如果将 greeting 赋予另外一个值又会怎样呢？

```
greeting = "Howdy";
```

这样做会不会产生内存遗漏呢？毕竟，原始字符串放置在堆中。十分幸运，Java 将自动地进行垃圾回收。如果一块内存不再使用了，系统最终会将其回收。

对于一名使用 ANSI C++ 定义的 string 类的 C++ 程序员，会感觉使用 Java 的 String 类型更为舒适。C++ string 对象也自动地进行内存的分配与回收。内存管理是通过构造器、赋值操作和析构器显式执行的。然而，C++ 字符串是可修改的，也就是说，可以修改字符串中的单个字符。

3.6.4 检测字符串是否相等

可以使用 equals 方法检测两个字符串是否相等。对于表达式：

```
s.equals(t)
```

如果字符串 s 与字符串 t 相等，则返回 true；否则，返回 false。需要注意，s 与 t 可以是字符

串变量，也可以是字符串字面量。例如，下列表达式是合法的：

```
"Hello".equals(greeting)
```

要想检测两个字符串是否相等，而不区分大小写，可以使用 equalsIgnoreCase 方法。

```
"Hello".equalsIgnoreCase("hello")
```

一定不要使用 == 运算符检测两个字符串是否相等！这个运算符只能够确定两个字符串是否放置在同一个位置上。当然，如果字符串放置在同一个位置上，它们必然相等。但是，完全有可能将内容相同的多个字符串的拷贝放置在不同的位置上。

```
String greeting = "Hello"; //initialize greeting to a string
if (greeting == "Hello") . .
    // probably true
if (greeting.substring(0, 3) == "Hel") . .
    // probably false
```

如果虚拟机始终将相同的字符串共享，就可以使用 == 运算符检测是否相等。但实际上只有字符串常量是共享的，而 + 或 substring 等操作产生的结果并不是共享的。因此，千万不要使用 == 运算符测试字符串的相等性，以免在程序中出现糟糕的 bug。从表面上看，这种 bug 很像随机产生的间歇性错误。

C++ 注释：对于习惯使用 C++ 的 string 类的人来说，在进行相等性检测的时候一定要特别小心。C++ 的 string 类重载了 == 运算符以便检测字符串内容的相等性。可惜 Java 没有采用这种方式，它的字符串“看起来、感觉起来”与数值一样，但进行相等性测试时，其操作方式又类似于指针。语言的设计者本应该像对 + 那样也进行特殊处理，即重定义 == 运算符。当然，每一种语言都会存在一些不太一致的地方。

C 程序员从不使用 == 对字符串进行比较，而使用 strcmp 函数。Java 的 compareTo 方法与 strcmp 完全类似，因此，可以这样使用：

```
if (greeting.compareTo("Hello") == 0) . . .
```

不过，使用 equals 看起来更为清晰。

3.6.5 空串与 Null 串

空串 "" 是长度为 0 的字符串。可以调用以下代码检查一个字符串是否为空：

```
if (str.length() == 0)
```

或

```
if (str.equals(""))
```

空串是一个 Java 对象，有自己的串长度（0）和内容（空）。不过，String 变量还可以存放一个特殊的值，名为 null，这表示目前没有任何对象与该变量关联（关于 null 的更多信息请参见第 4 章）。要检查一个字符串是否为 null，要使用以下条件：

```
if (str == null)
```

有时要检查一个字符串既不是 null 也不为空串，这种情况下就需要使用以下条件：

```
if (str != null && str.length() != 0)
```

首先要检查 str 不为 null。在第 4 章会看到，如果在一个 null 值上调用方法，会出现错误。

3.6.6 码点与代码单元

Java 字符串由 char 值序列组成。从 3.3.3 节“char 类型”已经看到，char 数据类型是一个采用 UTF-16 编码表示 Unicode 码点的代码单元。大多数的常用 Unicode 字符使用一个代码单元就可以表示，而辅助字符需要一对代码单元表示。

`length` 方法将返回采用 UTF-16 编码表示的给定字符串所需要的代码单元数量。例如：

```
String greeting = "Hello";
int n = greeting.length(); // is 5.
```

要想得到实际的长度，即码点数量，可以调用：

```
int cpCount = greeting.codePointCount(0, greeting.length());
```

调用 `s.charAt(n)` 将返回位置 n 的代码单元，n 介于 0 ~ `s.length()`-1 之间。例如：

```
char first = greeting.charAt(0); // first is 'H'
char last = greeting.charAt(4); // last is 'o'
```

要想得到第 i 个码点，应该使用下列语句

```
int index = greeting.offsetByCodePoints(0, i);
int cp = greeting.codePointAt(index);
```

 **注释：**类似于 C 和 C++，Java 对字符串中的代码单元和码点从 0 开始计数。

为什么会对代码单元如此大惊小怪？请考虑下列语句：

∅ is the set of octonions

使用 UTF-16 编码表示字符∅ (U+1D546) 需要两个代码单元。调用

```
char ch = sentence.charAt(1)
```

返回的不是一个空格，而是∅的第二个代码单元。为了避免这个问题，不要使用 `char` 类型。这太底层了。

如果想要遍历一个字符串，并且依次查看每一个码点，可以使用下列语句：

```
int cp = sentence.codePointAt(i);
if (Character.isSupplementaryCodePoint(cp)) i += 2;
else i++;
```

可以使用下列语句实现回退操作：

```
i--;
if (Character.isSurrogate(sentence.charAt(i))) i--;
int cp = sentence.codePointAt(i);
```

显然，这很麻烦。更容易的办法是使用 `codePoints` 方法，它会生成一个 int 值的“流”，每个 int 值对应一个码点。(流将在卷Ⅱ的第 2 章中讨论)。可以将它转换为一个数组(见 3.10 节)，再完成遍历。

```
int[] codePoints = str.codePoints().toArray();
```

反之，要把一个码点数组转换为一个字符串，可以使用构造函数(我们将在第 4 章详细讨论构造函数和 new 操作符)。

```
String str = new String(codePoints, 0, codePoints.length);
```

3.6.7 String API

Java 中的 `String` 类包含了 50 多个方法。令人惊讶的是绝大多数都很有用，可以设想使用的频繁非常高。下面的 API 注释汇总了一部分最常用的方法。

注释：可以发现，本书中给出的 API 注释会有助于理解 Java 应用程序编程接口 (API)。每一个 API 的注释都以形如 `java.lang.String` 的类名开始。(`java.lang` 包的重要性将在第 4 章给出解释。) 类名之后是一个或多个方法的名字、解释和参数描述。

在这里，一般不列出某个类的所有方法，而是选择一些最常用的方法，并以简洁的方式给予描述。完整的方法列表请参看联机文档(请参看 3.6.8 节)。

这里还列出了所给类的版本号。如果某个方法是在这个版本之后添加的，就会给出一个单独的版本号。

API `java.lang.String` 1.0

- `char charAt (int index)`

返回给定位置的代码单元。除非对底层的代码单元感兴趣，否则不需要调用这个方法。

- `int codePointAt(int index) 5.0`

返回从给定位置开始的码点。

- `int offsetByCodePoints(int startIndex, int cpCount) 5.0`

返回从 `startIndex` 码点开始，位移 `cpCount` 后的码点索引。

- `int compareTo(String other)`

按照字典顺序，如果字符串位于 `other` 之前，返回一个负数；如果字符串位于 `other` 之后，返回一个正数；如果两个字符串相等，返回 0。

- `IntStream codePoints() 8`

将这个字符串的码点作为一个流返回。调用 `toArray` 将它们放在一个数组中。

- `new String(int[] codePoints, int offset, int count) 5.0`

用数组中从 `offset` 开始的 `count` 个码点构造一个字符串。

- `boolean equals(Object other)`

如果字符串与 `other` 相等，返回 `true`。

- **boolean equalsIgnoreCase(String other)**
如果字符串与 other 相等（忽略大小写），返回 true。
- **boolean startsWith(String prefix)**
- **boolean endsWith(String suffix)**
如果字符串以 suffix 开头或结尾，则返回 true。
- **int indexOf(String str)**
- **int indexOf(String str, int fromIndex)**
- **int indexOf(int cp)**
- **int indexOf(int cp, int fromIndex)**
返回与字符串 str 或代码点 cp 匹配的第一个子串的开始位置。这个位置从索引 0 或 fromIndex 开始计算。如果在原始串中不存在 str，返回 -1。
- **int lastIndexOf(String str)**
- **int lastIndexOf(String str, int fromIndex)**
- **int lastindex0f(int cp)**
- **int lastindex0f(int cp, int fromIndex)**
返回与字符串 str 或代码点 cp 匹配的最后一个子串的开始位置。这个位置从原始串尾端或 fromIndex 开始计算。
- **int length()**
返回字符串的长度。
- **int codePointCount(int startIndex, int endIndex) 5.0**
返回 startIndex 和 endIndex-1 之间的代码点数量。没有配成对的代用字符将计入代码点。
- **String replace(CharSequence oldString,CharSequence newString)**
返回一个新字符串。这个字符串用 newString 代替原始字符串中所有的 oldString。可以用 String 或 StringBuilder 对象作为 CharSequence 参数。
- **String substring(int beginIndex)**
- **String substring(int beginIndex, int endIndex)**
返回一个新字符串。这个字符串包含原始字符串中从 beginIndex 到串尾或 endIndex-1 的所有代码单元。
- **String toLowerCase()**
- **String toUpperCase()**
返回一个新字符串。这个字符串将原始字符串中的大写字母改为小写，或者将原始字符串中的所有小写字母改成了大写字母。
- **String trim()**
返回一个新字符串。这个字符串将删除了原始字符串头部和尾部的空格。
- **String join(CharSequence delimiter, CharSequence... elements) 8**
返回一个新字符串，用给定的定界符连接所有元素。

注释：在 API 注释中，有一些 CharSequence 类型的参数。这是一种接口类型，所有字符串都属于这个接口。第 6 章将介绍更多有关接口类型的内容。现在只需要知道只要看到一个 CharSequence 形参，完全可以传入 String 类型的实参。

3.6.8 阅读联机 API 文档

正如前面所看到的，String 类包含许多方法。而且，在标准库中有几千个类，方法数量更加惊人。要想记住所有的类和方法是一件不太不可能的事情。因此，学会使用在线 API 文档十分重要，从中可以查阅到标准类库中的所有类和方法。API 文档是 JDK 的一部分，它是 HTML 格式的。让浏览器指向安装 JDK 的 docs/api/index.html 子目录，就可以看到如图 3-2 所示的屏幕。

