

第3章 Java 的基本程序设计结构

- ▲ 一个简单的 Java 程序
- ▲ 注释
- ▲ 数据类型
- ▲ 变量与常量
- ▲ 运算符

- ▲ 字符串
- ▲ 输入与输出
- ▲ 控制流程
- ▲ 大数
- ▲ 数组

现在，你应该已经成功地安装了 JDK，并且能够执行第 2 章中的示例程序。下面开始介绍程序设计。本章主要介绍如何在 Java 中实现基本程序设计概念（如数据类型、分支以及循环）。

3.1 一个简单的 Java 程序

下面仔细分析一个最简单的 Java 程序，它只是向控制台打印一个消息：

```
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，但这些内容在所有 Java 应用中都会出现，因此还是值得花一些时间来研究的。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 章中将会更详细地介绍访问修饰符的有关内容。关键字 `class` 表明 Java 程序中的全部内容都包含在类中。下一章你会更多地了解 Java 类，不过现在只需要将类看作是程序逻辑的一个容器，定义了应用程序的行为。正如第 1 章所述，类是所有 Java 应用的构建模块。Java 程序中的所有内容都必须放在类中。

关键字 `class` 后面紧跟类名。Java 中定义类名的规则很宽松。类名必须以字母开头，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（保留字列表请参见附录）。

标准命名约定为：类名是以大写字母开头的名词（类名 `FirstSample` 就使用了这个命名约定）。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有时称为骆驼命名法 (camel case)。以其自身为例，应该写为 CamelCase。

源代码的文件名必须与公共类的类名相同，并用.java 作为扩展名。因此，存储这个代码时，文件名必须为 FirstSample.java（再次提醒大家注意，大小写非常重要，千万不能写成 firstsample.java）。

如果已经正确地命名文件，并且源代码中没有任何录入错误，在编译这个源代码之后，会得到一个包含这个类字节码的文件。Java 编译器将这个字节码文件自动地命名为 FirstSample.class，并存储在源文件所在的同一个目录下。最后，使用下面这个命令运行这个程序：

```
java FirstSample
```

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

当使用以下命令

```
java ClassName
```

运行一个已编译的程序时，Java 虚拟机总是从指定类中 main 方法的代码开始执行（这里的“方法”就是 Java 中对“函数”的叫法），因此为了能够执行代码，类的源代码中必须包含一个 main 方法。当然，也可以将你自己的方法添加到类中，并从 main 方法调用这些方法（第 4 章将介绍如何编写你自己的方法）。

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> 阅读或下载）。

不过，即使 main 方法没有声明为 public，有些版本的 Java 解释器也会执行 Java 程序。有个程序员报告了这个 bug。如果感兴趣，可以访问 <https://bugs.openjdk.java.net/browse/JDK-4252539> 查看这个 bug。1999 年，这个 bug 被标记为“关闭，不予修复”（Closed, Will not be fixed）。Sun 公司的一个工程师解释说：Java 虚拟机规范并没有强制要求 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 1.4 及以后的版本中，Java 解释器强制要求 main 方法必须是 public。

当然，让质量保证工程师对 bug 报告做出决定不仅让人生疑，也让他们自己很头疼，因为他们的工作量很大，而且他们对 Java 的所有细节也未必了解得很清楚。不过，Sun 公司在 Java 开源很久以前就把 bug 报告及其解决方案放在网站上让所有人监督检查，这是一个非常了不起的举措。

注意源代码中的大括号 {}。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必须以“{”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对齐。不过，由于 Java 编译器会忽略空白符，所以你可以选用自己喜欢的任何大括号风格。

我们暂且不考虑关键字 static void，只把它们当作编译 Java 程序必要的部分就行了。在学习完第 4 章后，这些部分的作用就会揭晓。现在需要记住的重点是：每个 Java 应用都必须有一个 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，你一定知道类是什么。Java 的类与 C++ 的类很相似，但有些差异还是会使人感到困惑。例如，Java 中的所有函数都是某个类的方法（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（shell）类。你可能对 C++ 中的静态成员函数（static member function）也很熟悉。它们是类中定义的成员函数，而且不对对象进行操作。Java 中的 main 方法总是静态的。最后，与 C/C++ 一样，关键字 void 表示这个方法不返回值，但与 C/C++ 不同的是，main 方法不会为操作系统返回一个“退出码”。如果 main 方法正常退出，那么 Java 程序的退出码为 0，表示成功地运行了程序。如果要以其他退出码终止程序，则需要使用 System.exit 方法。

来看以下代码片段：

```
{
    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，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 语句看成是这个语言中的句子。在 Java 中，每个语句必须用分号结束。特别需要说明，回车不是语句的结束标志，因此，如果需要，一条语句可以跨多行。

这个 main 方法体中只包含一条语句，其功能是将一个文本行输出到控制台。

在这里，我们使用 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)

这等价于一个函数调用。

在这个示例中，println 方法接收一个字符串参数。这个方法将这个字符串参数显示在控制台上。然后，终止这个输出行，所以每次调用 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，都使用双引号界定字符串。（本章稍后会介绍更多有关字符串的知识。）

与其他程序设计语言中的函数一样，Java 中的方法可以没有参数，也可以有一个或多个参数（有的程序员把参数叫作实参（argument））。即使一个方法没有参数，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。可以使用下面的语句来调用：

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

注释：System.out 还有一个 print 方法，它不在输出之后增加换行符。例如，System.out.print("Hello") 打印 "Hello" 之后不换行，下一个输出将紧跟在字母 "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，而不必担心代码膨胀。在 Java 中，有 3 种标记注释的方式。最常用的方式是使用 //。使用这种方式时，从 // 开始到本行结尾都是注释。

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

当需要更长的注释时，可以在每一行注释的前面加 //，或者也可以使用 /* 和 */ 注释界定符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生成文档。这种注释以 /** 开始，以 */ 结束。在程序清单 3-1 中可以看到这种注释。有关这种注释以及自动生成文档的更多内容请参见第 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包含一个 */ 界定符。

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个类型。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 种字符类型 char（用于表示 Unicode 编码的代码单元，请参见 3.3.3 节“char 类型”）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，所谓的“大数”（big number）是 Java 对象，而不是一个基本 Java 类型。本章稍后将会详细地介绍如何使用大数。

3.3.1 整型

整型用于表示没有小数部分的数，可以是负数。Java 提供了 4 种整型，如表 3-1 所示。

表 3-1 Java 整型

类 型	存储需求	取值范围
int	4 字节	-2 147 483 648 ~ 2 147 483 647 (略高于 20 亿)
short	2 字节	-32 768 ~ 32 767
long	8 字节	-9 223 372 036 854 775 808 ~ 9 223 372 036 854 775 807
byte	1 字节	-128 ~ 127

在通常情况下，int 类型最常用。但如果想要表示整个地球的居住人口，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存储空间有限时的大数组。

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时（或者甚至在同一个平台中不同操作系统之间移植时）让程序员头疼的主要问题。与此相反，C 和 C++ 程序会针对不同的处理器选择最高效的整型，这样一来，一个在 32 位处理器上运行得很好的 C 程序在 16 位系统上运行时可能会发生整数溢出。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是固定的。

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 对应十进制中的 8）。显然，八进制表示法比较容易混淆，所以很少有程序员使用八进制常数。

加上前缀 0b 或 0B 还可以写二进制数。例如，0b1001 就是 9。另外，可以为数字字面量加下画线，如用 1_000_000（或 0b1111_0100_0010_0100_0000）表示 100 万。这些下画线只是为了让人更易读。Java 编译器会去除这些下画线。

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，整数占 2 字节；不过，在 32 位处理器上（比如 Pentium 或 SPARC），整数则为 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，这给编写跨平台程序带来了很大难度。在 Java 中，所有数值类型的大小都与平台无关。

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

注释：如果使用不可能为负的整数值而且确实需要额外的一位（bit），也可以把有符号整数值解释为无符号数，但是要非常仔细。例如，一个 byte 值 b 可以不表示 -128 ~ 127 的范围，如果你想表示 0 ~ 255 的范围，也可以存储在一个 byte 中。基于二进制算术运算的性质，只要不溢出，加法、减法和乘法都能正常计算。但对于其他运算，需要调用 Byte.toUnsignedInt(b) 来得到一个 0 ~ 255 的 int 值，然后处理这个整数值，再把它转换回 byte。Integer 和 Long 类都提供了处理无符号除法和求余数的方法。

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，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 $\pm 3.402\ 823\ 47 \times 10^{38}$ (6 ~ 7 位有效数字)
double	8 字节	大约 $\pm 1.797\ 693\ 134\ 862\ 315\ 70 \times 10^{308}$ (15 位有效数字)

`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（double-precision））。很多情况下，`float` 类型的精度（6 ~ 7 位有效数字）都不能满足需求。实际上，只有很少的情况适合使用 `float` 类型，例如，所使用的库需要单精度数，或者需要存储大量单精度数时。

`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）总是默认为 `double` 类型。可选地，也可以在 `double` 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字面量。例如， $0.125 = 2^{-3}$ 可以写为 0x1.0p-3。在十六进制表示法中，使用 p 表示指数，而不是 e。（e 是一个十六进制数位。）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。

所有的浮点数计算都遵循 IEEE 754 规范。具体来说，有 3 个特殊的浮点数值表示溢出和出错情况：

- 正无穷大
- 负无穷大
- NaN (不是一个数)

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` 类型常量）分别表示这三个特殊的值，但在实际中很少用到。特别要说明的是，不能如下检测一个特定结果是否等于 `Double.NaN`：

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

所有 NaN 的值都认为是不相同的。不过，可以使用 `Double.isNaN` 方法来判断：

```
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。例如，命令 `System.out.println(2.0-1.1)` 将打印出 0.8999999999999999，而不是我们期望的 0.9。这种舍入误差的主要原因是浮点数值采用二进制表示，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。如果需要精确的数值计算，不允许有舍入误差，则应该使用 `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 的字符常量。它与 "A" 不同，"A" 是包含一个字符的字符串。char 类型的值可以表示为十六进制值，其范围从 \u0000 ~ \uFFFF。例如，\u2122 表示商标符号 (™)，\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，请见表 3-3。可以在加引号的字符字面量或字符串中使用这些转义序列。例如，'\u2122' 或 "Hello\n"。转义序列 \u 还可以在加引号字符常量或字符串之外使用（而其他所有转义序列不可以）。例如：

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

就是完全合法的，\u005B 和 \u005D 分别是 [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	'	单引号	\u0027
\t	制表	\u0009	\\"	反斜线	\u005c
\n	换行	\u000a	\s	空格。在文本块中用来保留末尾空白符	\u0020
\r	回车	\u000d	\newline	只在文本块中使用：连接这一行和下一行	—
\f	换页	\u000c	—	—	—
\"	双引号	\u0022	—	—	—

◆ 警告：Unicode 转义序列会在解析代码之前处理。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。以下注释

```
// \u000a is a newline
```

会产生一个语法错误，因为读程序时 \u000a 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 后面并没有跟着 4 位十六进制数。

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，它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：一个是对一个特定的代码值，在不同的编码机制中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，而另外一些字符则需要两个或多个字节编码。

设计 Unicode 编码的目的就是要解决这些问题。在 20 世纪 80 年代开始启动统一工作时，人们认为两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扩展，当时所有人都这么想。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。设计 Java 时决定采用 16 位的 Unicode 字符集，这比使用 8 位字符集的其他程序设计语言有了很大的改进。

遗憾的是，经过一段时间后，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汉语、日语和韩语中的大量表意文字。现在，16 位的 `char` 类型已经不足以描述所有 Unicode 字符了。

下面利用一些专用术语来解释 Java 语言是如何解决这个问题的。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 个代码平面（code plane）。第一个代码平面称为基本多语言平面（basic multilingual plane），包括码点从 `U+0000` 到 `U+FFFF` 的“经典” Unicode 代码；其余的 16 个平面的码点从 `U+10000` 到 `U+10FFFF`，包含各种辅助字符（supplementary character）。

UTF-16 编码采用不同长度的代码表示所有 Unicode 码点。在基本多语言平面中，每个字符用 16 位表示，通常称为代码单元（code unit）；而辅助字符编码为一对连续的代码单元。采用这种编码对表示的每个值都属于基本多语言平面中未用的 2048 个值范围，通常称为替代区域（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，因为我们可以很快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 <https://tools.ietf.org/html/rfc2781>。）

在 Java 中，`char` 类型描述了采用 UTF-16 编码的一个代码单元。

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来处理（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++ 注释：在 C++ 中，数值甚至指针可以代替布尔值。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 程序员不会遇到以下麻烦：

```
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 变量与常量

与所有程序设计语言一样，Java 也使用变量来存储值。常量就是值不变的变量。在下面几节中，你会了解如何声明变量和常量。

3.4.1 声明变量

在 Java 中，每个变量都有一个类型（type）。声明一个变量时，先指定变量的类型，然后是变量名。这里给出一些声明变量的示例：

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

注意每个声明都以分号结束。由于声明是一个完整的 Java 语句，而所有 Java 语句都以分号结束，所以这里的分号是必须的。

作为变量名（以及其他名字）的标识符由字母、数字、货币符号以及“标点连接符”组成。第一个字符不能是数字。

'+' 和 '©' 之类的符号不能出现在变量名中，空格也不行。字母区分大小写：main 和 Main 是不同的标识符。标识符的长度基本上没有限制。

与大多数程序设计语言相比，Java 中“字母”“数字”和“货币符号”的范围更大。字母是指一种语言中表示字母的任何 Unicode 字符。例如，德国用户可以在变量名中使用字母 'ä'；讲希腊语的人可以使用 π。类似地，数字包括 '0'~'9' 和表示一位数字的任何 Unicode 字符。货币符号为 \$、€、¥ 等。标点连接符包括下画线字符 _、“波浪线” __ 以及其他一些符号。实际上大多数程序员都总是使用 A~Z、a~z、0~9 和下画线 _。