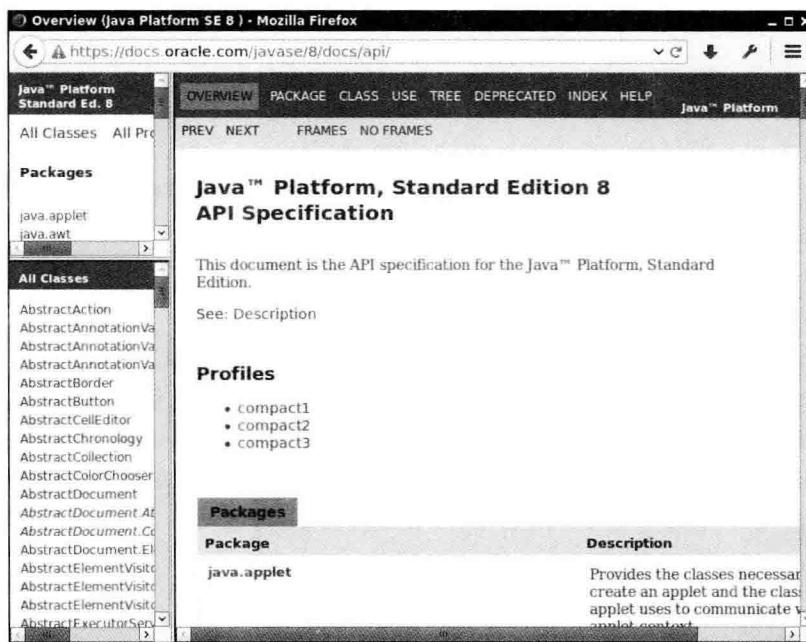


图 3-2 API 文档的三个窗格

可以看到，屏幕被分成三个窗框。在左上方的小窗框中显示了可使用的所有包。在它下面稍大的窗框中列出了所有的类。点击任何一个类名之后，这个类的 API 文档就会显示在右侧的大窗框中（请参看图 3-3）。例如，要获得有关 String 类方法的更多信息，可以滚动第二个窗框，直到看见 String 链接为止，然后点击这个链接。

接下来，滚动右面的窗框，直到看见按字母顺序排列的所有方法为止（请参看图 3-4）。点击任何一个方法名便可以查看这个方法的详细描述（参见图 3-5）。例如，如果点击 compareToIgnoreCase 链接，就会看到 compareToIgnoreCase 方法的描述。

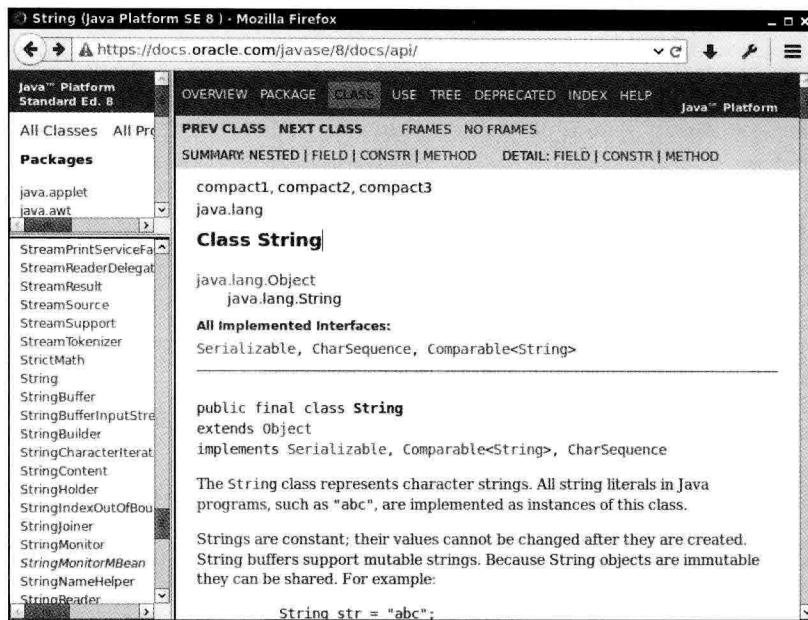


图 3-3 String 类的描述

提示：马上在浏览器中将 docs/api/index.html 页面建一个书签。

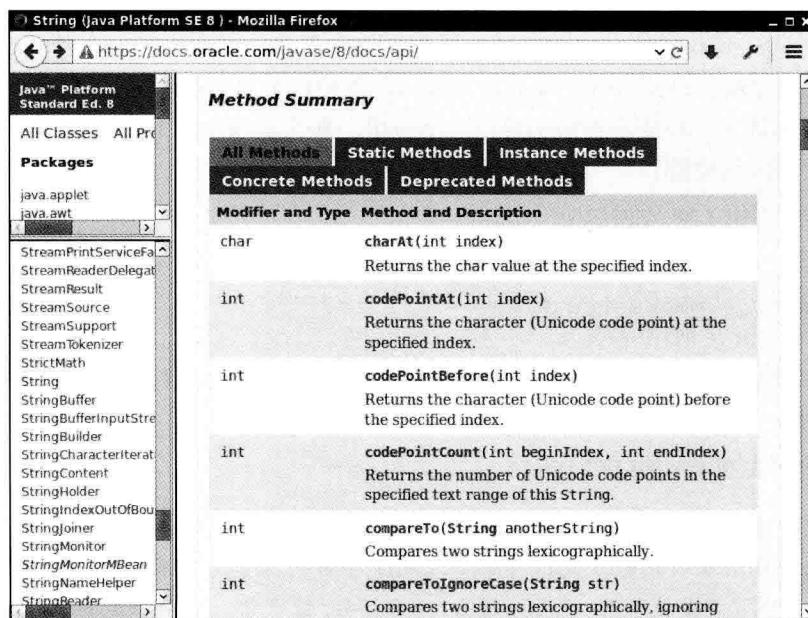


图 3-4 String 类方法的小结

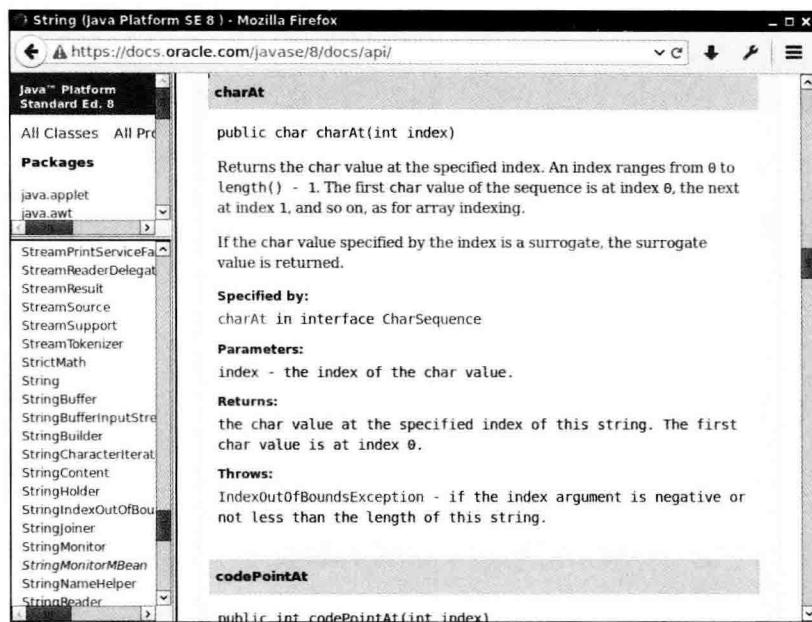


图 3-5 String 方法的详细描述

3.6.9 构建字符串

有些时候，需要由较短的字符串构建字符串，例如，按键或来自文件中的单词。采用字符串连接的方式达到此目的效率比较低。每次连接字符串，都会构建一个新的 String 对象，既耗时，又浪费空间。使用 StringBuilder 类就可以避免这个问题的发生。

如果需要用许多小段的字符串构建一个字符串，那么应该按照下列步骤进行。首先，构建一个空的字符串构建器：

```
StringBuilder builder = new StringBuilder();
```

当每次需要添加一部分内容时，就调用 append 方法。

```
builder.append(ch); // appends a single character
builder.append(str); // appends a string
```

在需要构建字符串时就调用 `toString` 方法，将可以得到一个 String 对象，其中包含了构建器中的字符序列。

```
String completedString = builder.toString();
```

注释：在 JDK5.0 中引入 `StringBuilder` 类。这个类的前身是 `StringBuffer`，其效率稍有些低，但允许采用多线程的方式执行添加或删除字符的操作。如果所有字符串在一个单线程中编辑（通常都是这样），则应该用 `StringBuilder` 替代它。这两个类的 API 是相同的。

下面的 API 注释包含了 `StringBuilder` 类中的重要方法。

API `java.lang.StringBuilder` 5.0

- `StringBuilder()`
构造一个空的字符串构建器。
- `int length()`
返回构建器或缓冲器中的代码单元数量。
- `StringBuilder append(String str)`
追加一个字符串并返回 this。
- `StringBuilder append(char c)`
追加一个代码单元并返回 this。
- `StringBuilder appendCodePoint(int cp)`
追加一个代码点，并将其转换为一个或两个代码单元并返回 this。
- `void setCharAt(int i,char c)`
将第 i 个代码单元设置为 c。
- `StringBuilder insert(int offset,String str)`
在 offset 位置插入一个字符串并返回 this。
- `StringBuilder insert(int offset,Char c)`
在 offset 位置插入一个代码单元并返回 this。
- `StringBuilder delete(int startIndex,int endIndex)`
删除偏移量从 startIndex 到 endIndex-1 的代码单元并返回 this。
- `String toString()`
返回一个与构建器或缓冲器内容相同的字符串。

3.7 输入输出

为了增加后面示例程序的趣味性，需要程序能够接收输入，并以适当的格式输出。当然，现代的程序都使用 GUI 收集用户的输入，然而，编写这种界面的程序需要使用较多的工具与技术，目前还不具备这些条件。主要原因是需要熟悉 Java 程序设计语言，因此只要有简单的用于输入输出的控制台就可以了。第 10 章~第 12 章将详细地介绍 GUI 程序设计。

3.7.1 读取输入

前面已经看到，打印输出到“标准输出流”（即控制台窗口）是一件非常容易的事情，只要调用 `System.out.println` 即可。然而，读取“标准输入流”`System.in` 就没有那么简单了。要想通过控制台进行输入，首先需要构造一个 `Scanner` 对象，并与“标准输入流”`System.in` 关联。

```
Scanner in = new Scanner(System.in);
```

(构造函数和 new 操作符将在第 4 章中详细地介绍。)

现在，就可以使用 Scanner 类的各种方法实现输入操作了。例如，`nextLine` 方法将输入一行。

```
System.out.print("What is your name? ");
String name = in.nextLine();
```

在这里，使用 `nextLine` 方法是因为在输入行中有可能包含空格。要想读取一个单词（以空白符作为分隔符），就调用

```
String firstName = in.next();
```

要想读取一个整数，就调用 `nextInt` 方法。

```
System.out.print("How old are you? ");
int age = in.nextInt();
```

与此类似，要想读取下一个浮点数，就调用 `nextDouble` 方法。

在程序清单 3-2 的程序中，询问用户名和年龄，然后打印一条如下格式的消息：

```
Hello, Cay. Next year, you'll be 57
```

最后，在程序的最开始添加上一行：

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

`Scanner` 类定义在 `java.util` 包中。当使用的类不是定义在基本 `java.lang` 包中时，一定要使用 `import` 指示字将相应的包加载进来。有关包与 `import` 指示字的详细描述请参看第 4 章。

程序清单 3-2 InputTest/InputTest.java

```
1 import java.util.*;
2 /**
3 * This program demonstrates console input.
4 * @version 1.10 2004-02-10
5 * @author Cay Horstmann
6 */
7
8 public class InputTest
9 {
10     public static void main(String[] args)
11     {
12         Scanner in = new Scanner(System.in);
13
14         // get first input
15         System.out.print("What is your name? ");
16         String name = in.nextLine();
17
18         // get second input
19         System.out.print("How old are you? ");
20         int age = in.nextInt();
21
22         // display output on console
23         System.out.println("Hello, " + name + ". Next year, you'll be " + (age + 1));
24     }
25 }
```

注释：因为输入是可见的，所以 Scanner 类不适用于从控制台读取密码。Java SE 6 特别引入了 Console 类实现这个目的。要想读取一个密码，可以采用下列代码：

```
Console cons = System.console();
String username = cons.readLine("User name: ");
char[] passwd = cons.readPassword("Password: ");
```

为了安全起见，返回的密码存放在一维字符数组中，而不是字符串中。在对密码进行处理之后，应该马上用一个填充值覆盖数组元素（数组处理将在 3.10 节介绍）。

采用 Console 对象处理输入不如采用 Scanner 方便。每次只能读取一行输入，而没有能够读取一个单词或一个数值的方法。

API java.util.Scanner 5.0

- **Scanner (InputStream in)**

用给定的输入流创建一个 Scanner 对象。

- **String nextLine()**

读取输入的下一行内容。

- **String next()**

读取输入的下一个单词（以空格作为分隔符）。

- **int nextInt()**

读取并转换下一个表示整数或浮点数的字符序列。

- **boolean hasNext()**

检测输入中是否还有其他单词。

- **boolean hasNextInt()**

- **boolean hasNextDouble()**

检测是否还有表示整数或浮点数的下一个字符序列。

API java.lang.System 1.0

- **static Console console() 6**

如果有可能进行交互操作，就通过控制台窗口为交互的用户返回一个 Console 对象，否则返回 null。对于任何一个通过控制台窗口启动的程序，都可使用 Console 对象。否则，其可用性将与所使用的系统有关。

API java.io.Console 6

- **static char[] readPassword(String prompt, Object...args)**

- **static String readLine(String prompt, Object...args)**

显示字符串 prompt 并且读取用户输入，直到输入行结束。args 参数可以用来提供输入格式。有关这部分内容将在下一节中介绍。

3.7.2 格式化输出

可以使用 `System.out.print(x)` 将数值 `x` 输出到控制台上。这条命令将以 `x` 对应的数据类型所允许的最大非 0 数字位数打印输出 `x`。例如：

```
double x = 10000.0 / 3.0;
System.out.print(x);
```

打印

3333.333333333335

如果希望显示美元、美分等符号，则有可能会出现问题。

在早期的 Java 版本中，格式化数值曾引起过一些争议。庆幸的是，Java SE 5.0 沿用了 C 语言库函数中的 `printf` 方法。例如，调用

```
System.out.printf("%8.2f", x);
```

可以用 8 个字符的宽度和小数点后两个字符的精度打印 `x`。也就是说，打印输出一个空格和 7 个字符，如下所示：

3333.33

在 `printf` 中，可以使用多个参数，例如：

```
System.out.printf("Hello, %s. Next year, you'll be %d", name, age);
```

每一个以 % 字符开始的格式说明符都用相应的参数替换。格式说明符尾部的转换符将指示被格式化的数值类型：f 表示浮点数，s 表示字符串，d 表示十进制整数。表 3-5 列出了所有转换符。

表 3-5 用于 `printf` 的转换符

转换符	类 型	举 例	转换符	类 型	举 例
d	十进制整数	159	s	字符串	Hello
x	十六进制整数	9f	c	字符	H
o	八进制整数	237	b	布尔	True
f	定点浮点数	15.9	h	散列码	42628b2
e	指数浮点数	1.59e+01	tx 或 Tx	日期 时间 (T 强制大写)	已经过时，应当改为使用 java.time 类，参见卷 II 第 6 章
g	通用浮点数	—	%	百分号	%
a	十六进制浮点数	0x1.fccdp3	n	与平台有关 的行分隔符	—

另外，还可以给出控制格式化输出的各种标志。表 3-6 列出了所有的标志。例如，逗号标志增加了分组的分隔符。即

```
System.out.printf("%.2f", 10000.0 / 3.0);
```

打印

3,333.33

可以使用多个标志，例如，“%, (.2f” 使用分组的分隔符并将负数括在括号内。

表 3-6 用于 printf 的标志

标 志	目 的	举 例
+	打印正数和负数的符号	+3333.33
空格	在正数之前添加空格	3333.33
0	数字前面补 0	003333.33
-	左对齐	3333.33
(将负数括在括号内	(3333.33)
,	添加分组分隔符	3,333.33
# (对于 f 格式)	包含小数点	3,333.
# (对于 x 或 0 格式)	添加前缀 0x 或 0	0xcafe
\$	给定被格式化的参数索引。例如，%1\$d, %1\$x 将以十进制和十六进制格式打印第 1 个参数	159 9F
<	格式化前面说明的数值。例如，%d%<x 以十进制和十六进制打印同一个数值	159 9F

注释：可以使用 s 转换符格式化任意的对象。对于任意实现了 Formattable 接口的对象都将调用 formatTo 方法；否则将调用 toString 方法，它可以将对象转换为字符串。在第 5 章中将讨论 toString 方法，在第 6 章中将讨论接口。

可以使用静态的 String.format 方法创建一个格式化的字符串，而不打印输出：

```
String message = String.format("Hello, %s. Next year, you'll be %d", name, age);
```

基于完整性的考虑，下面简略地介绍 printf 方法中日期与时间的格式化选项。在新代码中，应当使用卷 II 第 6 章中介绍的 java.time 包的方法。不过你可能会在遗留代码中看到 Date 类和相关的格式化选项。格式包括两个字母，以 t 开始，以表 3-7 中的任意字母结束。例如，

```
System.out.printf("%tc", new Date());
```

这条语句将用下面的格式打印当前的日期和时间：

```
Mon Feb 09 18:05:19 PST 2015
```

表 3-7 日期和时间的转换符

转换符	类 型	举 例
c	完整的日期和时间	Mon Feb 09 18:05:19 PST 2015
F	ISO 8601 日期	2015-02-09
D	美国格式的日期（月 / 日 / 年）	02/09/2015
T	24 小时时间	18:05:19

(续)

转换符	类 型	举 例
r	12 小时时间	06:05:19 pm
R	24 小时时间没有秒	18:05
Y	4 位数字的年 (前面补 0)	2015
y	年的后两位数字 (前面补 0)	15
C	年的前两位数字 (前面补 0)	20
B	月的完整拼写	February
b 或 h	月的缩写	Feb
m	两位数字的月 (前面补 0)	02
d	两位数字的日 (前面补 0)	09
e	两位数字的日 (前面不补 0)	9
A	星期几的完整拼写	Monday
a	星期几的缩写	Mon
j	三位数的年中的日子 (前面补 0), 在 001 到 366 之间	069
H	两位数字的小时 (前面补 0), 在 0 到 23 之间	18
k	两位数字的小时 (前面不补 0), 在 0 到 23 之间	18
I	两位数字的小时 (前面补 0), 在 0 到 12 之间	06
l	两位数字的小时 (前面不补 0), 在 0 到 12 之间	6
M	两位数字的分钟 (前面补 0)	05
S	两位数字的秒 (前面补 0)	19
L	三位数字的毫秒 (前面补 0)	047
N	九位数字的毫微秒 (前面补 0)	047000000
p	上午或下午的标志	pm
z	从 GMT 起, RFC822 数字位移	-0800
Z	时区	PST
s	从格林威治时间 1970-01-01 00:00:00 起的秒数	1078884319
Q	从格林威治时间 1970-01-01 00:00:00 起的毫秒数	1078884319047

从表 3-7 可以看到, 某些格式只给出了指定日期的部分信息。例如, 只有日期或月份。如果需要多次对日期操作才能实现对每一部分进行格式化的目的就太笨拙了。为此, 可以采用一个格式化的字符串指出要被格式化的参数索引。索引必须紧跟在 % 后面, 并以 \$ 终止。例如,

```
System.out.printf("%1$s %2$tB %2$te, %2$tY", "Due date:", new Date());
```

打印

```
Due date: February 9, 2015
```

还可以选择使用 < 标志。它指示前面格式说明中的参数将被再次使用。也就是说, 下列语句将产生与前面语句同样的输出结果:

```
System.out.printf("%s %tB %<te, %<tY", "Due date:", new Date());
```

 提示: 参数索引值从 1 开始, 而不是从 0 开始, %1\$... 对第 1 个参数格式化。这就避免了与 0 标志混淆。

现在，已经了解了 `printf` 方法的所有特性。图 3-6 给出了格式说明符的语法图。

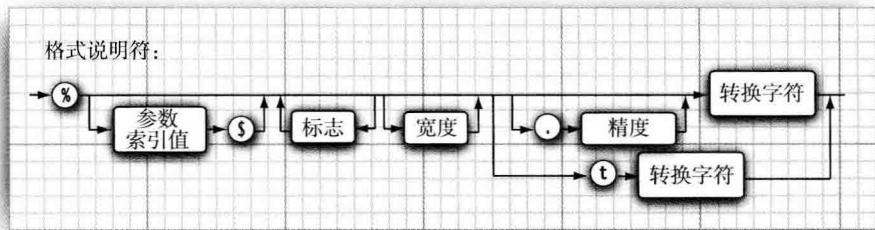


图 3-6 格式说明符语法

注释：许多格式化规则是本地环境特有的。例如，在德国，组分隔符是句号而不是逗号，`Monday` 被格式化为 `Montag`。在卷Ⅱ第 5 章中将介绍如何控制应用的国际化行为。

3.7.3 文件输入与输出

要想对文件进行读取，就需要一个用 `File` 对象构造一个 `Scanner` 对象，如下所示：

```
Scanner in = new Scanner(Paths.get("myfile.txt"), "UTF-8");
```

如果文件名中包含反斜杠符号，就要记住在每个反斜杠之前再加一个额外的反斜杠：“`c:\\mydirectory\\myfile.txt`”。

注释：在这里指定了 `UTF-8` 字符编码，这对于互联网上的文件很常见（不过并不是普遍适用）。读取一个文本文件时，要知道它的字符编码——更多信息参见卷Ⅱ第 2 章。如果省略字符编码，则会使用运行这个 Java 程序的机器的“默认编码”。这不是一个好主意，如果在不同的机器上运行这个程序，可能会有不同的表现。

现在，就可以利用前面介绍的任何一个 `Scanner` 方法对文件进行读取。

要想写入文件，就需要构造一个 `PrintWriter` 对象。在构造器中，只需要提供文件名：

```
PrintWriter out = new PrintWriter("myfile.txt", "UTF-8");
```

如果文件不存在，创建该文件。可以像输出到 `System.out` 一样使用 `print`、`println` 以及 `printf` 命令。

警告：可以构造一个带有字符串参数的 `Scanner`，但这个 `Scanner` 将字符串解释为数据，而不是文件名。例如，如果调用：

```
Scanner in = new Scanner("myfile.txt"); // ERROR?
```

这个 `scanner` 会将参数作为包含 10 个字符的数据：‘m’，‘y’，‘f’ 等。在这个示例中所显示的并不是人们所期望的效果。

注释：当指定一个相对文件名时，例如，“`myfile.txt`”，“`mydirectory/myfile.txt`” 或 “`./myfile`”。

txt”，文件位于 Java 虚拟机启动路径的相对位置。如果在命令行方式下用下列命令启动程序：

```
java MyProg
```

启动路径就是命令解释器的当前路径。然而，如果使用集成开发环境，那么启动路径将由 IDE 控制。可以使用下面的调用方式找到路径的位置：

```
String dir = System.getProperty("user.dir");
```

如果觉得定位文件比较烦恼，则可以考虑使用绝对路径，例如：“c:\\mydirectory\\myfile.txt” 或者 “/home/me/mydirectory/myfile.txt”。