 提示：如果想要知道标识符中可以使用哪些 Unicode 字符，可以使用 Character 类的 isJavaIdentifierStart 和 isJavaIdentifierPart 方法来检查。

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

另外，不能使用 Java 关键字作为变量名。

在 Java 9 中，单下画线 _ 是一个保留字。将来的版本可能使用 _ 作为通配符。
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。分别声明每一个变量可以提高程序的可读性。

注释：如前所述，名字是区分大小写的，例如，`hireday` 和 `hireDay` 是两个不同的名字。一般来讲，两个名字不应该只是大小写有区别。不过，有些时候，可能确实很难给变量取一个好名字。于是，许多程序员就以类型名为变量命名，例如：

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

还有一些程序员更喜欢在变量名前加上前缀“a”：

```
Box aBox;
```

3.4.2 初始化变量

声明一个变量之后，必须用赋值语句显式地初始化变量，千万不要使用未初始化的变量的值。例如，Java 编译器会认为下面的语句序列有错误：

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

要想对一个已声明的变量进行赋值，需要将变量名放在等号 (=) 左侧，再把一个有适当值的 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，Java 中可以将声明放在代码中的任何地方。例如，以下代码在 Java 中都是合法的：

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

在 Java 中，变量的声明要尽可能靠近第一次使用这个变量的地方，这是一种很好的编程风格。

注释：从 Java 10 开始，对于局部变量，如果可以从变量的初始值推断出它的类型，就不再需要声明类型。只需要使用关键字 `var` 而无须指定类型：

```
var vacationDays = 12; // vacationDays is an int
var greeting = "Hello"; // greeting is a String
```

下一章会看到，这个特性可以让对象声明更为简洁。

C++ 注释：C 和 C++ 区分变量的声明和定义。例如：

```
int i = 10;
```

是一个定义，而

```
extern int i;
```

是一个声明。在 Java 中，并不区分变量的声明和定义。

3.4.3 常量

在 Java 中，可以用关键字 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。一旦赋值，就不能再更改了。习惯上，常量名使用全大写。

在 Java 中，可能经常需要创建一个常量以便在一个类的多个方法中使用，通常将这些常量称为类常量（class constant）。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一个例子：

```
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 方法之外。这样一来，同一个类的其他方法也可以使用这个常量。另外，如果一个常量被声明为 public（如这个例子中所示），那么其他类的方法也可以使用这个常量。对于这个例子，其他类可以通过 Constants2.CM_PER_INCH 使用这个类常量。

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 声明常量。

3.4.4 枚举类型

有时候，一个变量只包含有限的一组值。例如，销售的服装或比萨只有小、中、大和超大这四种尺寸。当然，可以将这些尺寸分别编码为整数 1、2、3、4 或字符 S、M、L、X。但这种设置很容易出错。很可能在变量中保存一个错误的值（如 0 或 m）。

针对这种情况，可以自定义枚举类型（enumerated type）。枚举类型包括有限个命名值。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所列的某个值，或者特殊值 null，null 表示这个变量没有设置任何值。

第 5 章将更详细地讨论枚举类型。

3.5 运算符

运算符用于连接值。在后面几节可以看到，Java 提供了一组丰富的算术和逻辑运算符以及数学函数。

3.5.1 算术运算符

在 Java 中，使用通常的算术运算符 +、-、*、/ 分别表示加、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，这表示浮点除法。整数的求余操作（有时称为取模（modulus））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 除将产生一个异常，而浮点数被 0 除将会得到一个无穷大或 NaN 结果。

注释：可移植性是 Java 程序设计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步骤的计算精度。

不过，结果可能与始终使用 64 位计算的结果不一样。为了提高可移植性，Java 虚拟机的最初规范规定所有的中间计算都必须完成截断。这种做法遭到了数值社区的反对。常用处理器上的截断操作会耗费时间，所以计算速度会减慢。因此，Java 程序设计语言认识到最优性能与理想的可再生性之间存在冲突，并做出了相应改进。允许虚拟机设计者对中间计算采用扩展精度。不过，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。

如今，处理器已经非常灵活，它们可以高效地完成 64 位运算。在 Java 17 中，再次要求虚拟机完成严格的 64 位运算，strictfp 关键字现在已经过时了。

3.5.2 数学函数与常量

Math 类中包含你可能会用到的各种数学函数，这取决于你要编写什么类型的程序。

要想计算一个数的平方根，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` 方法有一个微小的差异。`println` 方法处理 `System.out` 对象，而 `Math` 类中的 `sqrt` 方法并不处理任何对象，这样的方法被称为静态方法（static method）。有关静态方法的详细内容请参见第 4 章。

在 Java 中，没有完成幂运算的运算符，因此必须使用 `Math` 类的 `pow` 方法。以下语句：

```
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` 方法是为了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，这个表达式则为 1。当然，除非 `n` 是负奇数。如果是这样，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操作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规则（或称“欧几里得规则”）：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。这里要做一个调整，你想归一化为一个 0 ~ 11 之间的数。这很简单：`(position + adjustment) % 12`。不过，如果这个调整为负数会怎么样呢？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`。不管怎样都很麻烦。

`floorMod` 方法就让这个问题变得容易了：`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不常出现。）

`Math` 类提供了一些常用的三角函数：

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

还提供了指数函数以及它的反函数——自然对数和以 10 为底的对数：

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

最后，还提供了两个常量来表示 π 和 e 常量最接近的近似值：

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

 **提示：**不必在数学方法名和常量名前添加前缀“`Math`”，只要在源文件最上面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));
```

第 4 章中将讨论静态导入。

注释：在 Math 类中，为了达到最佳的性能，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，就应该使用 StrictMath 类。它实现了“可自由分发数学库（Freely Distributable Math Library，FDLIBM）”（www.netlib.org/fdlibm）的算法，确保在所有平台上得到相同的结果。

注释：Math 类提供了一些方法使整数运算更安全。如果一个计算溢出，数学运算符只是悄悄地返回错误的结果而不做任何提醒。例如，10 亿乘以 3 (`1000000000 * 3`) 的计算结果将是 -1 294 967 296，因为最大的 int 值也只是刚刚超过 20 亿。不过，如果调用 `Math.multiplyExact(1000000000, 3)`，就会生成一个异常。你可以捕获这个异常或者让程序终止，而不是允许它给出一个错误的结果然后悄无声息地继续运行。另外，还有一些方法（`addExact`、`subtractExact`、`incrementExact`、`decrementExact`、`negateExact` 和 `absExact`）也可以正确地处理 int 和 long 参数。

3.5.3 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。

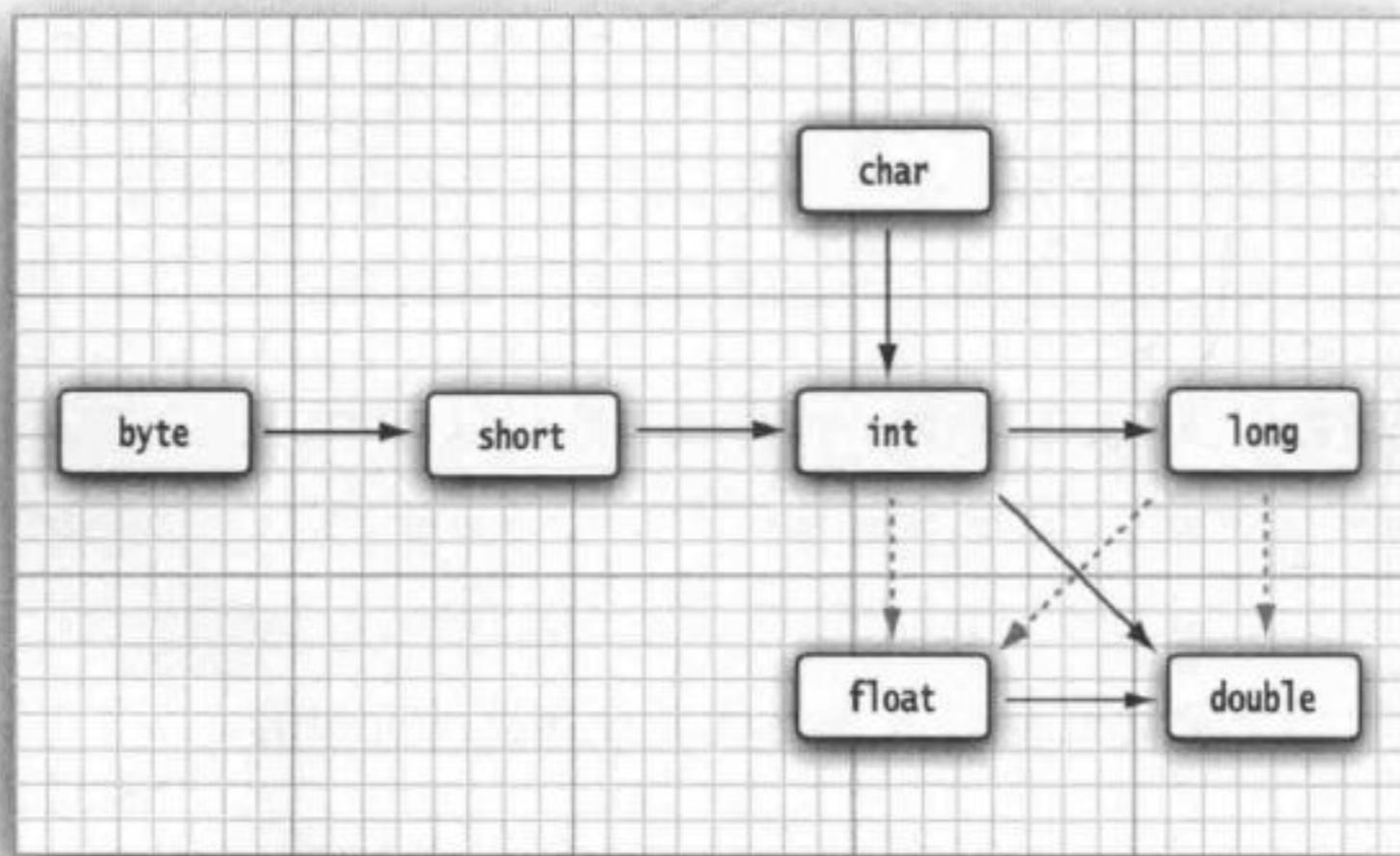


图 3-1 数值类型之间的合法转换

在图 3-1 中有 6 个实线箭头，表示无信息丢失的转换；另外有 3 个虚线箭头，表示可能有精度损失的转换。例如，123456789 是一个大整数，它包含的位数多于 float 类型所能表示的

位数。将这个整数转换为 float 类型时，数量级是正确的，但是会损失一些精度。

```
int n = 123456789;
float f = n; // f is 1.23456792E8
```

当用一个二元运算符连接两个值时（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。

3.5.4 强制类型转换

在上一小节中我们看到，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` 类型转换成 `int` 类型。在 Java 中，允许进行这种数值转换，不过当然可能会丢失一些信息。这种可能损失信息的转换要通过强制类型转换（cast）来完成。强制类型转换的语法格式是在圆括号中指定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，因为强制类型转换通过截断小数部分将浮点值转换为整型。

如果想舍入（round）一个浮点数来得到最接近的整数（大多数情况下，这种操作更有用），可以使用 `Math.round` 方法：

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

现在，变量 `nx` 的值为 10。调用 `round` 时，仍然需要使用强制类型转换（`int`）。原因是 `round` 方法的返回值是 `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一个 `long` 值赋给 `int` 类型的变量。

! **警告：**如果试图将一个数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` 实际上会得到 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，这样可以防止发生一些常见的错误。只有极少数的情况下需要将一个 `boolean` 值转换为一个数，此时可以使用条件表达式 `b? 1:0`。

3.5.5 赋值

可以在赋值中使用二元运算符，为此有一种很方便的简写形式。例如，

```
x += 4;
```

等价于：

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

(一般来说，要把运算符放在 = 号左边，如 *= 或 %=)。

◆ 警告：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)。

需要说明，在 Java 中，赋值是一个表达式 (expression)。也就是说，它有一个值，具体来讲就是所赋的那个值。可以使用这个值完成一些操作，例如，可以把它赋给另一个变量。考虑以下语句：

```
int x = 1;
int y = x += 4;
```

x += 4 的值是 5，因为这是赋给 x 的值。然后将这个值赋给 y。

很多程序员发现这种嵌套赋值很容易混淆，他们更喜欢分别清楚地写出这些赋值，如下所示：

```
int x = 1;
x += 4;
int y = x;
```

3.5.6 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++ 中的做法，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，所以不能对数值本身应用这些运算符。例如，4++ 就不是一个合法的语句。

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
```

很多程序员认为这种行为容易让人困惑。在 Java 中，很少在表达式中使用 ++。

3.5.7 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 `&&` 运算符结合两个表达式，

`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。可以利用这种行为来避免错误。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`，就不会出现除以 0 的错误。

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

3.5.8 条件运算符

Java 提供了 `conditional ?:` 运算符，可以根据一个布尔表达式选择一个值。如果条件 (`condition`) 为 `true`，表达式

`condition ? expression1 : expression2`

就计算为第一个表达式的值，否则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。

3.5.9 switch 表达式

需要在两个以上的值中做出选择时，可以使用 `switch` 表达式（这是 Java 14 中引入的）。如下所示：

```
String seasonName = switch (seasonCode)
{
    case 0 -> "Spring";
    case 1 -> "Summer";
    case 2 -> "Fall";
    case 3 -> "Winter";
    default -> "???";
};
```

case 标签还可以是字符串或枚举类型常量。

注释：与所有表达式类似，switch 表达式也有一个值。注意各个分支中箭头 \rightarrow 放在值前面。

注释：在 Java 14 中，switch 有 4 种形式。这一节重点讨论最有用的一种形式。参见 3.8.5 节，其中全面讨论了所有不同形式的 switch 表达式和语句。

可以为各个 case 提供多个标签，用逗号分隔：

```
int numLetters = switch (seasonName)
{
    case "Spring", "Summer", "Winter" -> 6;
    case "Fall" -> 4;
    default -> -1;
};
```

switch 表达式中使用枚举常量时，不需要为各个标签提供枚举名，这可以从 switch 值推导得出。例如：enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE }；

```
enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE };
...
Size itemSize = . . .;
String label = switch (itemSize)
{
    case SMALL -> "S"; // no need to use Size.SMALL
    case MEDIUM -> "M";
    case LARGE -> "L";
    case EXTRA_LARGE -> "XL";
};
```

在这个例子中，完全可以省略 default，因为每一个可能的值都有相应的一个 case。

警告：使用整数或 String 操作数的 switch 表达式必须有一个 default，因为不论操作数值是什么，这个表达式都必须生成一个值。

警告：如果操作数为 null，会抛出一个 NullPointerException。

3.5.10 位运算符

处理整型类型时，还有一些运算符可以直接处理组成整数的各个位。这意味着可以使用掩码技术得到一个数中的各个位。位运算符包括：

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

这些运算符按位模式操作。例如，如果 n 是一个整数变量，而且 n 的二进制表示中从右边数第 4 位为 1，则

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

会返回 1，否则返回 0。利用 & 并结合适当的 2 的幂，可以屏蔽其他位，而只留下其中的某一位。

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，计算结果之前，两个操作数都需要计算。

另外，还有 `>>` 和 `<<` 运算符可以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，`>>` 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完成模 64 运算）。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的任何一种做法。这意味着 C/C++ 中的 `>>` 运算符对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。

3.5.11 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，就按照这里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（但右结合运算符除外，如表中所示）。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
```

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（完成加法后 `b` 的值）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` 语句的第一部分和第三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
<code>! . ()</code> (方法调用)	从左向右
<code>! ~ ++ -- + (一元运算) - (一元运算) ()</code> (强制类型转换) <code>new</code>	从右向左
<code>* / %</code>	从左向右

(续)

运算符	结合性
+	从左向右
<< >> >>>	从左向右
<<= >>= instanceof	从左向右
== !=	从左向右
&	从左向右
^	从左向右
	从左向右
&&	从左向右
	从左向右
?:	从右向左
= += -= *= /= %= &= = ^= <<= >>= >>>=	从右向左

3.6 字符串

从概念上讲，Java 字符串就是 Unicode 字符序列。例如，字符串 "Java\u2122" 由 5 个 Unicode 字符 J、a、v、a 和 ™ 组成。Java 没有内置的字符串类型，而是标准 Java 类库中提供了一个预定义类，很自然地叫作 `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);
```

会创建一个由字符 "Hel" 组成的字符串。

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

`substring` 方法的第二个参数是你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。`substring` 会计数，这说明会从 0 开始，直到 3 为止，但不包含 3。

`substring` 的工作方式有一个优点：很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;
```

上述代码将字符串 "Expletive deleted" 赋给变量 message (注意，单词之间没有空格，+号完全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，后者会转换成字符串 (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;
```

将把 rating 设置为 "PG13"。

这个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，会打印出你希望的结果 (因为单词 is 后面加了一个空格，输出时也会有这个空格)。

如果需要把多个字符串放在一起，用一个界定符分隔，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"
```

在 Java 11 中，还提供了一个 repeat 方法：

```
String repeated = "Java".repeat(3); // repeated is "JavaJavaJava"
```

3.6.3 字符串不可变

String 类没有提供任何方法来修改字符串中的某个字符。如果希望将 greeting 的内容修改为 "Help!"，不能直接将 greeting 的最后两个位置的字符修改为 'p' 和 '!'。对于 C 程序员来说，这会让他们茫然无措。如何修改这个字符串呢？在 Java 中这很容易实现。可以提取想要保留的子串，再与希望替换的字符拼接：

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

上面这条语句将把 greeting 变量的当前值修改为 "Help!"。

由于不能修改 Java 字符串中的单个字符，所以在 Java 文档中将 String 类对象称为是不可变的 (immutable)，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。你不能修改这些值，不过，我们已经看到，可以修改字符串变量 greeting 的内容，让它指向另外一个字符串，这就如同可以让原本存放 3 的数值变量改成存放 4 一样。

这样做难道不会大大降低效率吗？看起来好像修改代码单元要比从头创建一个新字符串更简单。答案是：也对，也不对。的确，通过拼接 "Hel" 和 "p!" 来生成一个新字符串的效率确实不高。但是，不可变字符串有一个很大的优点：编译器可以让字符串共享。

为了弄清具体如何工作，可以想象各个字符串存放一个在公共存储池中。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，原始字符串和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远超过编辑字符串 (提取子串和拼

接字符串)所带来的低效率。可以看看你自己的程序,你会发现,大多数情况下都不会修改字符串,而只是需要对字符串进行比较。(有一种例外情况,将来自文件或键盘的单个字符或较短字符串组装成更大的字符串。为此,Java 提供了一个单独的类,在 3.6.9 节中将详细介绍)。

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

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

这样对比是错误的,实际上,Java 字符串大致类似于 `char*` 指针,

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

当把 `greeting` 替换为另一个字符串的时候,Java 代码大致完成以下操作:

```
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!"。即使最顽固的 C 程序员也得承认 Java 语法比一连串的 `strncpy` 调用要简洁得多。不过,如果再对 `greeting` 赋值会怎么样呢?

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

不会产生内存泄漏吗?毕竟,原始字符串在堆中分配。十分幸运,Java 会自动完成垃圾回收。如果一个内存块不再使用了,系统最终会将其回收。

如果你是 C++ 程序员,并使用 ANSI C++ 定义的 `string` 类,会感觉使用 Java 的 `String` 类型更舒服。C++ `string` 对象也会自动完成内存的分配与回收。要通过构造器、赋值操作符和析构器显式完成内存管理。不过,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`。需要注意的是,`s` 与 `t` 可以是字符串变量,也可以是字符串字面量。例如,以下表达式是合法的:

```
"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")
```

不要使用 `=` 运算符检测两个字符串是否相等!这个运算符只能确定两个字符串是否存放在同一个位置上。当然,如果字符串在同一个位置上,它们必然相等。但是,完全有可能将多个相等的字符串副本存放在不同的位置上。

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

如果虚拟机总是共享相等的字符串，则可以使用`==`运算符检测字符串是否相等。但实际上只有字符串字面量会共享，而`+`或`substring`等操作得到的字符串并不共享。因此，千万不要使用`==`运算符测试字符串的相等性，否则程序中会出现最糟糕的一种bug，这种bug可能会间歇性地随机出现。

C++注释：对于习惯使用C++`string`类的人来说，在完成相等性检测时一定要特别小心。C++的`string`类重载了`==`运算符，从而能检测字符串内容的相等性。可惜Java没有采用这种方式，尽管它的字符串“外观”看起来就像数值一样，但进行相等性测试时，则表现得类似于指针。Java语言的设计者也可以像对`+`那样进行特殊处理，为字符串重新定义`==`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，而是使用`strcmp`函数。Java的`compareTo`方法就类似于`strcmp`，因此，可以如下这样使用：

```
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`，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`，可以使用以下代码：

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

有时要检查一个字符串既不是`null`也不是空串，这种情况下可以使用：

```
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字符串是一个`char`值序列。从3.3.3节已经看到，`char`数据类型是采用UTF-16编码表示Unicode码点的一个代码单元。最常用的Unicode字符可以用一个代码单元表示，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 编码表示给定字符串所需要的代码单元个数。例如：

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

要想得到实际长度，即码点个数，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，可以使用以下语句：

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

为什么会对代码单元如此大惊小怪？请考虑下面这个句子：

Ø is the set of octonions.

使用 UTF-16 编码表示字符 Ø (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 Ø 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。

注释：不要以为可以忽略代码单元在 U+FFFF 以上的奇怪字符，喜欢 emoji 表情符号的用户可能会在字符串中加入类似 🍺 (U+1F37A, 啤酒杯) 的字符。

如果想要遍历一个字符串，并且依次查看每一个码点，可以使用以下语句：

```
int cp = sentence.codePointAt(i);
i += Character.charCount(cp);
```

可以使用以下语句实现反向遍历：

```
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 `int` 值的一个“流”，每个 `int` 值对应一个码点。(流在卷 II 中介绍。) 可以将流转换为一个数组(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，可以使用构造器(我们将在第 4 章详细讨论构造器和 `new` 操作符)。

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

要把单个码点转换为一个字符串，可以使用 `Character.toString(int)` 方法：

```
int codePoint = 0x1F37A;
str = Character.toString(codePoint);
```

注释：虚拟机不一定把字符串实现为代码单元序列。在 Java 9 中使用了一个更紧凑的表示。只包含单字节代码单元的字符串使用 byte 数组实现，所有其他字符串使用 char 数组。

3.6.7 String API

Java 中的 String 类包含近 100 个方法。下面的 API 注释汇总了最常用的一些方法。

本书中给出的 API 注释可以帮助你理解 Java 应用程序编程接口（API）。每一个 API 注释首先给出类名，如 `java.lang.String`。（`java.lang` 包名的重要性将在第 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。

API 注释不会列出一个特定类的所有方法，而是会以简洁的方式给出最常用的一些方法，完整的方法列表请参见联机文档（请参见 3.6.8 节）。

类名后面的编号是引入这个类的 JDK 版本号。如果某个方法是之后添加的，那么这个方法后面还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`

返回从给定位置开始的码点。

- `int offsetByCodePoints(int startIndex, int cpCount) 5`

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

- `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`

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

- `boolean isEmpty()`

`boolean isBlank() 11`

如果字符串为空或者由空白符组成，返回 `true`。

- `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 prefix 开头或以 suffix 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 或码点 cp 相等的第一个子串的开始位置。从索引 0 或 fromIndex 开始匹配。如果 str 或 cp 不在字符串中，则返回 -1。

- int lastIndexOf(String str)
- int lastIndexOf(String str, int fromIndex)
- int lastIndexOf(int cp)
- int lastIndexOf(int cp, int fromIndex)

返回与字符串 str 或码点 cp 相等的最后一个子串的开始位置。从字符串末尾或 fromIndex 开始匹配。如果 str 或 cp 不在字符串中，则返回 -1。

- int length()

返回字符串代码单元的个数。

- int codePointCount(int startIndex, int endIndex) 5

返回 startIndex 到 endIndex-1 之间的码点个数。

- String replace(CharSequence oldString, CharSequence newString)

返回一个新字符串，这是用 newString 替换原始字符串中与 oldString 匹配的所有子串得到的。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 到字符串末尾或 endIndex-1 的所有代码单元。

- String toLowerCase()

- String toUpperCase()

返回一个新字符串，这个字符串包含原始字符串中的所有字符，不过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小写字母改成大写母。

- String strip() 11

String stripLeading() 11

String stripTrailing() 11

返回一个新字符串，这个字符串要删除原始字符串头部和尾部或者只是头部或尾部的空白符。要使用这些方法，而不要使用古老的 trim 方法删除小于等于 U+0020 的字符。

- 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。

- `String repeat(int count)` 11

返回一个字符串，将当前字符串重复 `count` 次。

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，当看到一个 `CharSequence` 形参（parameter）时，完全可以传入 `String` 类型的实参（argument）。

3.6.8 阅读联机 API 文档

正如前面看到的，`String` 类包含许多方法。而且，标准库中有几千个类，方法数量更加惊人。要想记住所有有用的类和方法显然不太可能。因此，学会使用联机 API 文档十分重要，从中可以查找标准类库的所有类和方法。可以从 Oracle 下载 API 文档，并保存在本地。也可以在浏览器中访问 <https://docs.oracle.com/en/java/javase/17/docs/api>。

在 Java 9 中，API 文档有一个搜索框（见图 3-2）。较老的版本会有一些帧窗口包含包列表和类列表。仍然可以单击 `Frames` 菜单项得到这些列表。例如，要获得有关 `String` 类方法的更多信息，可以在搜索框中键入“`String`”，选择类型 `java.lang.String`，或者在帧窗口中找到类名的相应链接，并单击这个链接。你会看到这个类的描述，如图 3-3 所示。

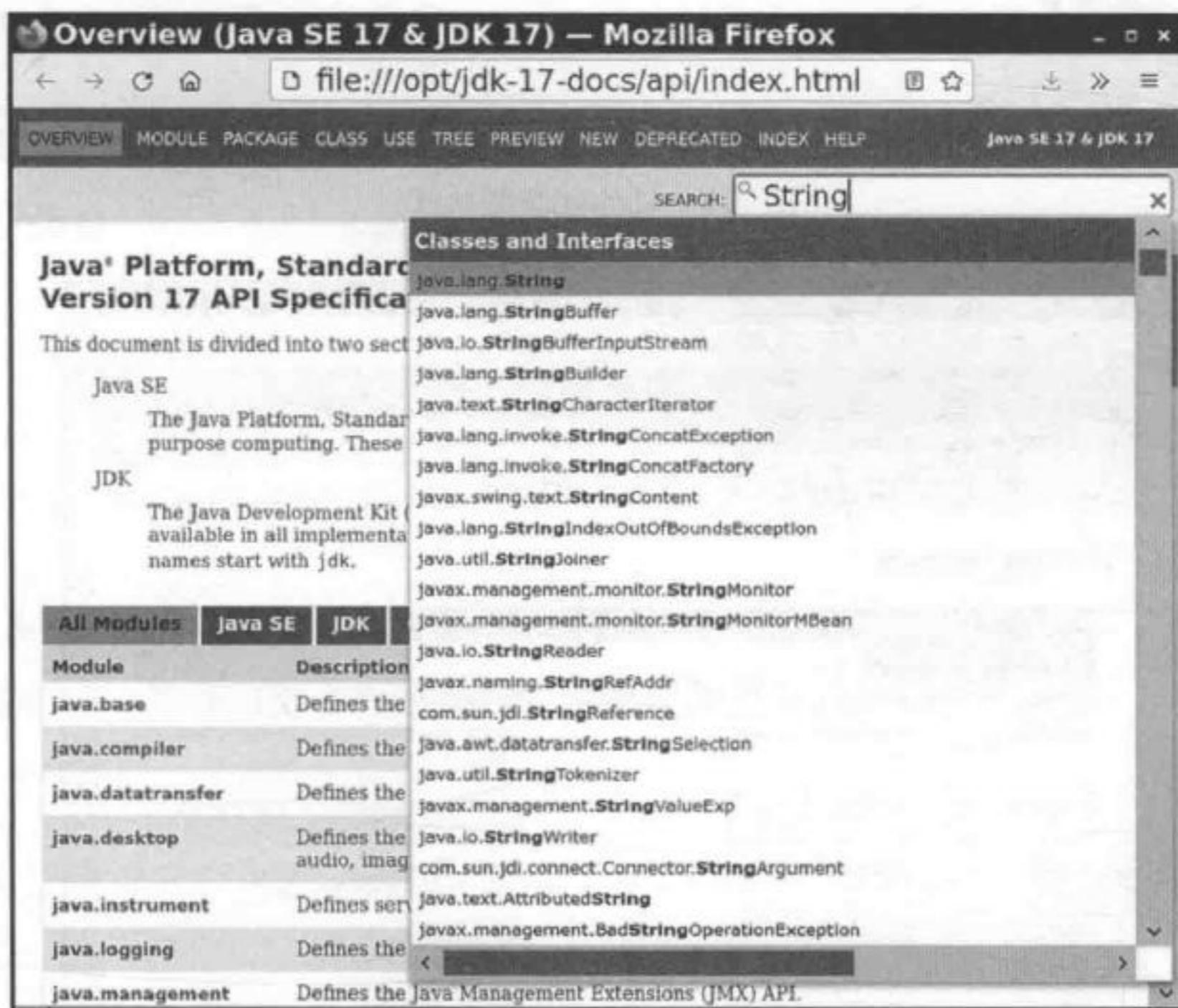


图 3-2 Java API 文档

接下来，向下滚动，直到看见按字母顺序排列的所有方法的小结（参见图 3-4）。单击任何一个方法名可以查看这个方法的详细描述（参见图 3-5）。例如，如果单击 `compareToIgnoreCase`

链接，就会看到 `compareToIgnoreCase` 方法的描述。

Module java.base
Package java.lang

Class String

java.lang.Object
java.lang.String

All Implemented Interfaces:
Serializable, CharSequence, Comparable<String>, Constable, ConstantDesc

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence, Constable, ConstantDesc
```