正如读者所看到的，访问文件与使用 `System.in` 和 `System.out` 一样容易。要记住一点：如果用一个不存在的文件构造一个 `Scanner`，或者用一个不能被创建的文件名构造一个 `PrintWriter`，那么就会发生异常。Java 编译器认为这些异常比“被零除”异常更严重。在第 7 章中，将会学习各种处理异常的方式。现在，应该告知编译器：已经知道有可能出现“输入 / 输出”异常。这需要在 `main` 方法中用 `throws` 子句标记，如下所示：

```
public static void main(String[] args) throws IOException
{
    Scanner in = new Scanner(Paths.get("myfile.txt"), "UTF-8");
    ...
}
```

现在读者已经学习了如何读写包含文本数据的文件。对于更加高级的技术，例如，处理不同的字符编码、处理二进制数据、读取目录以及写压缩文件，请参看卷 II 第 2 章。

注释：当采用命令行方式启动一个程序时，可以利用 Shell 的重定向语法将任意文件关联到 `System.in` 和 `System.out`：

```
java MyProg < myfile.txt > output.txt
```

这样，就不必担心处理 `IOException` 异常了。

API `java.util.Scanner` 5.0

- `Scanner(File f)`
构造一个从给定文件读取数据的 `Scanner`。
- `Scanner(String data)`
构造一个从给定字符串读取数据的 `Scanner`。

API `java.io.PrintWriter` 1.1

- `PrintWriter(String fileName)`
构造一个将数据写入文件的 `PrintWriter`。文件名由参数指定。

API `java.nio.file.Paths` 7

- `static Path get(String pathname)`
根据给定的路径名构造一个 `Path`。

3.8 控制流程

与任何程序设计语言一样，Java 使用条件语句和循环结构确定控制流程。本节先讨论条件语句，然后讨论循环语句，最后介绍看似有些笨重的 switch 语句，当需要对某个表达式的多个值进行检测时，可以使用 switch 语句。

C++ 注释：Java 的控制流程结构与 C 和 C++ 的控制流程结构一样，只有很少的例外情况。没有 goto 语句，但 break 语句可以带标签，可以利用它实现从内层循环跳出的目的（这种情况 C 语言采用 goto 语句实现）。另外，还有一种变形的 for 循环，在 C 或 C++ 中没有这类循环。它有点类似于 C# 中的 foreach 循环。

3.8.1 块作用域

在深入学习控制结构之前，需要了解块（block）的概念。

块（即复合语句）是指由一对大括号括起来的若干条简单的 Java 语句。块确定了变量的作用域。一个块可以嵌套在另一个块中。下面就是在 main 方法块中嵌套另一个语句块的示例。

```
public static void main(String[] args)
{
    int n;
    . .
    {
        int k;
        . .
    } // k is only defined up to here
}
```

但是，不能在嵌套的两个块中声明同名的变量。例如，下面的代码就有错误，而无法通过编译：

```
public static void main(String[] args)
{
    int n;
    . .
    {
        int k;
        int n; // Error--can't redefine n in inner block
        . .
    }
}
```

C++ 注释：在 C++ 中，可以在嵌套的块中重定义一个变量。在内层定义的变量会覆盖在外层定义的变量。这样，有可能会导致程序设计错误，因此在 Java 中不允许这样做。

3.8.2 条件语句

在 Java 中，条件语句的格式为

```
if (condition) statement
```

这里的条件必须用括号括起来。

与绝大多数程序设计语言一样，Java 常常希望在某个条件为真时执行多条语句。在这种情况下，应该使用块语句（block statement），形式为

```
{
    statement1
    statement2
    ...
}
```

例如：

```
if (yourSales >= target)
{
    performance = "Satisfactory";
    bonus = 100;
}
```

当 yourSales 大于或等于 target 时，将执行括号中的所有语句（请参看图 3-7）。

注释：使用块（有时称为复合语句）可以在 Java 程序结构中原本只能放置一条（简单）语句的地方放置多条语句。

在 Java 中，更一般的条件语句格式如下所示（请参看图 3-8）：

```
if (condition) statement1 else statement2
```

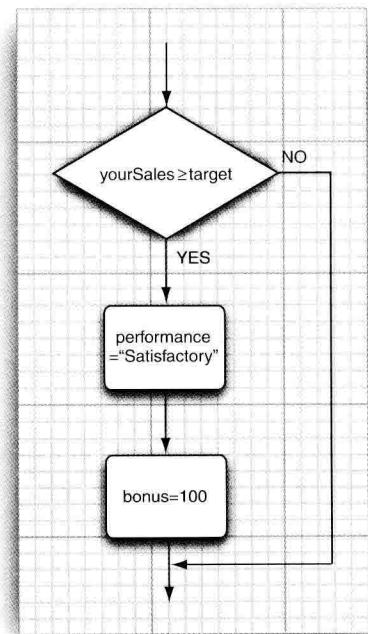


图 3-7 if 语句的流程图

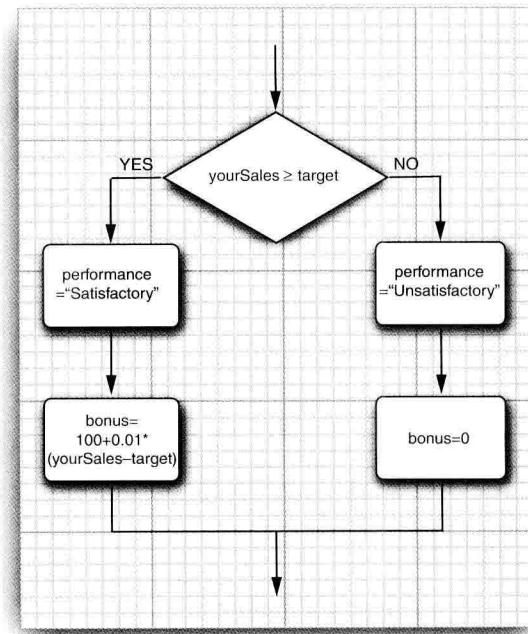


图 3-8 if/else 语句的流程图

例如：

```
if (yourSales >= target)
{
    performance = "Satisfactory";
    bonus = 100 + 0.01 * (yourSales - target);
}
else
{
    performance = "Unsatisfactory";
    bonus = 0;
}
```

其中 `else` 部分是可选的。`else` 子句与最邻近的 `if` 构成一组。因此，在语句

```
if (x <= 0) if (x == 0) sign = 0; else sign = -1;
```

中 `else` 与第 2 个 `if` 配对。当然，用一对括号将会使这段代码更加清晰：

```
if (x <= 0) { if (x == 0) sign = 0; else sign = -1; }
```

重复地交替出现 `if...else if...` 是一种很常见的情况（请参看图 3-9）。例如：

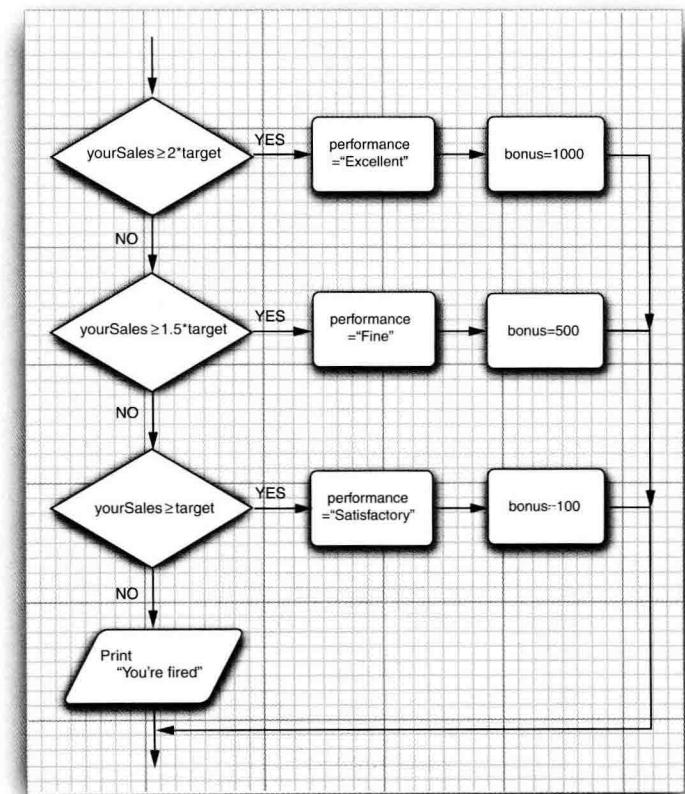


图 3-9 `if/else if` (多分支) 的流程图

```

if (yourSales >= 2 * target)
{
    performance = "Excellent";
    bonus = 1000;
}
else if (yourSales >= 1.5 * target)
{
    performance = "Fine";
    bonus = 500;
}
else if (yourSales >= target)
{
    performance = "Satisfactory";
    bonus = 100;
}
else
{
    System.out.println("You're fired");
}

```

3.8.3 循环

当条件为 true 时, while 循环执行一条语句 (也可以是一个语句块)。一般格式为
`while (condition) statement`

如果开始循环条件的值就为 false, 则 while 循环体一次也不执行 (请参看图 3-10)。

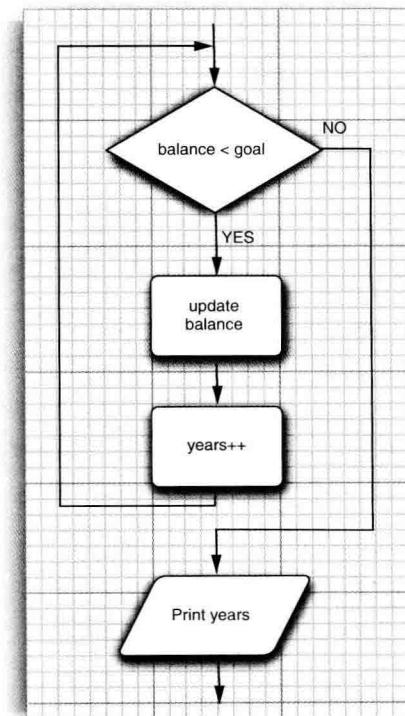


图 3-10 while 语句的流程图

程序清单 3-3 中的程序将计算需要多长时间才能够存储一定数量的退休金，假定每年存入相同数量的金额，而且利率是固定的。

程序清单 3-3 Retirement/Retirement.java

```

1 import java.util.*;
2
3 /**
4  * This program demonstrates a <code>while</code> loop.
5  * @version 1.20 2004-02-10
6  * @author Cay Horstmann
7 */
8 public class Retirement
9 {
10    public static void main(String[] args)
11    {
12        // read inputs
13        Scanner in = new Scanner(System.in);
14
15        System.out.print("How much money do you need to retire? ");
16        double goal = in.nextDouble();
17
18        System.out.print("How much money will you contribute every year? ");
19        double payment = in.nextDouble();
20
21        System.out.print("Interest rate in %: ");
22        double interestRate = in.nextDouble();
23
24        double balance = 0;
25        int years = 0;
26
27        // update account balance while goal isn't reached
28        while (balance < goal)
29        {
30            // add this year's payment and interest
31            balance += payment;
32            double interest = balance * interestRate / 100;
33            balance += interest;
34            years++;
35        }
36
37        System.out.println("You can retire in " + years + " years.");
38    }
39 }
```

在这个示例中，增加了一个计数器，并在循环体中更新当前的累积数量，直到总值超过目标值为止。

```

while (balance < goal)
{
    balance += payment;
    double interest = balance * interestRate / 100;
    balance += interest;
    years++;
}
System.out.println(years + " years.");
```