The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

图 3-3 String 类的描述

Modifier and Type	Method	Description
char	charAt(int index)	Returns the char value at the specified index.
IntStream	chars()	Returns a stream of int zero-extending the char values from this sequence.
int	codePointAt(int index)	Returns the character (Unicode code point) at the specified index.
int	codePointBefore(int index)	Returns the character (Unicode code point) before the specified index.
int	codePointCount(int beginIndex, int endIndex)	Returns the number of Unicode code points in the specified text range of this String.
IntStream	codePoints()	Returns a stream of code point values from this sequence.
int	compareTo(String anotherString)	Compares two strings lexicographically.

图 3-4 String 类方法的小结



图 3-5 一个 String 方法的详细描述

 提示：如果还没有下载 JDK 文档，现在就请按第 2 章中的介绍下载 JDK 文档。在浏览器中为这个文档的 index.html 页面建一个书签，就是现在，不要迟疑！

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，按键或文件中的单词。如果采用字符串拼接的方式来达到这个目的，效率会比较低。每次拼接字符串时，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。

如果需要用许多小字符串来构建一个字符串，可以采用以下步骤。首先，构建一个空的字符串构建器：

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

当每次需要添加另外一部分时，就调用 append 方法。

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

字符串构建完成时，调用 `toString` 方法。你会得到一个 String 对象，其中包含了构建器中的字符序列。

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

 注释：`StringBuffer` 类的效率不如 `StringBuilder` 类，不过它允许采用多线程的方式添加或删除字符。如果所有字符串编辑操作都在单个线程中执行（通常都是这样），则应当使用 `StringBuilder` 类。这两个类的 API 是一样的。

下面的 API 注释包含了 `StringBuilder` 类中最重要的方法。

API `java.lang.StringBuilder` 5

- `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个数。
- `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)`
追加一个码点，将它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)`
删除从 `startIndex` 到 `endIndex-1` 的代码单元并返回 `this`。
- `String toString()`
返回一个字符串，其数据与构建器或缓冲器内容相同。

3.6.10 文本块

利用 Java 15 新增的文本块 (text block) 特性，可以很容易地提供跨多行的字符串字面量。文本块以 `"""` 开头（这是开始 `"""`），后面是一个换行符，并以另一个 `"""` 结尾（这是结束 `"""`）：

```
String greeting = """
Hello
World
""";
```

文本块比相应的字符串字面量更易于读写：

```
"Hello\nWorld\n"
```

这个字符串包含两个 `\n`：一个在 `Hello` 后面，另一个在 `World` 后面。开始 `"""` 后面的换行符不作为字符串字面量的一部分。

如果不想要最后一行后面的换行符，可以让结束 `"""` 紧跟在最后一个字符后面：

```
String prompt = """
```

```
Hello, my name is Hal.  
Please enter your name: """;
```

文本块特别适合包含用其他语言编写的代码，如 SQL 或 HTML。可以直接将那些代码粘贴到一对三重引号之间：

```
String html = """  
<div class="Warning">  
    Beware of those who say "Hello" to the world  
</div>  
""";
```

需要说明的是，一般不用对引号转义。只有两种情况下需要对引号转义：

- 文本块以一个引号结尾。
- 文本块中包含三个或更多引号组成的一个序列。

遗憾的是，所有反斜线都需要转义。

常规字符串中的所有转义序列在文本块中也有同样的作用。

有一个转义序列只能在文本块中使用。行尾的 \ 会把这一行与下一行连接起来。例如：

```
"""  
Hello, my name is Hal. \  
Please enter your name: """;
```

等同于：

```
"Hello, my name is Hal. Please enter your name: "
```

文本块会对行结束符进行标准化，删除末尾的空白符，并把 Windows 的行结束符 (\r\n) 改为简单的换行符 (\n)。尽管不太可能，不过假如确实需要保留末尾的空格，这种情况下可以把最后一个空格转换为一个 \s 转义序列。

对于前导空白符则更为复杂。考虑一个从左边界缩进的典型的变量声明。文本块也可以缩进：

```
String html = """  
<div class="Warning">  
    Beware of those who say "Hello" to the world  
</div>  
""";
```

将去除文本块中所有行的公共缩进。实际字符串为：

```
"<div class="Warning">\n    Beware of those who say \"Hello\" to the world\n</div>\n"
```

注意，第一行和第三行没有缩进。

你的 IDE 很可能会使用制表符、空格或者制表符以及空格缩进所有文本块。

Java 很清晰，它没有规定制表符的宽度。空白符前缀必须与文本块中的所有行完全匹配。

去除缩进过程中不考虑空行。不过，结束 """" 前面的空白符很重要。一定要缩进到想要去除的空白符的末尾。

◆ 警告：要当心缩进文本块的公共前缀中混用制表符和空格的情况。不小心漏掉一个空格很容易得到一个缩进错误的字符串。

 提示：如果一个文本块中包含非 Java 代码，实际上最好沿左边界放置。这样可以与 Java 代码区分开，而且可以为长代码行留出更多空间。

3.7 输入与输出

为了增加后面示例程序的趣味性，我们希望程序能够接收输入，并适当地格式化程序输出。当然，现代的程序都使用 GUI 收集用户输入，然而，编写这样一个界面需要使用更多工具与技术，目前还不具备这些条件。我们的第一要务是熟悉 Java 程序设计语言，因此我们将使用基本的控制台来实现输入和输出。

3.7.1 读取输入

前面已经看到，将输出打印到“标准输出流”（即控制台窗口）是一件非常容易的事情，只需要调用 `System.out.println`。不过，读取“标准输入流”`System.in` 就没有那么简单了。要想读取控制台输入，首先需要构造一个与“标准输入流”`System.in` 关联的 `Scanner` 对象。

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

（构造器和 `new` 操作符将在第 4 章中详细介绍。）

现在，就可以使用 `Scanner` 类的各种方法读取输入了。例如，`nextLine` 方法将读取一行输入。

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

在这里，使用 `nextLine` 方法是因为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），可以调用

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

要想读取一个整数，要使用 `nextInt` 方法。

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

与此类似，`nextDouble` 方法可以读取下一个浮点数。

在程序清单 3-2 的程序中，首先询问用户姓名和年龄，然后打印一条如下的消息：

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

最后，在程序的最前面添加一行代码：

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

`Scanner` 类在 `java.util` 包中定义。当使用的类不是定义在基本 `java.lang` 包中时，需要使用 `import` 指令导入相应的包。有关包与 `import` 指令的详细描述请参见第 4 章。

程序清单 3-2 InputTest/InputTest.java

```
1 import java.util.*;
2
```

```
3 /**
4  * This program demonstrates console input.
5  * @version 1.10 2004-02-10
6  * @author Cay Horstmann
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：**因为输入对所有人都可见，所以 Scanner 类不适用于从控制台读取密码。可以使用 Console 类来达到这个目的。要想读取一个密码，可以使用以下代码：

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

为安全起见，将返回的密码存放在一个字符数组中，而不是字符串中。完成对密码的处理之后，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）。

使用 Console 对象处理输入不如使用 Scanner 方便。必须一次读取一行输入，而且 Console 类没有提供方法来读取单个单词或数字。

API `java.util.Scanner` 5

- `Scanner(InputStream in)`
用给定的输入流构造一个 Scanner 对象。
- `String nextLine()`
读取下一行输入。
- `String next()`
读取输入的下一个单词（以空白符作为分隔符）。
- `int nextInt()`
- `double nextDouble()`
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()`

检测下一个字符序列是否表示一个整数或浮点数。

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`。对于任何一个在控制台窗口启动的程序，都可使用 `Console` 对象。否则，是否可用取决于所使用的系统。

API `java.io.Console 6`

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

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

显示提示符（`prompt`）并读取用户输入，直到输入行结束。可选的 `args` 参数用来提供格式参数。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)` 语句将数值 `x` 输出到控制台。这个命令将以 `x` 的类型所允许的最大非 0 位数打印 `x`。例如：

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

会打印

3333.333333333335

如果希望显示美元、美分数，这就会有问题。

这个问题可以利用 `printf` 方法来解决，它沿用了 C 语言函数库中的古老约定。例如，以下调用

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

打印 `x` 时字段宽度（field width）为 8 个字符，精度为 2 个字符。也就是说，结果包含一个前导的空格和 7 个字符，如下所示：

3333.33

可以为 `printf` 提供多个参数，例如：

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

每一个以 % 字符开头的格式说明符（format specifiers）都替换为相应的参数。格式说明符末尾的转换字符（conversion character）指示要格式化的数值的类型：`f` 表示浮点数，`s` 表示

字符串，d 表示十进制整数。表 3-5 列出了用于 printf 的转换字符。

大写形式会生成大写字母。例如，"%8.2E" 将 3333.33 格式化为 3.33E+03，这里有一个大写的 E。

表 3-5 用于 printf 的转换字符

转换字符	类 型	示 例	转换字符	类 型	示 例
d	十进制整数	159	s 或 S	字符串	Hello
x 或 X	十六进制整数。要想对十六进制格式化有更多控制，可以使用 <code>HexFormat</code> 类		c 或 C	字符	H
o	八进制整数	237	b 或 B	布尔	true
f 或 F	定点浮点数	15.9	h 或 H	散列码	42628b2
e 或 E	指数浮点数	1.59e+01	tx 或 Tx	遗留的日期时间格式化。应当改为 使用 <code>java.time</code> 类，参见卷 II 第 6 章	—
g 或 G	通用浮点数 (e 和 f 中较短的一个) —		%	百分号	%
a 或 A	十六进制浮点数	0x1.fccdp3	n	与平台有关的行分隔符	—

注释：可以使用 s 转换字符格式化任意的对象。如果一个任意对象实现了 `Formattable` 接口，会调用这个对象的 `formatTo` 方法。否则，会调用 `toString` 方法将这个对象转换为一个字符串。`toString` 方法将在第 5 章讨论，第 6 章将介绍接口。

另外，还可以指定控制格式化输出外观的各种标志 (flag)。表 3-6 列出了用于 printf 的标志。例如，逗号标志会增加分组分隔符。即

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

会打印

3,333.33

可以使用多个标志，例如，"%, (.2f" 会使用分组分隔符，并将负数包围在括号内。

表 3-6 用于 printf 的标志

标 志	作 用	示 例
+	打印正数和负数的符号	+3333.33
空格	在正数前面增加一个空格	3333.33
0	增加前导 0	003333.33
-	字段左对齐	3333.33
(将负数包围在括号内	(3333.33)
,	增加分组分隔符	3,333.33
# (对于 f 格式)	总是包含一个小数点	3,333.
# (对于 x 或 o 格式)	添加前缀 0x 或 0	0xcafe
\$	指定要格式化的参数索引。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
<	格式化前面指定的同一个值。例如，%d%<x 将以十进制和十六进制打印同一个数	159 9F

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 + 1);
```

注释：在 Java 15 中，可以使用 `formatted` 方法，这样可以少敲 5 个字符：

```
String message = "Hello, %s. Next year, you'll be %d".formatted(name, age + 1);
```

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

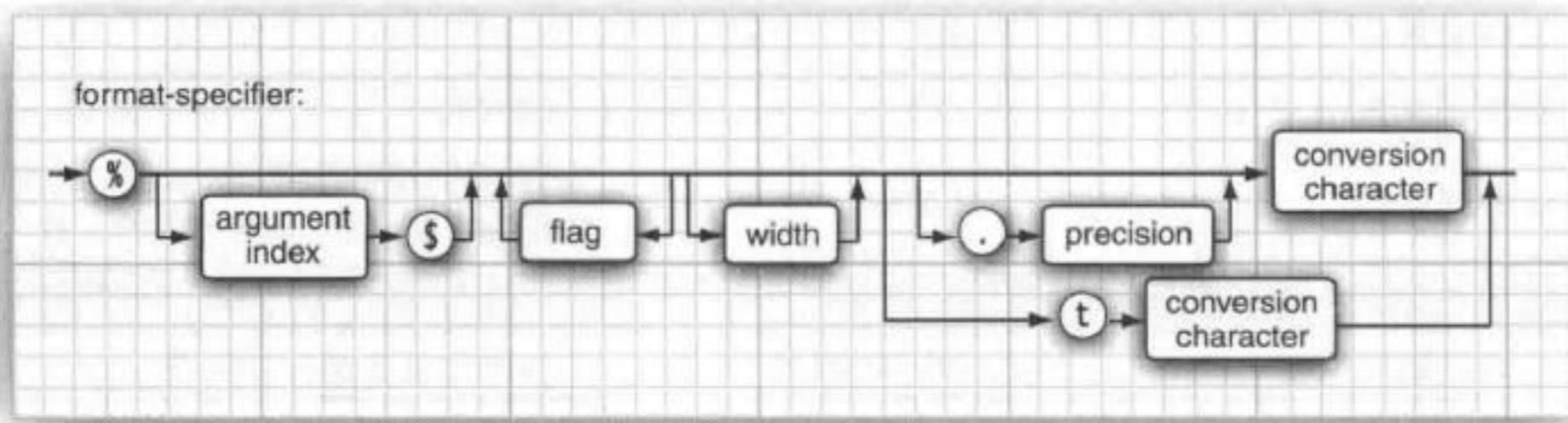


图 3-6 格式说明符语法

注释：格式化规则是特定于本地化环境的。例如，在德国，分组分隔符是点号而不是逗号。在卷 II 第 7 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

要想读取一个文件，需要构造一个 `Scanner` 对象，如下所示：

```
Scanner in = new Scanner(Path.of("myfile.txt"), StandardCharsets.UTF_8);
```

如果文件名中包含反斜线符号，记住要在每个反斜线之前再加一个额外的反斜线转义：`"c:\\mydirectory\\myfile.txt"`。

现在就可以使用之前见过的任何 `Scanner` 方法读取这个文件了。

注释：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。

警告：可以提供一个字符串参数来构造一个 `Scanner`，但这个 `Scanner` 会把字符串解释为数据，而不是文件名。例如，如果调用：

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

这个 `scanner` 会将参数看作是包含 10 个字符（'m'、'y'、'f' 等）的数据。这可能不是我们的原意。

要想写入文件，需要构造一个 `PrintWriter` 对象。在构造器（constructor）中，需要提供文件名和字符编码：

```
PrintWriter out = new PrintWriter("myfile.txt", StandardCharsets.UTF_8);
```

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

注释：当指定一个相对文件名时，例如，“`myfile.txt`”、“`mydirectory/myfile.txt`”或“`../myfile.txt`”，文件将相对于启动 Java 虚拟机的那个目录放置。如果从一个命令 shell 执行以下命令启动程序：

```
java MyProg
```

启动目录就是命令 shell 的当前目录。不过，如果使用集成开发环境，那么启动目录将由 IDE 控制。可以使用下面的调用找到这个目录的位置：

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

如果觉得文件定位太麻烦，可以考虑使用绝对路径名，例如：“`c:\\mydirectory\\myfile.txt`”或者“`/home/me/mydirectory/myfile.txt`”。

如你所见，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`，或者用一个无法创建的文件名构造一个 `PrintWriter`，就会产生异常。Java 编译器认为这些异常比“被零除”异常更严重。在第 7 章中，你会学习各种处理异常的方法。至于现在，只需要告诉编译器：你已经知道有可能出现“输入 / 输出”异常。为此，要用一个 `throws` 子句标记 `main` 方法，如下所示：

```
public static void main(String[] args) throws IOException
{
    Scanner in = new Scanner(Path.of("myfile.txt"), StandardCharsets.UTF_8);
    ...
}
```

现在你已经学习了如何读写包含文本数据的文件。对于更高级的内容，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 zip 压缩文件，请参见卷 II 第 2 章。

注释：从命令 shell 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

- `Scanner(Path p, String encoding)`

构造一个 `Scanner` 使用给定字符编码从给定路径读取数据。

- `Scanner(String data)`

构造一个 `Scanner` 从给定字符串读取数据。

API `java.io.PrintWriter 1.1`

- `PrintWriter(String fileName)`

构造一个 `PrintWriter` 将数据写入指定文件。

API `java.nio.file.Path`

- `static Path of(String pathname) 11`

由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 支持使用条件语句和循环结构来确定控制流程。这里首先讨论条件语句，然后介绍循环语句，最后介绍 `switch` 语句，它可以用来检测一个表达式的多个值。

C++ 注释：Java 的控制流程结构与 C 和 C++ 的控制流程结构基本相同，只有很少几个例外。Java 中没有 `goto` 语句，但 `break` 语句可以带标签，可以利用它从嵌套循环中跳出（对于这种情况，C 语言中可能就要使用 `goto` 语句了）。最后，还有一种变形的 `for` 循环，有点类似于 C++ 中基于范围的 `for` 循环和 C# 中的 `foreach` 循环。

3.8.1 块作用域

在学习控制结构之前，需要了解块（block）的概念。

块（即复合语句）由若干条 Java 语句组成，并用一对大括号括起来。块确定了变量的作用域。一个块可以嵌套在另一个块中。下面就是嵌套在 `main` 方法块中的一个块。

```
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 redeclare 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遮蔽 (shadow) 在外层定义的变量。这就有可能带来编程错误，因此 Java 中不允许这样做。

3.8.2 条件语句

在 Java 中，条件语句的形式为

```
if (condition) statement
```

这里的条件必须用小括号括起来。

与大多数程序设计语言一样，在 Java 中，常常希望在某个条件为真时执行多条语句。在这种情况下，就可以使用块语句 (block statement)，形式如下：

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

例如：

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

当 yourSales 大于或等于 target 时，将执行大括号中的所有语句（请参见图 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，更一般的条件语句如下所示（请参见图 3-8）：

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

例如：

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

其中 else 部分总是可选的。else 子句与最邻近的 if 构成一组。因此，在以下语句中：

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

else 与第 2 个 if 配对。当然，使用大括号可以让这段代码更加清晰：

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



图 3-7 if 语句的流程图

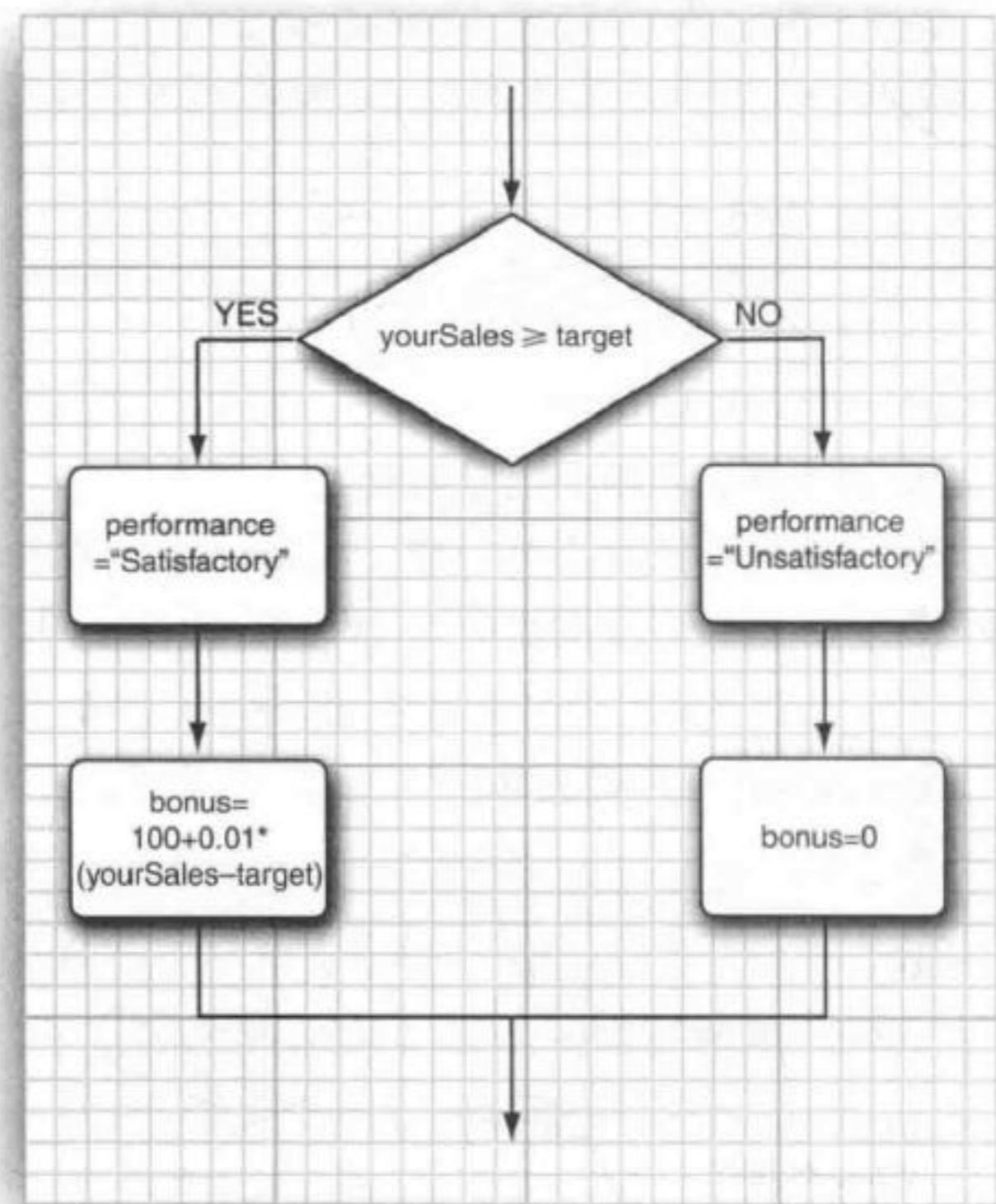


图 3-8 if/else 语句的流程图

反复使用 if...else if... 很常见（如图 3-9 所示）。例如：