(千万不要使用这个程序安排退休计划。这里忽略了通货膨胀和所期望的生活水准。)

while 循环语句首先检测循环条件。因此，循环体中的代码有可能不被执行。如果希望循环体至少执行一次，则应该将检测条件放在最后。使用 do/while 循环语句可以实现这种操作方式。它的语法格式为：

```
do statement while (condition);
```

这种循环语句先执行语句（通常是一个语句块），再检测循环条件；然后重复语句，再检测循环条件，以此类推。在程序清单 3-4 中，首先计算退休账户中的余额，然后再询问是否打算退休：

```
do
{
    balance += payment;
    double interest = balance * interestRate / 100;
    balance += interest;
    year++;
    // print current balance
    .
    .
    // ask if ready to retire and get input
    .
}
while (input.equals("N"));
```

只要用户回答“N”，循环就重复执行（见图 3-11）。这是一个需要至少执行一次的循环的很好示例，因为用户必须先看到余额才能知道是否满足退休所用。

程序清单 3-4 Retirement2/Retirement2.java

```
1 import java.util.*;
2
3 /**
4  * This program demonstrates a <code>do/while</code> loop.
5  * @version 1.20 2004-02-10
6  * @author Cay Horstmann
7 */
8 public class Retirement2
9 {
10     public static void main(String[] args)
11     {
12         Scanner in = new Scanner(System.in);
13
14         System.out.print("How much money will you contribute every year? ");
15         double payment = in.nextDouble();
16
17         System.out.print("Interest rate in %: ");
18         double interestRate = in.nextDouble();
19
20         double balance = 0;
21         int year = 0;
22
23         String input;
24
25         // update account balance while user isn't ready to retire
26         do
27         {
28             // add this year's payment and interest
```

```

29     balance += payment;
30     double interest = balance * interestRate / 100;
31     balance += interest;
32
33     year++;
34
35     // print current balance
36     System.out.printf("After year %d, your balance is %.2f\n", year, balance);
37
38     // ask if ready to retire and get input
39     System.out.print("Ready to retire? (Y/N) ");
40     input = in.next();
41 }
42 while (input.equals("N"));
43 }
44 }
```

3.8.4 确定循环

for 循环语句是支持迭代的一种通用结构，利用每次迭代之后更新的计数器或类似的变量来控制迭代次数。如图 3-12 所示，下面的程序将数字 1 ~ 10 输出到屏幕上。

```
for (int i = 1; i <= 10; i++)
    System.out.println(i);
```

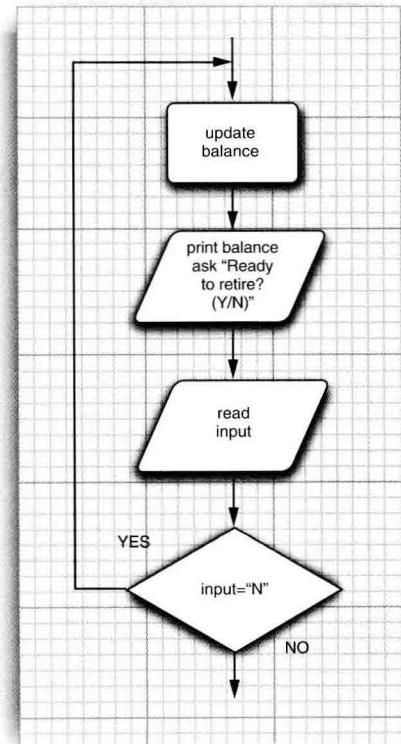


图 3-11 do/while 语句的流程图

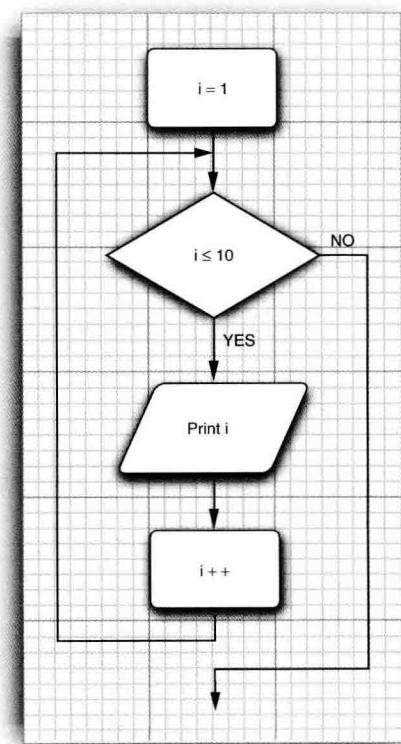


图 3-12 for 语句的流程图

for 语句的第 1 部分通常用于对计数器初始化；第 2 部分给出每次新一轮循环执行前要检测的循环条件；第 3 部分指示如何更新计数器。

与 C++ 一样，尽管 Java 允许在 for 循环的各个部分放置任何表达式，但有一条不成文的规则：for 语句的 3 个部分应该对同一个计数器变量进行初始化、检测和更新。若不遵守这一规则，编写的循环常常晦涩难懂。

即使遵守了这条规则，也还有可能出现很多问题。例如，下面这个倒计数的循环：

```
for (int i = 10; i > 0; i--)
    System.out.println("Counting down . . . " + i);
System.out.println("Blastoff!");
```

 **警告：**在循环中，检测两个浮点数是否相等需要格外小心。下面的 for 循环

```
for (double x = 0; x != 10; x += 0.1) . . .
```

可能永远不会结束。由于舍入的误差，最终可能得不到精确值。例如，在上面的循环中，因为 0.1 无法精确地用二进制表示，所以，x 将从 9.999 999 999 999 98 跳到 10.099 999 999 999 98。

当在 for 语句的第 1 部分中声明了一个变量之后，这个变量的作用域就为 for 循环的整个循环体。

```
for (int i = 1; i <= 10; i++)
{
    . .
}
// i no longer defined here
```

特别指出，如果在 for 语句内部定义一个变量，这个变量就不能在循环体之外使用。因此，如果希望在 for 循环体之外使用循环计数器的最终值，就要确保这个变量在循环语句的前面且在外部声明！

```
int i;
for (i = 1; i <= 10; i++)
{
    . .
}
// i is still defined here
```

另一方面，可以在各自独立的不同 for 循环中定义同名的变量：

```
for (int i = 1; i <= 10; i++)
{
    . .
}
. .
for (int i = 11; i <= 20; i++) // OK to define another variable named i
{
    . .
}
```

for 循环语句只不过是 while 循环的一种简化形式。例如，

```
for (int i = 10; i > 0; i--)
    System.out.println("Counting down . . . " + i);
```

可以重写为：

```
int i = 10;
while (i > 0)
{
    System.out.println("Counting down . . . " + i);
    i--;
}
```

程序清单 3-5 给出了一个应用 for 循环的典型示例。这个程序用来计算抽奖中奖的概率。例如，如果必须从 1 ~ 50 之间的数字中取 6 个数字来抽奖，那么会有 $(50 \times 49 \times 48 \times 47 \times 46 \times 45) / (1 \times 2 \times 3 \times 4 \times 5 \times 6)$ 种可能的结果，所以中奖的几率是 1/15 890 700。祝你好运！

程序清单 3-5 LotteryOdds/LotteryOdds.java

```
1 import java.util.*;
2
3 /**
4 * This program demonstrates a <code>for</code> loop.
5 * @version 1.20 2004-02-10
6 * @author Cay Horstmann
7 */
8 public class LotteryOdds
9 {
10    public static void main(String[] args)
11    {
12        Scanner in = new Scanner(System.in);
13
14        System.out.print("How many numbers do you need to draw? ");
15        int k = in.nextInt();
16
17        System.out.print("What is the highest number you can draw? ");
18        int n = in.nextInt();
19
20        /*
21         * compute binomial coefficient n*(n-1)*(n-2)*...*(n-k+1)/(1*2*3*...*k)
22         */
23
24        int lotteryOdds = 1;
25        for (int i = 1; i <= k; i++)
26            lotteryOdds = lotteryOdds * (n - i + 1) / i;
27
28        System.out.println("Your odds are 1 in " + lotteryOdds + ". Good luck!");
29    }
30}
```

一般情况下，如果从 n 个数字中抽取 k 个数字，就可以使用下列公式得到结果。

$$\frac{n \times (n-1) \times (n-2) \times \cdots \times (n-k+1)}{1 \times 2 \times 3 \times 4 \times \cdots \times k}$$

下面的 for 循环语句计算了上面这个公式的值：

```
int lotteryOdds = 1;
for (int i = 1; i <= k; i++)
    lotteryOdds = lotteryOdds * (n - i + 1) / i;
```

注释：3.10.1 节将会介绍“通用 for 循环”（又称为 for each 循环），这是 Java SE 5.0 新增加的一种循环结构。

3.8.5 多重选择：switch 语句

在处理多个选项时，使用 if/else 结构显得有些笨拙。Java 有一个与 C/C++ 完全一样的 switch 语句。

例如，如果建立一个如图 3-13 所示的包含 4 个选项的菜单系统，可以使用下列代码：

```
Scanner in = new Scanner(System.in);
System.out.print("Select an option (1, 2, 3, 4) ");
int choice = in.nextInt();
switch (choice)
{
    case 1:
        . . .
        break;
    case 2:
        . . .
        break;
    case 3:
        . . .
        break;
    case 4:
        . . .
        break;
    default:
        // bad input
        . . .
        break;
}
```

switch 语句将从与选项值相匹配的 case 标签处开始执行直到遇到 break 语句，或者执行到 switch 语句的结束处为止。如果没有相匹配的 case 标签，而有 default 子句，就执行这个子句。

！警告：有可能触发多个 case 分支。如果在 case 分支语句的末尾没有 break 语句，那么就会接着执行下一个 case 分支语句。这种情况相当危险，常常会引发错误。为此，我们在程序中从不使用 switch 语句。

如果你比我们更喜欢 switch 语句，编译代码时可以考虑加上 -Xlint:fallthrough 选项，如下所示：

```
javac -Xlint:fallthrough Test.java
```

这样一来，如果某个分支最后缺少一个 break 语句，编译器就会给出一个警告消息。

如果你确实正是想使用这种“直通式”（fallthrough）行为，可以为其外围方法加一个标注 @SuppressWarnings("fallthrough")。这样就不会对这个方法生成警告了。（标注是为

编译器或处理 Java 源文件或类文件的工具提供信息的一种机制。我们将在卷Ⅱ的第 8 章详细讨论标注。)

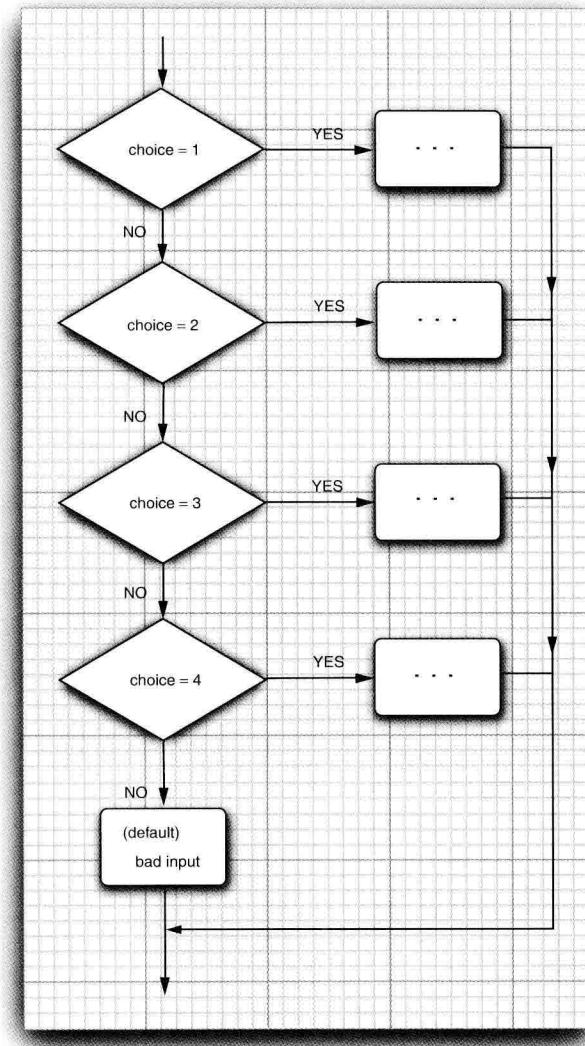


图 3-13 `switch` 语句的流程图

`case` 标签可以是：

- 类型为 `char`、`byte`、`short` 或 `int` 的常量表达式。
- 枚举常量。
- 从 Java SE 7 开始，`case` 标签还可以是字符串字面量。