```

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

while 循环会在条件为 true 时执行一个语句（也可以是一个块语句）。一般形式如下：

`while (condition) statement`

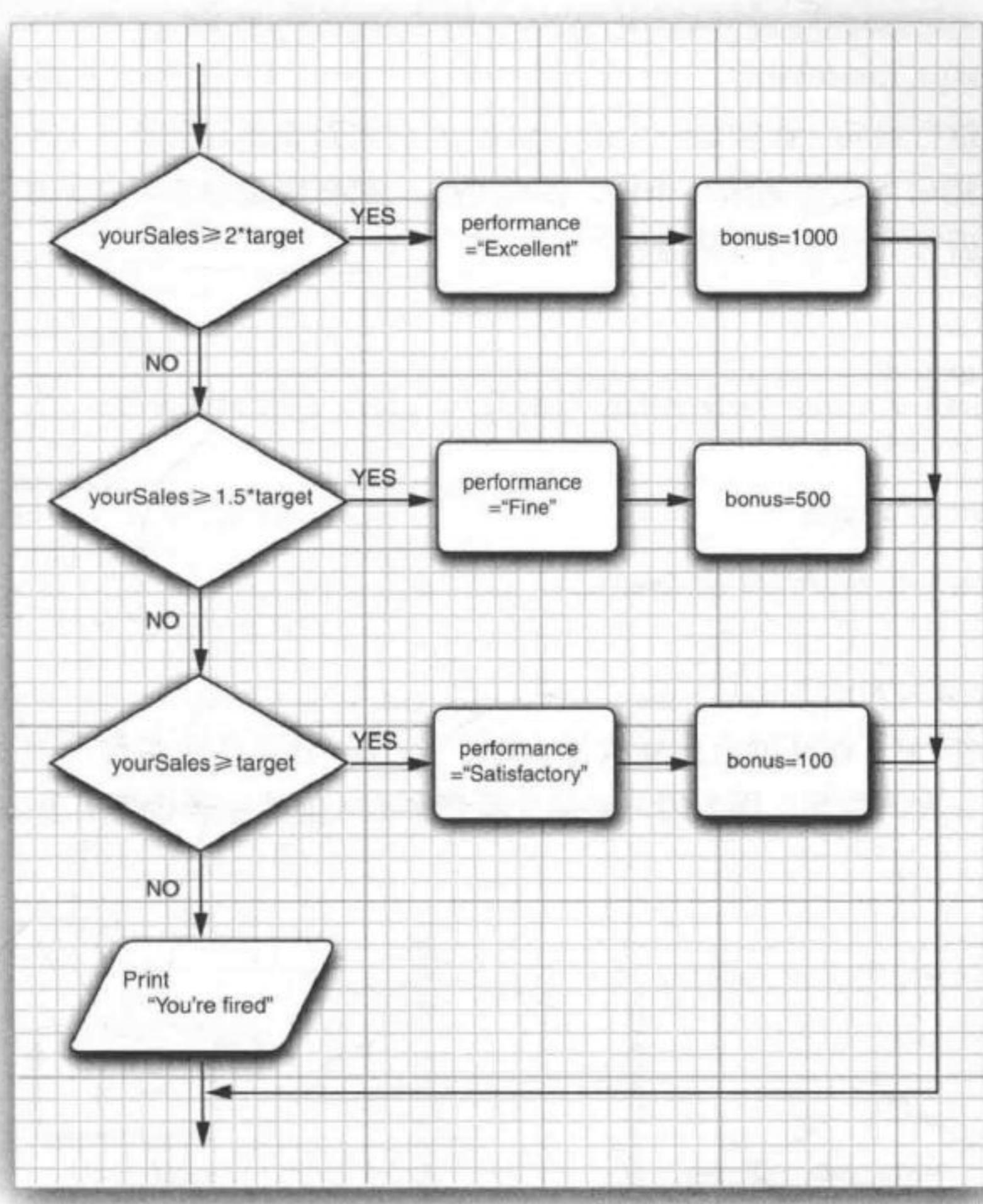


图 3-9 if/else if (多分支) 的流程图

如果开始时循环条件就为 `false`，那么 `while` 循环一次也不执行（请参见图 3-10）。

程序清单 3-3 中的程序将计算需要多长时间才能够存下一定数量的退休金，假定每年存入相同金额，而且利率是固定的。

在这个示例中，我们会让一个计数器递增，并在循环体中更新当前的累积金额，直到总金额超过目标金额为止。

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

（千万不要依赖这个程序安排退休计划。这里没有考虑通货膨胀和你期望的生活水准。）

`while` 循环语句在最前面检测循环条件。因此，循环体中的代码有可能一次都不执行。如

如果希望循环体至少执行一次，需要使用 do/while 循环将检测放在最后。它的语法如下：

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

这种循环先执行语句（通常是一个语句块），然后再检查循环条件。如果条件为 true，就重复执行语句，然后再次检测循环条件，依此类推。在程序清单 3-4 中，首先计算退休账户中新的余额，然后再询问是否打算退休：

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

只要用户回答 "N"，循环就会重复执行（见图 3-11）。这是一个很好的例子，它展示了一个至少需要执行一次的循环，因为用户必须先看到余额才能决定是否满足退休所用。

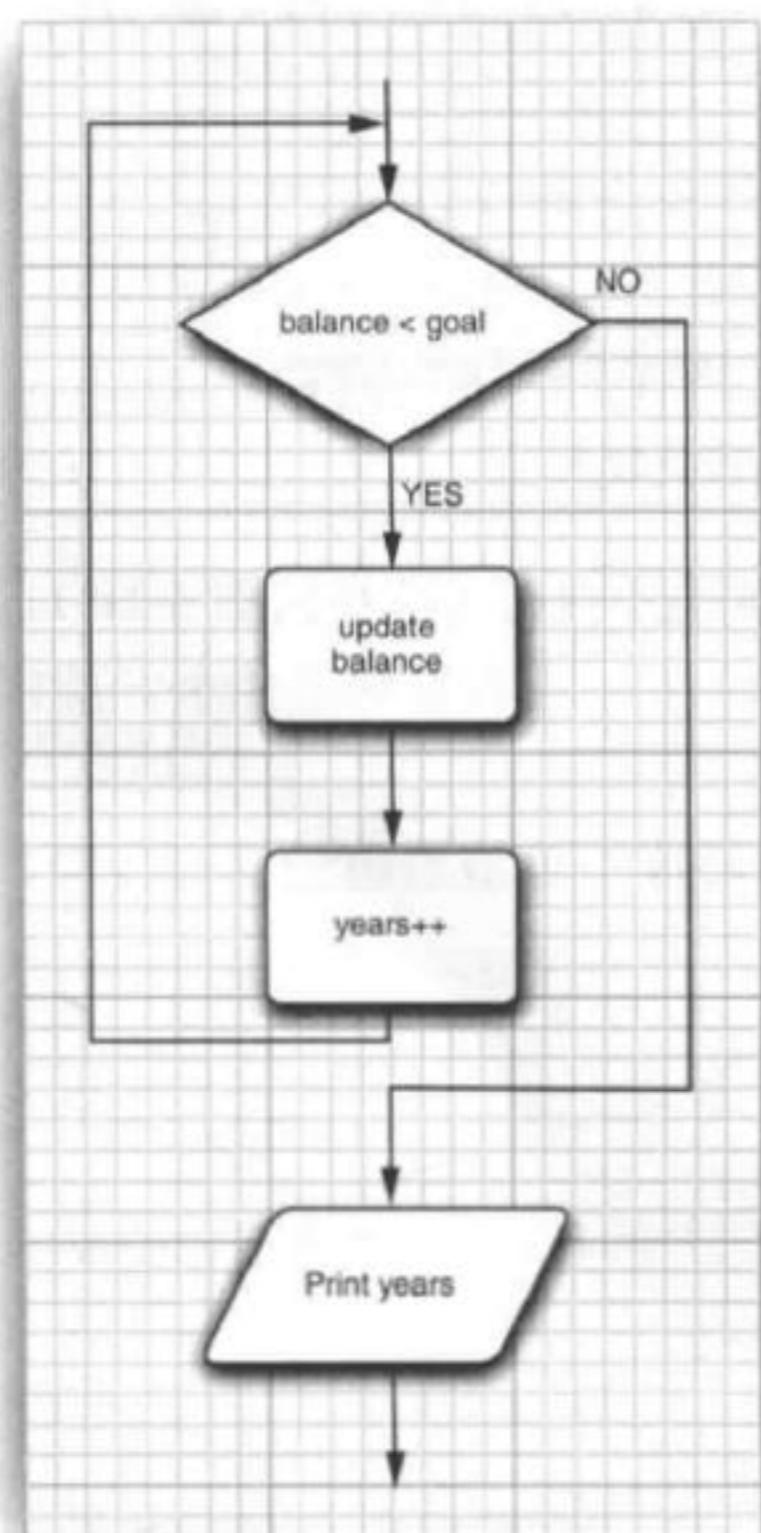


图 3-10 while 语句的流程图

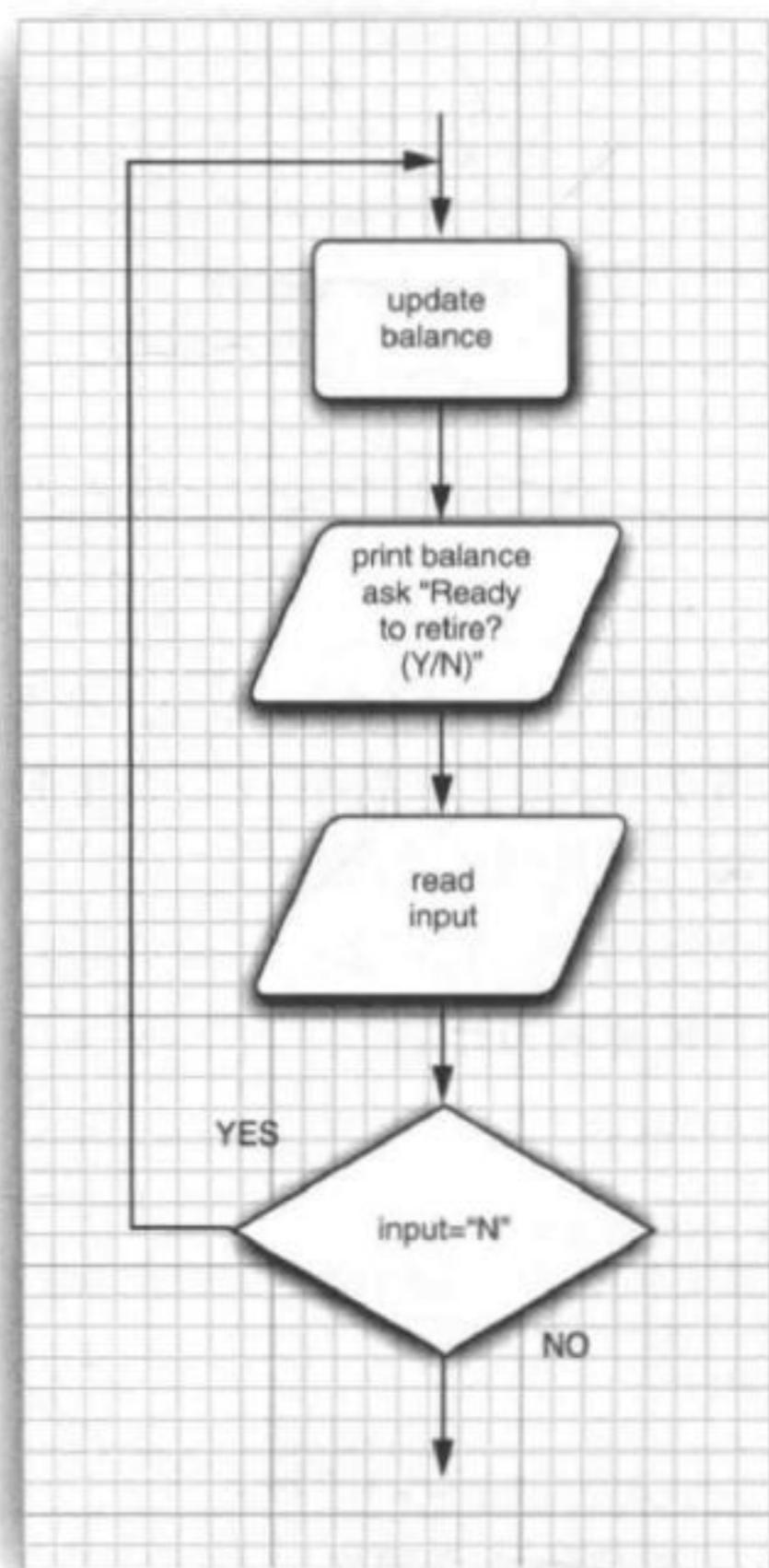


图 3-11 do/while 语句的流程图

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 }
```

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 确定性循环

`for` 循环语句是支持迭代的一种通用结构，它由一个计数器或类似的变量控制迭代次数，每次迭代后这个变量将会更新。如图 3-12 所示，下面的循环将在屏幕上显示出打印数字 1 ~ 10：

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

`for` 语句的第 1 部分通常是对计数器初始化；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 部分指定如何更新计数器。

与 C++ 类似，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，所写的循环很可能晦涩难懂。

即使受这个规则所限，仍有无尽可能，你可以写各种各样的 `for` 循环。例如，可以编写

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。由于存在舍入误差，可能永远达不到精确的最终值。在这个例子中，因为 0.1 无法精确地用二进制表示，所以，x 将从 9.9999999999998 跳到 10.0999999999998。

在 for 语句的第一部分中声明一个变量之后，这个变量的作用域会扩展到这个 for 循环体的末尾。