例如：

```

String input = . . .;
switch (input.toLowerCase())
{
    case "yes": // OK since Java SE 7
        . . .
        break;
    . . .
}

```

当在 switch 语句中使用枚举常量时，不必在每个标签中指明枚举名，可以由 switch 的表达式值确定。例如：

```

Size sz = . . .;
switch (sz)
{
    case SMALL: // no need to use Size.SMALL
        . . .
        break;
    . . .
}

```

3.8.6 中断控制流程语句

尽管 Java 的设计者将 goto 作为保留字，但实际上并没有打算在语言中使用它。通常，使用 goto 语句被认为是一种拙劣的程序设计风格。当然，也有一些程序员认为反对 goto 的呼声似乎有些过分（例如，Donald Knuth 就曾编著过一篇名为《Structured Programming with goto statements》的著名文章）。这篇文章说：无限制地使用 goto 语句确实是导致错误的根源，但在有些情况下，偶尔使用 goto 跳出循环还是有益处的。Java 设计者同意这种看法，甚至在 Java 语言中增加了一条带标签的 break，以此来支持这种程序设计风格。

下面首先看一下不带标签的 break 语句。与用于退出 switch 语句的 break 语句一样，它也可以用于退出循环语句。例如，

```

while (years <= 100)
{
    balance += payment;
    double interest = balance * interestRate / 100;
    balance += interest;
    if (balance >= goal) break;
    years++;
}

```

在循环开始时，如果 $years > 100$ ，或者在循环体中 $balance \geq goal$ ，则退出循环语句。当然，也可以在不使用 break 的情况下计算 years 的值，如下所示：

```

while (years <= 100 && balance < goal)
{
    balance += payment;
    double interest = balance * interestRate / 100;
    balance += interest;
    if (balance < goal)
        years++;
}

```

但是需要注意，在这个版本中，检测了两次 `balance < goal`。为了避免重复检测，有些程序员更加偏爱使用 `break` 语句。

与 C++ 不同，Java 还提供了一种带标签的 `break` 语句，用于跳出多重嵌套的循环语句。有时候，在嵌套很深的循环语句中会发生一些不可预料的事情。此时可能更加希望跳到嵌套的所有循环语句之外。通过添加一些额外的条件判断实现各层循环的检测很不方便。

这里有一个示例说明了 `break` 语句的工作状态。请注意，标签必须放在希望跳出的最外层循环之前，并且必须紧跟一个冒号。

```
Scanner in = new Scanner(System.in);
int n;
read_data:
while ( . . . ) // this loop statement is tagged with the label
{
    . .
    for ( . . . ) // this inner loop is not labeled
    {
        System.out.print("Enter a number >= 0: ");
        n = in.nextInt();
        if (n < 0) // should never happen-can't go on
            break read_data;
        // break out of read_data loop
    }
    // this statement is executed immediately after the labeled break
    if (n < 0) // check for bad situation
    {
        // deal with bad situation
    }
    else
    {
        // carry out normal processing
    }
}
```

如果输入有误，通过执行带标签的 `break` 跳转到带标签的语句块末尾。对于任何使用 `break` 语句的代码都需要检测循环是正常结束，还是由 `break` 跳出。

注释：事实上，可以将标签应用到任何语句中，甚至可以应用到 `if` 语句或者块语句中，如下所示：

```
label:
{
    . .
    if (condition) break label; // exits block
    . .
}
// jumps here when the break statement executes
```

因此，如果希望使用一条 `goto` 语句，并将一个标签放在想要跳到的语句块之前，就可以使用 `break` 语句！当然，并不提倡使用这种方式。另外需要注意，只能跳出语句块，而不能跳入语句块。

最后，还有一个 `continue` 语句。与 `break` 语句一样，它将中断正常的控制流程。`continue` 语句将控制转移到最内层循环的首部。例如：

```
Scanner in = new Scanner(System.in);
while (sum < goal)
{
    System.out.print("Enter a number: ");
    n = in.nextInt();
    if (n < 0) continue;
    sum += n; // not executed if n < 0
}
```

如果 $n < 0$ ，则 `continue` 语句越过了当前循环体的剩余部分，立刻跳到循环首部。

如果将 `continue` 语句用于 `for` 循环中，就可以跳到 `for` 循环的“更新”部分。例如，下面这个循环：

```
for (count = 1; count <= 100; count++)
{
    System.out.print("Enter a number, -1 to quit: ");
    n = in.nextInt();
    if (n < 0) continue;
    sum += n; // not executed if n < 0
}
```

如果 $n < 0$ ，则 `continue` 语句跳到 `count++` 语句。

还有一种带标签的 `continue` 语句，将跳到与标签匹配的循环首部。

 提示：许多程序员容易混淆 `break` 和 `continue` 语句。这些语句完全是可选的，即不使用它们也可以表达同样的逻辑含义。在本书中，将不使用 `break` 和 `continue`。

3.9 大数值

如果基本的整数和浮点数精度不能够满足需求，那么可以使用 `java.math` 包中的两个很有用的类：`BigInteger` 和 `BigDecimal`。这两个类可以处理包含任意长度数字序列的数值。`BigInteger` 类实现了任意精度的整数运算，`BigDecimal` 实现了任意精度的浮点数运算。

使用静态的 `valueOf` 方法可以将普通的数值转换为大数值：

```
BigInteger a = BigInteger.valueOf(100);
```

遗憾的是，不能使用人们熟悉的算术运算符（如：`+` 和 `*`）处理大数值。而需要使用大数值类中的 `add` 和 `multiply` 方法。

```
BigInteger c = a.add(b); // c = a + b
BigInteger d = c.multiply(b.add(BigInteger.valueOf(2))); // d = c * (b + 2)
```

 C++ 注释：与 C++ 不同，Java 没有提供运算符重载功能。程序员无法重定义 `+` 和 `*` 运算符，使其应用于 `BigInteger` 类的 `add` 和 `multiply` 运算。Java 语言的设计者确实为字符串的连接重载了 `+` 运算符，但没有重载其他的运算符，也没有给 Java 程序员在自己的类中重载运算符的机会。

程序清单 3-6 是对程序清单 3-5 中彩概率程序的改进，使其可以采用大数值进行运算。假设你被邀请参加抽奖活动，并从 490 个可能的数值中抽取 60 个，这个程序将会得到中彩概率 $1/71639584346199555741511622254009293341171761278926349349351013459481104668848$ 。祝你好运！

程序清单 3-6 BigIntegerTest/BigIntegerTest.java

```

1 import java.math.*;
2 import java.util.*;
3
4 /**
5 * This program uses big numbers to compute the odds of winning the grand prize in a lottery.
6 * @version 1.20 2004-02-10
7 * @author Cay Horstmann
8 */
9 public class BigIntegerTest
10 {
11     public static void main(String[] args)
12     {
13         Scanner in = new Scanner(System.in);
14
15         System.out.print("How many numbers do you need to draw? ");
16         int k = in.nextInt();
17
18         System.out.print("What is the highest number you can draw? ");
19         int n = in.nextInt();
20
21         /*
22          * compute binomial coefficient n*(n-1)*(n-2)*...*(n-k+1)/(1*2*3*...*k)
23         */
24
25         BigInteger lotteryOdds = BigInteger.valueOf(1);
26
27         for (int i = 1; i <= k; i++)
28             lotteryOdds = lotteryOdds.multiply(BigInteger.valueOf(n - i + 1)).divide(
29                 BigInteger.valueOf(i));
30
31         System.out.println("Your odds are 1 in " + lotteryOdds + ". Good luck!");
32     }
33 }
```

在程序清单 3-5 中，用于计算的语句是

```
lotteryOdds = lotteryOdds * (n - i + 1) / i;
```

如果使用大数值，则相应的语句为：

```
lotteryOdds = lotteryOdds.multiply(BigInteger.valueOf(n - i + 1)).divide(BigInteger.valueOf(i));
```

API java.math.BigInteger 1.1

- `BigInteger add(BigInteger other)`
- `BigInteger subtract(BigInteger other)`

- `BigInteger multiply(BigInteger other)`
- `BigInteger divide(BigInteger other)`
- `BigInteger mod(BigInteger other)`
返回这个大整数和另一个大整数 other 的和、差、积、商以及余数。
- `int compareTo(BigInteger other)`
如果这个大整数与另一个大整数 other 相等，返回 0；如果这个大整数小于另一个大整数 other，返回负数；否则，返回正数。
- `static BigInteger valueOf(long x)`
返回值等于 x 的大整数。

API**java.math.BigInteger 1.1**

- `BigDecimal add(BigDecimal other)`
- `BigDecimal subtract(BigDecimal other)`
- `BigDecimal multiply(BigDecimal other)`
- `BigDecimal divide(BigDecimal other RoundingMode mode) 5.0`
返回这个大实数与另一个大实数 other 的和、差、积、商。要想计算商，必须给出舍入方式 (rounding mode)。RoundingMode.HALF_UP 是在学校中学习的四舍五入方式 (即，数值 0 到 4 舍去，数值 5 到 9 进位)。它适用于常规的计算。有关其他的舍入方式请参看 API 文档。
- `int compareTo(BigDecimal other)`
如果这个大实数与另一个大实数相等，返回 0；如果这个大实数小于另一个大实数，返回负数；否则，返回正数。
- `static BigDecimal valueOf(long x)`
- `static BigDecimal valueOf(long x,int scale)`
返回值为 x 或 $x / 10^{scale}$ 的一个大实数。

3.10 数组

数组是一种数据结构，用来存储同一类型值的集合。通过一个整型下标可以访问数组中的每一个值。例如，如果 a 是一个整型数组，`a[i]` 就是数组中下标为 i 的整数。

在声明数组变量时，需要指出数组类型（数据元素类型紧跟 []）和数组变量的名字。下面声明了整型数组 a：

```
int[] a;
```

不过，这条语句只声明了变量 a，并没有将 a 初始化为一个真正的数组。应该使用 new 运算符创建数组。

```
int[] a = new int[100];
```

这条语句创建了一个可以存储 100 个整数的数组。数组长度不要求是常量：new int[n] 会创建一个长度为 n 的数组。

注释：可以使用下面两种形式声明数组

```
int[] a;
```

或

```
int a[];
```

大多数 Java 应用程序员喜欢使用第一种风格，因为它将类型 int[]（整型数组）与变量名分开了。

这个数组的下标从 0 ~ 99（不是 1 ~ 100）。一旦创建了数组，就可以给数组元素赋值。例如，使用一个循环：

```
int[] a = new int[100];
for (int i = 0; i < 100; i++)
    a[i] = i; // fills the array with numbers 0 to 99
```