```
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，就要确保在循环体之外声明这个变量。

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

另一方面，可以在不同的 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;
```

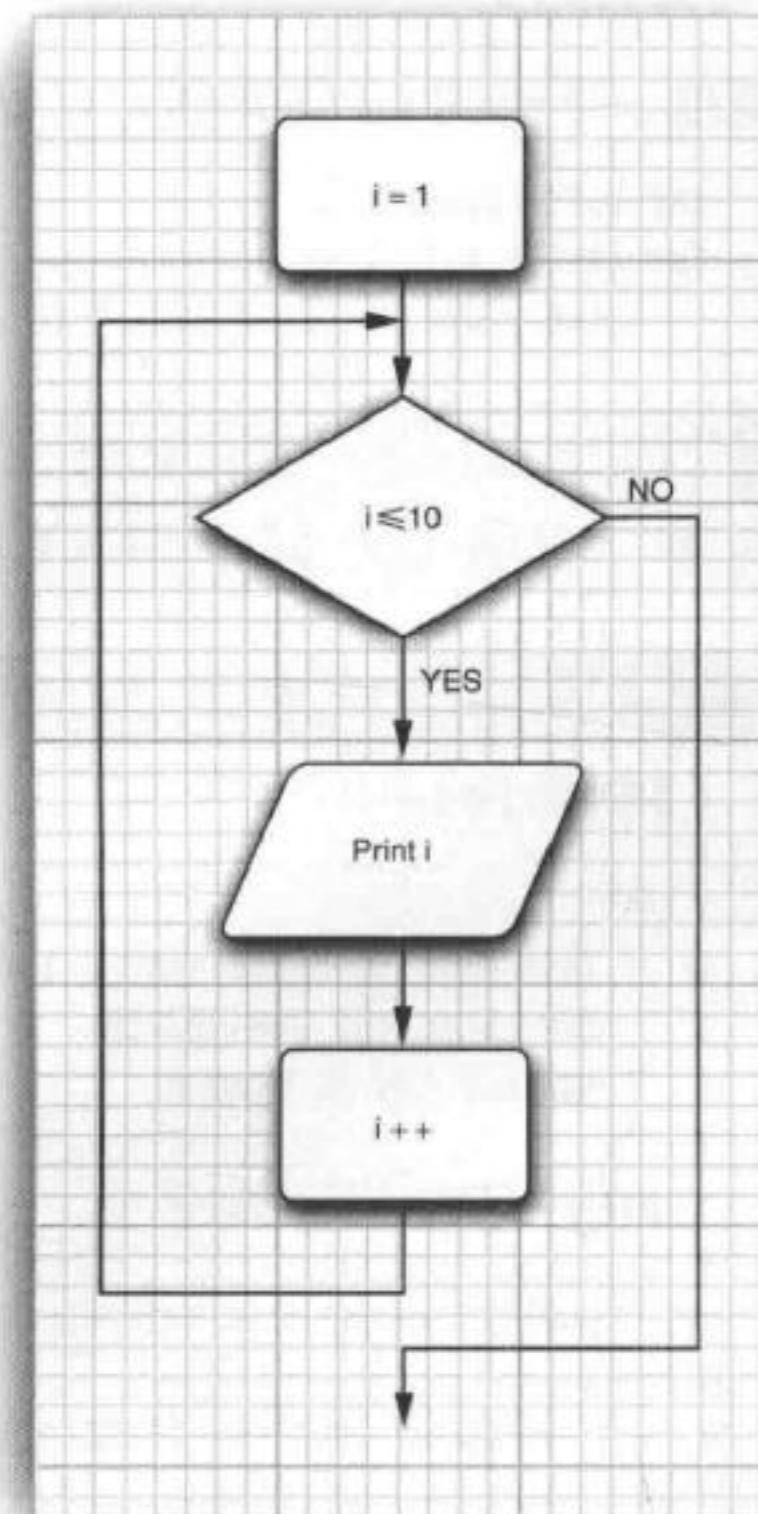


图 3-12 for 语句的流程图

```

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

```

程序清单 3-5 给出了一个 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 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，所以中奖的概率是 1/15 890 700。祝你好运！

一般情况下，如果从 n 个数字中抽取 k 个数字，就会有

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

种可能。下面的 for 循环语句可以计算这个值：

```

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

```

 **注释：**3.10.3 节将会介绍“泛型 for 循环”（又称为 for each 循环），利用这个循环可以很方便地访问一个数组或集合中的所有元素。

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 }
```

3.8.5 多重选择：switch 语句

在处理同一个表达式的多个选项时，使用 if/else 结构会显得有些笨拙。switch 语句会让这个工作变得容易，特别是采用 Java 14 引入的形式时会更简单。

例如，如果建立一个如图 3-13 所示的菜单系统，其中包含 4 个选项，可以使用以下代码：

```
Scanner in = new Scanner(System.in);
System.out.print("Select an option (1, 2, 3, 4) ");
int choice = in.nextInt();
switch (choice)
{
    case 1 ->
        . . .
    case 2 ->
        . . .
    case 3 ->
        . . .
    case 4 ->
        . . .
    default ->
        System.out.println("Bad input");
}
```

case 标签可以是：

- 类型为 char、byte、short 或 int 的常量表达式
- 枚举常量
- 字符串字面量
- 多个字符串，用逗号分隔

例如：

```
String input = . . .;
switch (input.toLowerCase())
{
    case "yes", "y" ->
        . . .
    case "no", "n" ->
        . . .
    default ->
        . . .
}
```

switch 语句的“经典”形式可以追溯到 C 语言，从 Java 1.0 开始就支持这种形式。具体形式如下：

```
int choice = . . .;
switch (choice)
{
    case 1:
        . . .
        break;
```

```
case 2:  
    ...  
    break;  
case 3:  
    ...  
    break;  
case 4:  
    ...  
    break;  
default:  
    // bad input  
    ...  
    break;  
}
```

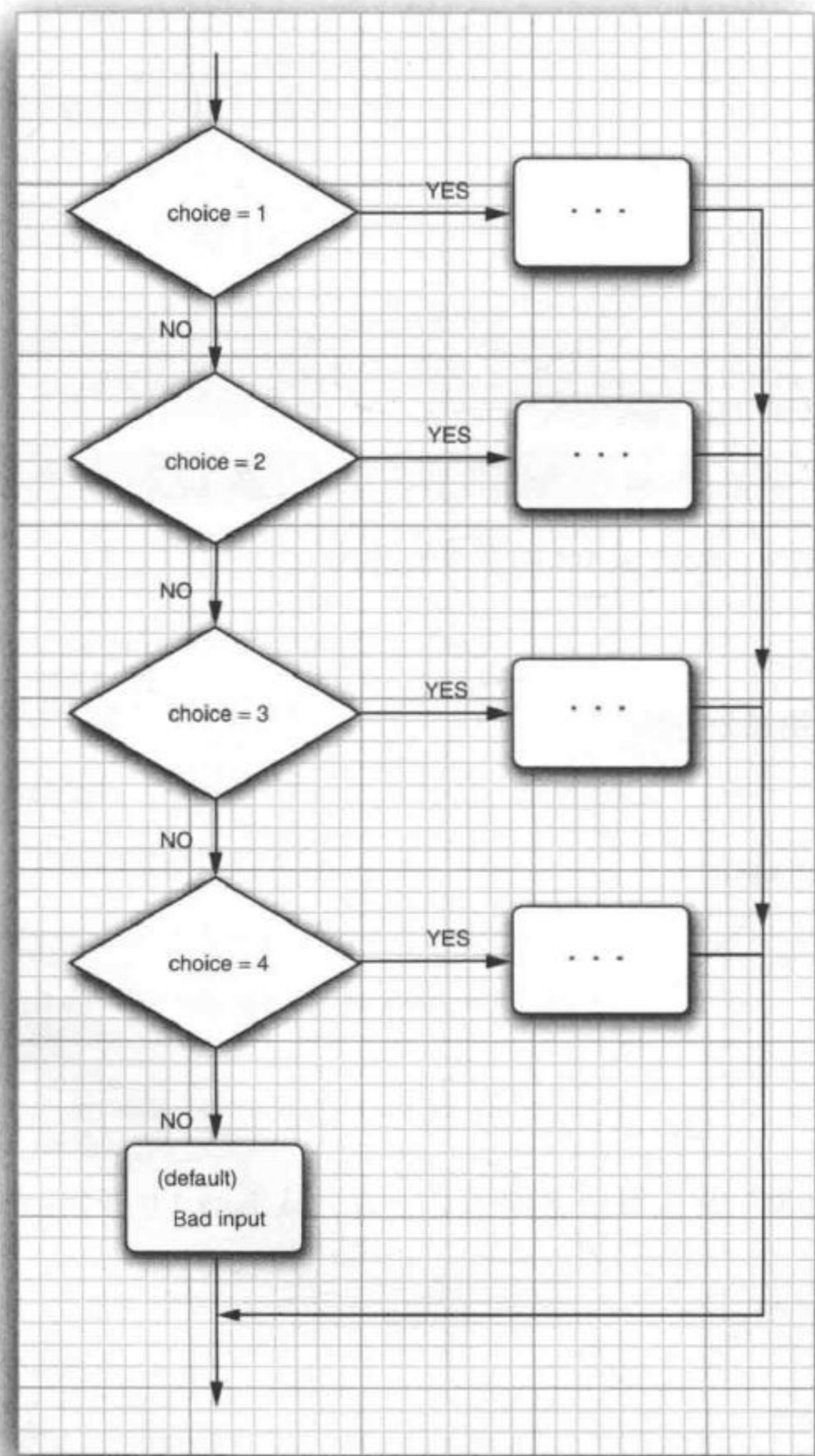


图 3-13 switch 语句的流程图

`switch` 语句从与选项值相匹配的 `case` 标签开始执行，直到遇到下一个 `break` 语句，或者执行到 `switch` 语句结束。如果没有匹配的 `case` 标签，则执行 `default` 子句（如果有 `default` 子句）。

◆ 警告：有可能触发多个分支。如果忘记在一个分支末尾增加 `break` 语句，就会接着执行下一个分支！这种情况相当危险，常常会引发错误。

为了检测这种问题，编译代码时可以加上 `-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。

如果你确实是想使用这种“直通式”（`fallthrough`）行为，可以为其外围方法加一个注解`@SuppressWarnings("fallthrough")`。这样就不会对这个方法生成警告了。（注解是为编译器或处理 Java 源文件或类文件的工具提供信息的一种机制。卷 II 会深入介绍注解。）

这两种 `switch` 形式都是语句。在 3.5.9 节中，我们已经见过一个 `switch` 表达式，它会生成一个值。`switch` 表达式没有“直通式”行为。

为了对称，Java 14 还引入了一个有直通行为的 `switch` 表达式，所以总共有 4 种不同形式的 `switch`。表 3-7 给出了这 4 种形式。

在有直通行为的形式中，每个 `case` 以一个冒号结束。如果 `case` 以箭头 `->` 结束，则没有直通行为。不能在一个 `switch` 语句中混合使用冒号和箭头。

注意 `switch` 表达式中的 `yield` 关键字。与 `break` 类似，`yield` 会终止执行。但与 `break` 不同的是，`yield` 还会生成一个值，这就是表达式的值。

要在 `switch` 表达式的一个分支中使用语句而不想有直通行为，就必须使用大括号和 `yield`，如表中示例所示，这个例子将为一个分支增加日志语句：

```
case "Spring" ->
{
    System.out.println("spring time!");
    yield 6;
}
```

`switch` 表达式的每个分支必须生成一个值。最常见的做法是，各个值跟在一个箭头 `->` 后面：

```
case "Summer", "Winter" -> 6;
```

如果无法做到，则使用 `yield` 语句。

表 3-7 4 种 `switch` 形式

	表达式	语句
无直通行为	<pre>int numLetters = switch (seasonName) { case "Spring" -> { System.out.println("spring time!"); yield 6; } case "Summer", "Winter" -> 6; case "Fall" -> 4; default -> -1; };</pre>	<pre>switch (seasonName) { case "Spring" -> { System.out.println("spring time!"); numLetters = 6; } case "Summer", "Winter" -> numLetters = 6; case "Fall" -> numLetters = 4; default -> numLetters = -1; }</pre>

(续)

	表达式	语句
有直通行为	<pre>int numLetters = switch (seasonName) { case "Spring": System.out.println("spring time!"); case "Summer", "Winter": yield 6; case "Fall": yield 4; default: yield -1; };</pre>	<pre>switch (seasonName) { case "Spring": System.out.println("spring time!"); case "Summer", "Winter": numLetters = 6; break; case "Fall": numLetters = 4; break; default: numLetters = -1; }</pre>

 **注释：**完全可以在 switch 表达式的一个分支中抛出异常。例如：

```
default -> throw new IllegalArgumentException("Not a valid season");
```

异常将在第 7 章详细介绍。

 **警告：**switch 表达式的关键是生成一个值（或者产生一个异常而失败）。不允许“跳出”switch 表达式：

```
default -> { return -1; } // ERROR
```

具体来讲，不能在 switch 表达式中使用 return、break 或 continue 语句。（后面两个语句参见 3.8.6 节的介绍）。

switch 有这么多种形式，要如何选择呢？

switch 表达式优于语句。如果每个分支会为一个变量赋值或方法调用计算值，则用一个表达式生成值，然后使用这个值。例如：

```
numLetters = switch (seasonName)
{
    case "Spring", "Summer", "Winter" -> 6
    case "Fall" -> 4
    default -> -1
};
```

要优于

```
switch (seasonName)
{
    case "Spring", "Summer", "Winter" ->
        numLetters = 6;
    case "Fall" ->
        numLetters = 4;
    default ->
        numLetters = -1;
}
```

只有在确实需要直通行为时，或者必须为一个 switch 表达式增加语句时，才需要使用

break 或 yield。不过这些情况非常少见。

3.8.6 中断控制流程的语句

尽管 Java 的设计者将 goto 仍作为一个保留字，但实际上并不打算在语言中包含 goto。通常，使用 goto 语句会被认为是一种拙劣的程序设计风格。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 就曾写过一篇名为 *Structured Programming with goto statements* 的著名文章）。他们认为，无限制地使用 goto 语句确实很容易导致错误，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 语言中增加了一条新的语句：带标签的 break，以此来支持这种程序设计风格。

下面首先来看不带标签的 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。例如，

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

循环开始时，如果 years > 100，或者如果循环中间 balance ≥ goal，则退出循环。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，允许跳出多重嵌套的循环。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。此时你可能希望完全跳出所有嵌套循环。如果只是为各层循环检测添加一些额外的条件，这会很不方便。

下面的例子展示了 break 语句如何工作。请注意，标签必须放在你想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，执行带标签的 `break` 会跳转到带标签的语句块末尾。与任何使用 `break` 语句的代码一样，接下来需要检测循环是正常退出，还是由于 `break` 提前退出。

注释：有意思的是，可以将标签应用到任何语句，甚至可以应用到 `if` 语句或者块语句，如下所示：

```

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

```

因此，如果确实希望使用 `goto` 语句，而且一个代码块恰好在你想要跳转到的位置之前结束，就可以使用 `break` 语句！当然，我并不提倡使用这种方法。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外围循环的首部。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` 语句会越过当前循环体的剩余部分，直接跳到循环首部。