创建一个数字数组时，所有元素都初始化为 0。boolean 数组的元素会初始化为 false。对象数组的元素则初始化为一个特殊值 null，这表示这些元素（还）未存放任何对象。初学者对此可能有些不解。例如，

```
String[] names = new String[10];
```

会创建一个包含 10 个字符串的数组，所有字符串都为 null。如果希望这个数组包含空串，可以为元素指定空串：

```
for (int i = 0; i < 10; i++) names[i] = "";
```

警告：如果创建了一个 100 个元素的数组，并且试图访问元素 a[100]（或任何在 0 ~ 99 之外的下标），程序就会引发“array index out of bounds”异常而终止执行。

要想获得数组中的元素个数，可以使用 array.length。例如，

```
for (int i = 0; i < a.length; i++)
    System.out.println(a[i]);
```

一旦创建了数组，就不能再改变它的大小（尽管可以改变每一个数组元素）。如果经常需要在运行过程中扩展数组的大小，就应该使用另一种数据结构——数组列表（array list）有关数组列表的详细内容请参看第 5 章。

3.10.1 for each 循环

Java 有一种功能很强的循环结构，可以用来依次处理数组中的每个元素（其他类型的元素集合亦可）而不必为指定下标值而分心。

这种增强的 for 循环的语句格式为：

```
for (variable : collection) statement
```

定义一个变量用于暂存集合中的每一个元素，并执行相应的语句（当然，也可以是语句块）。collection 这一集合表达式必须是一个数组或者是一个实现了 Iterable 接口的类对象（例如 ArrayList）。有关数组列表的内容将在第 5 章中讨论，有关 Iterable 接口的内容将在第 9 章中讨论。

例如，

```
for (int element : a)
    System.out.println(element);
```

打印数组 a 的每一个元素，一个元素占一行。

这个循环应该读作“循环 a 中的每一个元素”（for each element in a）。Java 语言的设计者认为应该使用诸如 foreach、in 这样的关键字，但这种循环语句并不是最初就包含在 Java 语言中的，而是后来添加进去的，并且没有人打算废除已经包含同名（例如 System.in）方法或变量的旧代码。

当然，使用传统的 for 循环也可以获得同样的效果：

```
for (int i = 0; i < a.length; i++)
    System.out.println(a[i]);
```

但是，for each 循环语句显得更加简洁、更不易出错（不必为下标的起始值和终止值而操心）。

注释：for each 循环语句的循环变量将会遍历数组中的每个元素，而不需要使用下标值。

如果需要处理一个集合中的所有元素，for each 循环语句对传统循环语句所进行的改进更是叫人称赞不已。然而，在很多场合下，还是需要使用传统的 for 循环。例如，如果不希望遍历集合中的每个元素，或者在循环内部需要使用下标值等。

提示：有个更加简单的方式打印数组中的所有值，即利用 Arrays 类的 `toString` 方法。调用 `Arrays.toString(a)`，返回一个包含数组元素的字符串，这些元素被放置在括号内，并用逗号分隔，例如，“[2,3,5,7,11,13]”。要想打印数组，可以调用

```
System.out.println(Arrays.toString(a));
```

3.10.2 数组初始化以及匿名数组

在 Java 中，提供了一种创建数组对象并同时赋予初始值的简化书写形式。下面是一个例子：

```
int[] smallPrimes = { 2, 3, 5, 7, 11, 13 };
```

请注意，在使用这种语句时，不需要调用 `new`。

甚至还可以初始化一个匿名的数组：

```
new int[] { 17, 19, 23, 29, 31, 37 }
```

这种表示法将创建一个新数组并利用括号中提供的值进行初始化，数组的大小就是初始值的个数。使用这种语法形式可以在不创建新变量的情况下重新初始化一个数组。例如：

```
smallPrimes = new int[] { 17, 19, 23, 29, 31, 37 };
```

这是下列语句的简写形式：

```
int[] anonymous = { 17, 19, 23, 29, 31, 37 };
smallPrimes = anonymous;
```

注释：在 Java 中，允许数组长度为 0。在编写一个结果为数组的方法时，如果碰巧结果为空，则这种语法形式就显得非常有用。此时可以创建一个长度为 0 的数组：

```
new elementType[0]
```

注意，数组长度为 0 与 null 不同。

3.10.3 数组拷贝

在 Java 中，允许将一个数组变量拷贝给另一个数组变量。这时，两个变量将引用同一个数组：

```
int[] luckyNumbers = smallPrimes;
luckyNumbers[5] = 12; // now smallPrimes[5] is also 12
```

图 3-14 显示了拷贝的结果。如果希望将一个数组的所有值拷贝到一个新的数组中去，就要使用 Arrays 类的 copyOf 方法：

```
int[] copiedLuckyNumbers = Arrays.copyOf(luckyNumbers, luckyNumbers.length);
```

第 2 个参数是新数组的长度。这个方法通常用来增加数组的大小：

```
luckyNumbers = Arrays.copyOf(luckyNumbers, 2 * luckyNumbers.length);
```

如果数组元素是数值型，那么多余的元素将被赋值为 0；如果数组元素是布尔型，则将赋值为 false。相反，如果长度小于原始数组的长度，则只拷贝最前面的数据元素。

C++ 注释：Java 数组与 C++ 数组在堆栈上有很大不同，但基本上与分配在堆（heap）上的数组指针一样。也就是说，

```
int[] a = new int[100]; // Java
```

不同于

```
int a[100]; // C++
```

而等同于

```
int* a = new int[100]; // C++
```

Java 中的 [] 运算符被预定义为检查数组边界，而且没有指针运算，即不能通过 a 加 1 得到数组的下一个元素。

3.10.4 命令行参数

前面已经看到多个使用 Java 数组的示例。每一个 Java 应用程序都有一个带 String arg[]

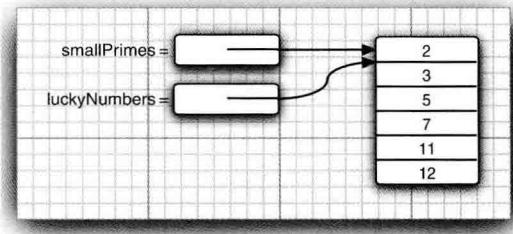


图 3-14 拷贝一个数组变量

参数的 main 方法。这个参数表明 main 方法将接收一个字符串数组，也就是命令行参数。

例如，看一看下面这个程序：

```
public class Message
{
    public static void main(String[] args)
    {
        if (args.length == 0 || args[0].equals("-h"))
            System.out.print("Hello,");
        else if (args[0].equals("-g"))
            System.out.print("Goodbye,");
        // print the other command-line arguments
        for (int i = 1; i < args.length; i++)
            System.out.print(" " + args[i]);
        System.out.println("!");
    }
}
```

如果使用下面这种形式运行这个程序：

```
java Message -g cruel world
```

args 数组将包含下列内容：

```
args[0]: "-g"
args[1]: "cruel"
args[2]: "world"
```

这个程序将显示下列信息：

```
Goodbye, cruel world!
```

C++ 注释：在 Java 应用程序的 main 方法中，程序名并没有存储在 args 数组中。例如，当使用下列命令运行程序时

```
java Message -h world
```

args[0] 是 “-h”，而不是 “Message” 或 “java”。

3.10.5 数组排序

要想对数值型数组进行排序，可以使用 Arrays 类中的 sort 方法：

```
int[] a = new int[10000];
...
Arrays.sort(a)
```

这个方法使用了优化的快速排序算法。快速排序算法对于大多数数据集合来说都是效率比较高的。Arrays 类还提供了几个使用很便捷的方法，在稍后的 API 注释中将介绍它们。

程序清单 3-7 中的程序用到了数组，它产生一个抽彩游戏中的随机数值组合。假如抽彩是从 49 个数值中抽取 6 个，那么程序可能的输出结果为：

```
Bet the following combination. It'll make you rich!
```

```
4
7
8
```

```

19
30
44

```

要想选择这样一个随机的数值集合，就要首先将数值 1, 2, ..., n 存入数组 numbers 中：

```

int[] numbers = new int[n];
for (int i = 0; i < numbers.length; i++)
    numbers[i] = i + 1;

```

而用第二个数组存放抽取出来的数值：

```
int[] result = new int[k];
```

现在，就可以开始抽取 k 个数值了。Math.random 方法将返回一个 0 到 1 之间（包含 0、不包含 1）的随机浮点数。用 n 乘以这个浮点数，就可以得到从 0 到 n-1 之间的一个随机数。

```
int r = (int) (Math.random() * n);
```

下面将 result 的第 i 个元素设置为 numbers[r] 存放的数值，最初是 r+1。但正如所看到的，numbers 数组的内容在每一次抽取之后都会发生变化。

```
result[i] = numbers[r];
```

现在，必须确保不会再次抽取到那个数值，因为所有抽彩的数值必须不相同。因此，这里用数组中的最后一个数值改写 number[r]，并将 n 减 1。

```

numbers[r] = numbers[n - 1];
n--;

```

关键在于每次抽取的都是下标，而不是实际的值。下标指向包含尚未抽取过的数组元素。在抽取了 k 个数值之后，就可以对 result 数组进行排序了，这样可以让输出效果更加清晰：

```

Arrays.sort(result);
for (int r : result)
    System.out.println(r);

```

程序清单 3-7 LotteryDrawing/LotteryDrawing.java

```

1 import java.util.*;
2
3 /**
4  * This program demonstrates array manipulation.
5  * @version 1.20 2004-02-10
6  * @author Cay Horstmann
7 */
8 public class LotteryDrawing
9 {
10     public static void main(String[] args)
11     {
12         Scanner in = new Scanner(System.in);
13
14         System.out.print("How many numbers do you need to draw? ");
15         int k = in.nextInt();
16
17         System.out.print("What is the highest number you can draw? ");
18         int n = in.nextInt();

```

```

19
20     // fill an array with numbers 1 2 3 . . . n
21     int[] numbers = new int[n];
22     for (int i = 0; i < numbers.length; i++)
23         numbers[i] = i + 1;
24
25     // draw k numbers and put them into a second array
26     int[] result = new int[k];
27     for (int i = 0; i < result.length; i++)
28     {
29         // make a random index between 0 and n - 1
30         int r = (int) (Math.random() * n);
31
32         // pick the element at the random location
33         result[i] = numbers[r];
34
35         // move the last element into the random location
36         numbers[r] = numbers[n - 1];
37         n--;
38     }
39
40     // print the sorted array
41     Arrays.sort(result);
42     System.out.println("Bet the following combination. It'll make you rich!");
43     for (int r : result)
44         System.out.println(r);
45 }
46 }
```

API `java.util.Arrays 1.2`

- `static String toString(type[] a)` 5.0

返回包含 a 中数据元素的字符串，这些数据元素被放在括号内，并用逗号分隔。

参数: a 类型为 int、long、short、char、byte、boolean、float 或 double 的数组。

- `static type copyOf(type[] a, int length)` 6

- `static type copyOfRange(type[] a, int start, int end)` 6

返回与 a 类型相同的一个数组，其长度为 length 或者 end-start，数组元素为 a 的值。

参数: a 类型为 int、long、short、char、byte、boolean、float 或 double 的数组。

start 起始下标 (包含这个值)。

end 终止下标 (不包含这个值)。这个值可能大于 a.length。在这种情况下，结果为 0 或 false。

length 拷贝的数据元素长度。如果 length 值大于 a.length，结果为 0 或 false；否则，数组中只有前面 length 个数据元素的拷贝值。

- `static void sort(type[] a)`

采用优化的快速排序算法对数组进行排序。

参数: a 类型为 int、long、short、char、byte、boolean、float 或 double 的数组。

- `static int binarySearch(type[] a, type v)`

- **static int binarySearch(type[] a, int start, int end, type v)** 6

采用二分搜索算法查找值 v。如果查找成功，则返回相应的下标值；否则，返回一个负数值 r。-r-1 是为保持 a 有序 v 应插入的位置。

参数：a 类型为 int、long、short、char、byte、boolean、float 或 double 的有序数组。

start 起始下标（包含这个值）。

end 终止下标（不包含这个值）。

v 同 a 的数据元素类型相同的值。

- **static void fill(type[] a, type v)**

将数组的所有数据元素值设置为 v。

参数：a 类型为 int、long、short、char、byte、boolean、float 或 double 的数组。

v 与 a 数据元素类型相同的一个值。

- **static boolean equals(type[] a, type[] b)**

如果两个数组大小相同，并且下标相同的元素都对应相等，返回 true。

参数：a、b 类型为 int、long、short、char、byte、boolean、float 或 double 的两个数组。

3.10.6 多维数组

多维数组将使用多个下标访问数组元素，它适用于表示表格或更加复杂的排列形式。这一节的内容可以先跳过，等到需要使用这种存储机制时再返回来学习。

假设需要建立一个数值表，用来显示在不同利率下投资 \$10,000 会增长多少，利息每年兑现，而且又被用于投资（见表 3-8）。

表 3-8 不同利率下的投资增长情况

10%	11%	12%	13%	14%	15%
10 000.00	10 000.00	10 000.00	10 000.00	10 000.00	10 000.00
11 000.00	11 100.00	11 200.00	11 300.00	11 400.00	11 500.00
12 100.00	12 321.00	12 544.00	12 769.00	12 996.00	13 225.00
13 310.00	13 676.31	14 049.28	14 428.97	14 815.44	15 208.75
14 641.00	15 180.70	15 735.19	16 304.74	16 889.60	17 490.06
16 105.10	16 850.58	17 623.42	18 424.35	19 254.15	20 113.57
17 715.61	18 704.15	19 738.23	20 819.52	21 949.73	23 130.61
19 487.17	20 761.60	22 106.81	23 526.05	25 022.69	26 600.20
21 435.89	23 045.38	24 759.63	26 584.44	28 525.86	30 590.23
23 579.48	25 580.37	27 730.79	30 040.42	32 519.49	35 178.76

可以使用一个二维数组（也称为矩阵）存储这些信息。这个数组被命名为 balances。

在 Java 中，声明一个二维数组相当简单。例如：

```
double[][] balances;
```

与一维数组一样，在调用 new 对多维数组进行初始化之前不能使用它。在这里可以这样初始化：

```
balances = new double[NYEARS][NRATES];
```

另外，如果知道数组元素，就可以不调用 new，而直接使用简化的书写形式对多维数组进行初始化。例如：

```
int[][] magicSquare =
{
    {16, 3, 2, 13},
    {5, 10, 11, 8},
    {9, 6, 7, 12},
    {4, 15, 14, 1}
};
```

一旦数组被初始化，就可以利用两个方括号访问每个元素，例如，balances[i][j]。

在示例程序中用到了一个存储利率的一维数组 interest 与一个存储余额的二维数组 balances。一维用于表示年，另一维用于表示利率，最初使用初始余额来初始化这个数组的第一行：

```
for (int j = 0; j < balances[0].length; j++)
    balances[0][j] = 10000;
```

然后，按照下列方式计算其他行：

```
for (int i = 1; i < balances.length; i++)
{
    for (int j = 0; j < balances[i].length; j++)
    {
        double oldBalance = balances[i - 1][j];
        double interest = . . .;
        balances[i][j] = oldBalance + interest;
    }
}
```

程序清单 3-8 给出了完整的程序。

 **注释：**for each 循环语句不能自动处理二维数组的每一个元素。它是按照行，也就是一维数组处理的。要想访问二维数组 a 的所有元素，需要使用两个嵌套的循环，如下所示：

```
for (double[] row : a)
    for (double value : row)
        do something with value
```

 **提示：**要想快速地打印一个二维数组的数据元素列表，可以调用：

```
System.out.println(Arrays.deepToString(a));
```

输出格式为：

```
[[16, 3, 2, 13], [5, 10, 11, 8], [9, 6, 7, 12], [4, 15, 14, 1]]
```

程序清单 3-8 CompoundInterest/CompoundInterest.java

```
1  /**
2   * This program shows how to store tabular data in a 2D array.
3   * @version 1.40 2004-02-10
4   * @author Cay Horstmann
5   */
6  public class CompoundInterest
7  {
8      public static void main(String[] args)
9      {
10         final double STARTRATE = 10;
11         final int NRATES = 6;
12         final int NYEARS = 10;
13
14         // set interest rates to 10 . . . 15%
15         double[] interestRate = new double[NRATES];
16         for (int j = 0; j < interestRate.length; j++)
17             interestRate[j] = (STARTRATE + j) / 100.0;
18
19         double[][] balances = new double[NYEARS][NRATES];
20
21         // set initial balances to 10000
22         for (int j = 0; j < balances[0].length; j++)
23             balances[0][j] = 10000;
24
25         // compute interest for future years
26         for (int i = 1; i < balances.length; i++)
27         {
28             for (int j = 0; j < balances[i].length; j++)
29             {
30                 // get last year's balances from previous row
31                 double oldBalance = balances[i - 1][j];
32
33                 // compute interest
34                 double interest = oldBalance * interestRate[j];
35
36                 // compute this year's balances
37                 balances[i][j] = oldBalance + interest;
38             }
39         }
40
41         // print one row of interest rates
42         for (int j = 0; j < interestRate.length; j++)
43             System.out.printf("%9.0f%%", 100 * interestRate[j]);
44
45         System.out.println();
46
47         // print balance table
48         for (double[] row : balances)
49         {
50             // print table row
51             for (double b : row)
52                 System.out.printf("%10.2f", b);
53
54             System.out.println();
55         }
```

```

56     }
57 }
```

3.10.7 不规则数组

到目前为止，读者所看到的数组与其他程序设计语言中提供的数组没有多大区别。但实际存在着一些细微的差异，而这正是 Java 的优势所在：Java 实际上没有多维数组，只有一维数组。多维数组被解释为“数组的数组。”

例如，在前面的示例中，balances 数组实际上是一个包含 10 个元素的数组，而每个元素又是一个由 6 个浮点数组成的数组（请参看图 3-15）。

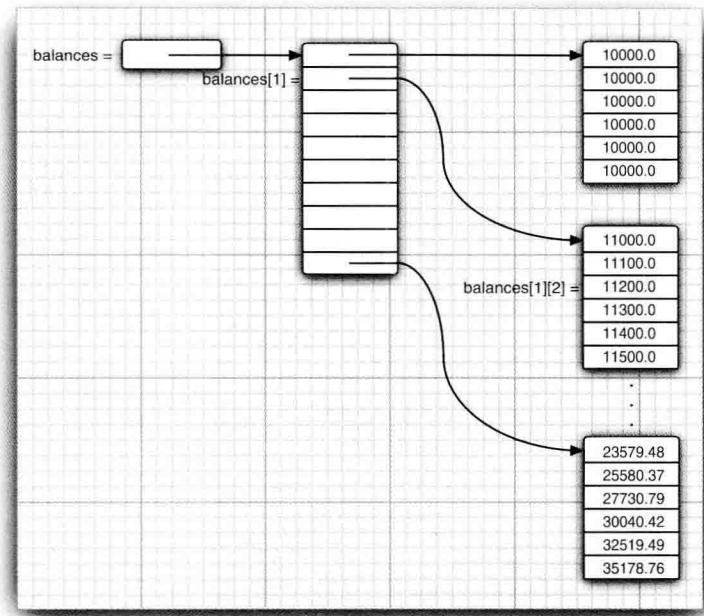


图 3-15 一个二维数组

表达式 `balances[i]` 引用第 i 个子数组，也就是二维表的第 i 行。它本身也是一个数组，`balances[i][j]` 引用这个数组的第 j 项。

由于可以单独地存取数组的某一行，所以可以让两行交换。

```

double[] temp = balances[i];
balances[i] = balances[i + 1];
balances[i + 1] = temp;
```

还可以方便地构造一个“不规则”数组，即数组的每一行有不同的长度。下面是一个典型的示例。在这个示例中，创建一个数组，第 i 行第 j 列将存放“从 i 个数值中抽取 j 个数值”产生的结果。

```

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1

```

由于 j 不可能大于 i ，所以矩阵是三角形的。第 i 行有 $i + 1$ 个元素（允许抽取 0 个元素，也是一种选择）。要想创建一个不规则的数组，首先需要分配一个具有所含行数的数组。

```
int[][] odds = new int[NMAX + 1][];
```

接下来，分配这些行。

```
for (int n = 0; n < NMAX; n++)
    odds[n] = new int[n + 1];
```

在分配了数组之后，假定没有超出边界，就可以采用通常的方式访问其中的元素了。

```
for (int n = 0; n < odds.length; n++)
    for (int k = 0; k < odds[n].length; k++)
    {
        // compute lotteryOdds
        .
        .
        odds[n][k] = lotteryOdds;
    }
```

程序清单 3-9 给出了完整的程序。

C++ 注释：在 C++ 中，Java 声明

```
double[][] balances = new double[10][6]; // Java
```

不同于

```
double balances[10][6]; // C++
```

也不同于

```
double (*balances)[6] = new double[10][6]; // C++
```

而是分配了一个包含 10 个指针的数组：

```
double** balances = new double*[10]; // C++
```

然后，指针数组的每一个元素被填充了一个包含 6 个数字的数组：

```
for (i = 0; i < 10; i++)
    balances[i] = new double[6];
```

庆幸的是，当创建 `new double[10][6]` 时，这个循环将自动地执行。当需要不规则的数组时，只能单独地创建行数组。

程序清单 3-9 LotteryArray/LotteryArray.java

```

1 /**
2 * This program demonstrates a triangular array.
3 * @version 1.20 2004-02-10

```

```
4  * @author Cay Horstmann
5  */
6 public class LotteryArray
7 {
8     public static void main(String[] args)
9     {
10         final int NMAX = 10;
11
12         // allocate triangular array
13         int[][] odds = new int[NMAX + 1][][];
14         for (int n = 0; n <= NMAX; n++)
15             odds[n] = new int[n + 1];
16
17         // fill triangular array
18         for (int n = 0; n < odds.length; n++)
19             for (int k = 0; k < odds[n].length; k++)
20             {
21                 /*
22                  * compute binomial coefficient n*(n-1)*(n-2)*...*(n-k+1)/(1*2*3*...*k)
23                  */
24                 int lotteryOdds = 1;
25                 for (int i = 1; i <= k; i++)
26                     lotteryOdds = lotteryOdds * (n - i + 1) / i;
27
28                 odds[n][k] = lotteryOdds;
29             }
30
31         // print triangular array
32         for (int[] row : odds)
33         {
34             for (int odd : row)
35                 System.out.printf("%4d", odd);
36             System.out.println();
37         }
38     }
39 }
```

现在，已经看到了 Java 语言的基本程序结构，下一章将介绍 Java 中的面向对象的程序设计。