如果在 `for` 循环中使用 `continue` 语句，会跳转到 `for` 循环的“更新”部分。例如：

```
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` 语句将跳转到 `count++` 语句。

还有一种带标签的 `continue` 语句, 将跳转到有匹配标签的循环的首部。

 提示: 许多程序员发现 `break` 和 `continue` 语句很容易混淆。这些语句完全是可选的, 即不使用这些语句也能表达同样的逻辑。在本书中, 所有程序都不会使用 `break` 和 `continue`。

3.9 大数

如果基本的整数和浮点数精度不足以满足需求, 那么可以使用 `java.math` 包中两个很有用的类: `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` 类实现任意精度的整数运算, `BigDecimal` 实现任意精度的浮点数运算。

使用静态的 `valueOf` 方法可以将一个普通的数转换为大数:

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

对于更长的数, 可以使用一个带字符串参数的构造器:

```
BigInteger reallyBig  
= new BigInteger("222232244629420445529739893461909967206666939096499764990979600");
```

另外还有一些常量: `BigInteger.ZERO`、`BigInteger.ONE` 和 `BigInteger.TEN`, Java 9 之后还增加了 `BigInteger.TWO`。

 警告: 对于 `BigDecimal` 类, 总是应当使用带一个字符串参数的构造器。还有一个 `BigDecimal(double)` 构造器, 不过这个构造器本质上很容易产生舍入误差, 例如, `new BigDecimal(0.1)` 会得到以下数位:

```
0.100000000000000055511151231257827021181583404541015625
```

遗憾的是, 不能使用人们熟悉的算术运算符 (如: + 和 *) 来组合大数, 而需要使用大数类中的 `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 不能通过编程实现运算符重载。使用 `BigInteger` 类的程序员无法重定义 + 和 * 运算符来提供 `BigInteger` 类的 `add` 和 `multiply` 运算。Java 语言的设计者重载了 + 运算符来完成字符串的拼接, 但没有重载其他的运算符, 也没有给 Java 程序员提供机会在他们自己的类中重载运算符。

程序清单 3-6 是对程序清单 3-5 中彩概率程序的改进，更新为使用大数。例如，假设你被邀请参加抽奖活动，并从 490 个可能的数中抽取 60 个，这个程序会告诉你中彩的概率是 $1/716395843461995557415116222540092933411717612789263493493351013459481104668848$ 。祝你好运！

在程序清单 3-5 中，会计算以下语句：

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

如果对 lotteryOdds 和 n 使用大数，则相应的语句为：

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

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         BigInteger n = in.nextBigInteger();
20
21         /*
22          * compute binomial coefficient n*(n-1)*(n-2)*...*(n-k+1)/(1*2*3*...*k)
23          */
24
25         BigInteger lotteryOdds = BigInteger.ONE;
26
27         for (int i = 1; i <= k; i++)
28             lotteryOdds = lotteryOdds
29                 .multiply(n.subtract(BigInteger.valueOf(i - 1)))
30                 .divide(BigInteger.valueOf(i));
31
32         System.out.printf("Your odds are 1 in %s. Good luck!%n", lotteryOdds);
33     }
34 }
```

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。

- BigInteger sqrt() 9

得到这个 BigInteger 的平方根。

- 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.BigDecimal 1.1

- BigDecimal(String digits)

用给定数位构造一个大实数。

- BigDecimal add(BigDecimal other)

- BigDecimal subtract(BigDecimal other)

- BigDecimal multiply(BigDecimal other)

- BigDecimal divide(BigDecimal other)

- BigDecimal divide(BigDecimal other, RoundingMode mode) 5

返回这个大实数与另一个大实数 other 的和、差、积、商。如果商是一个无限小数，第一个 divide 方法会抛出一个异常。要得到一个舍入的结果，就要使用第二个方法。RoundingMode.HALF_UP 是我们在学校里学习的四舍五入方式（即，0 到 4 舍去，5 到 9 进位）。它适用于常规的计算。其他舍入方式请参见 API 文档。

- int compareTo(BigDecimal other)

如果这个大实数与另一个大实数 other 相等，返回 0；如果这个大实数小于 other，返回负数；否则，返回正数。

3.10 数组

数组存储相同类型值的序列。下面几小节中，我们将学习在 Java 中如何使用数组。

3.10.1 声明数组

数组是一种数据结构，用来存储同一类型值的集合。通过一个整型索引（index，或称下标）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] 就是数组中索引为 i 的整数。

在声明数组变量时，需要指出数组类型（元素类型后面紧跟 []）和数组变量名。例如，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 操作符创建数组。

```
int[] a = new int[100]; // or var a = new int[100];
```

这条语句声明并初始化了一个可以存储 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。

一旦创建了数组，就不能再改变它的长度（不过，当然可以改变单个数组元素）。如果程序运行中需要经常扩展数组的大小，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见第 5 章。

 **注释：**可以使用下面两种形式定义一个数组变量：

```
int[] a;
```

或

```
int a[];
```

大多数 Java 程序员喜欢使用第一种风格，因为它可以将类型 int[]（整型数组）与变量名清晰地分开。

在 Java 中，提供了一种创建数组对象并同时提供初始值的简写形式。下面是一个例子：

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

请注意，这个语法中不需要使用 new，甚至不用指定长度。

最后一个值后面允许有逗号，如果你要不断为数组增加值，这样会很方便：

```
String[] authors =
{
    "James Gosling",
    "Bill Joy",
    "Guy Steele",
    // add more names here and put a comma after each name
};
```

还可以声明一个匿名数组（anonymous array）：

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

这个表达式会分配一个新数组并填入大括号中提供的值。它会统计初始值个数，并相应地设置数组大小。可以使用这种语法重新初始化一个数组而无须创建新变量。例如：

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

这是以下语句的简写形式：

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

注释：在 Java 中，允许有长度为 0 的数组。编写一个结果为数组的方法时，如果碰巧结果为空，这样一个长度为 0 的数组就很有用。可以如下构造一个长度为 0 的数组：

```
new elementType[0]
```

或

```
new elementType[] {}
```

注意，长度为 0 的数组与 null 并不相同。

3.10.2 访问数组元素

数组元素从 0 开始编号。最后一个合法的索引为数组长度减 1。在下面的例子中，索引值为 0 ~ 99。一旦创建了数组，就可以在数组中填入元素。例如，可以使用一个循环：

```
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，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，则必须为元素提供空串：

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

警告：如果创建了一个包含 100 个元素的数组，然后试图访问元素 a[100]（或在 0 ~ 99 之外的任何其他索引），就会出现“array index out of bounds”（数组索引越界）异常。

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]);
```

3.10.3 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（或者任何其他元素集合）中的每个元素，而不必考虑指定索引值。

这种增强的 for 循环形式如下：

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

它将给定变量（variable）设置为集合中的每一个元素，然后执行语句（statement）（当然，也可以是语句块）。collection 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，另外 Iterable 接口将在第 9 章中讨论。

例如，

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

会打印数组 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 语言的设计者也曾考虑过使用诸如 foreach 和 in 这样的关键字，但这种循环并不是最初就包含在 Java 语言中，而是后来添加的，没有人希望破坏已经包含同名方法或变量（例如 System.in）的老代码。

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”循环更加简洁、更不易出错，因为你不必为起始和终止索引值而操心。

注释：“for each”循环的循环变量将会遍历数组中的每个元素，而不是索引值。

如果需要处理一个集合中的所有元素，相比传统循环，“for each”循环是个让人欣喜的改进。不过，很多情况下还是需要使用传统的 for 循环。例如，可能不希望遍历整个集合，或者可能需要在循环内部使用索引值。

提示：有一个更容易的方法可以打印数组中的所有值，即利用 Arrays 类的 `toString` 方法。调用 `Arrays.toString(a)` 会返回一个包含数组元素的字符串，这些元素包围在中括号内，并用逗号分隔，例如，"[2,3,5,7,11,13]"。要想打印数组，只需要调用

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

3.10.4 数组拷贝

在 Java 中，允许将一个数组变量拷贝到另一个数组变量。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。

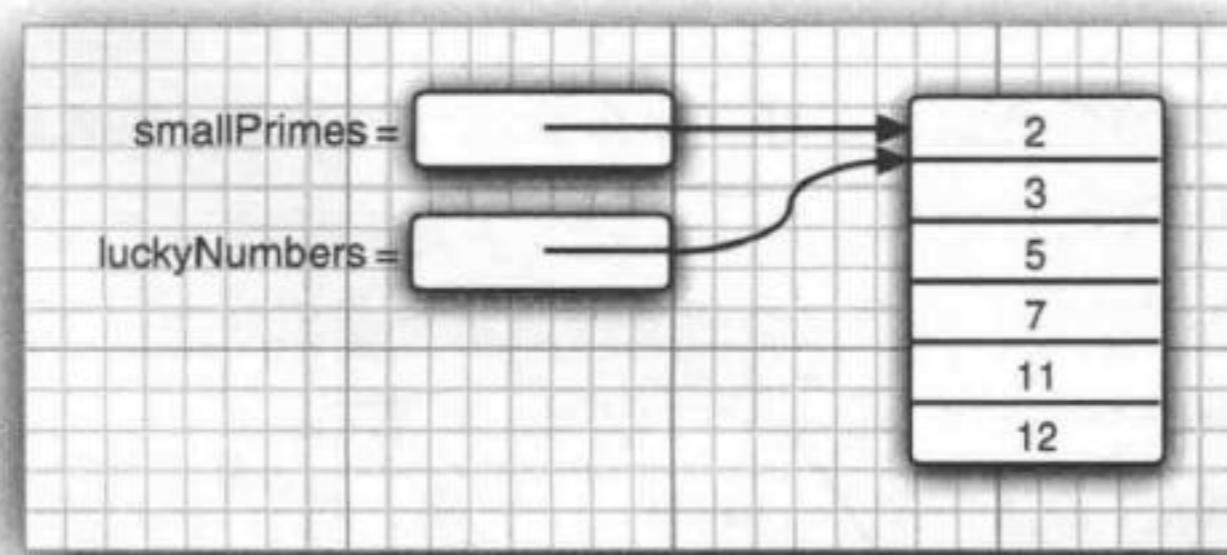


图 3-14 拷贝一个数组变量

如果确实希望将一个数组的所有值拷贝到一个新的数组中，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，那么新增的元素将填入 0；如果数组元素是布尔型，则填入 `false`。相反，如果长度小于原数组的长度，则只拷贝前面的值。

 **C++ 注释：**Java 数组与堆栈上的 C++ 数组有很大不同，但基本上与在堆（heap）上分配的数组指针一样。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 `[]` 运算符预定义为会完成越界检查（bounds checking）。另外，没有指针运算，就意味着不能通过 `a` 加 1 得到数组中的下一个元素。

3.10.5 命令行参数

前面已经看到一个例子，其中一个 Java 数组重复出现过很多次。每一个 Java 程序都有一个带 `String args[]` 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，也就是命令行上指定的参数。

例如，来看下面这个程序：

```
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("!");
    }
}
```

如果如下调用这个程序：

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

`args` 数组将包含以下内容：

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

这个程序会显示下面这个消息：

```
Goodbye, cruel world!
```

C++ 注释：在 Java 程序的 `main` 方法中，程序名并不存储在 `args` 数组中。例如，从命令行如下运行一个程序时

```
java Message -h world
```

`args[0]` 是 `"-h"`，而不是 `"Message"` 或 `"java"`。

3.10.6 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（QuickSort）算法。快速排序算法对于大多数数据集都很高效。`Arrays` 类还提供了另外一些很便捷的方法，在这一节最后的 API 注释中将介绍这些方法。

程序清单 3-7 中的程序具体使用了数组，它会为一个抽彩游戏生成一个随机的数字组合。例如，假如从 49 个数字中抽取 6 个数，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
```

要想选择这样一组随机的数字，首先将值 $1, 2, \dots, n$ 填入数组 `numbers` 中：

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

第二个数组存放抽取出来的数：

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

现在可以开始抽取 k 个数了。`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，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 个元素设置为该索引对应的数 (`numbers[r]`)，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到那个数，因为所有抽彩数字必须各不相同。因此，这里用数组中的最后一个数覆盖 `number[r]`，并将 `n` 减 1。

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

关键在于每次抽取的都是索引，而不是实际的值。这个索引指向一个数组，其中包含尚未抽取过的值。

在抽取了 `k` 个数之后，可以对 `result` 数组进行排序，来得到更美观的输出：

```
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         // sort the result array
41         Arrays.sort(result);
42
43         // print the result
44         for (int r : result)
45             System.out.println(r);
46     }
47 }
```

```

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(xxx[] a) 5`

返回一个字符串，其中包含 a 中的元素，这些元素用中括号包围，并用逗号分隔。在这个方法以及后面的方法中，数组元素类型 xxx 可以是 int、long、short、char、byte、boolean、float 或 double。

- `static xxx[] copyOf(xxx[] a, int end) 6`

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

返回与 a 类型相同的一个数组，其长度为 end 或者 end-start，并填入 a 的值。如果 end 大于 a.length，结果会填充 0 或 false 值。

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

使用优化的快速排序算法对数组进行排序。

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

- `static int binarySearch(xxx[] a, int start, int end, xxx v) 6`

使用二分查找算法在有序数组 a 中查找值 v。如果找到 v，则返回相应的索引；否则，返回一个负数值 r。-r-1 是 v 应插入的位置（为保持 a 有序）。

- `static void fill(xxx[] a, xxx v)`

将数组的所有元素设置为 v。

- `static boolean equals(xxx[] a, xxx[] b)`

如果两个数组长度相同，并且相同索引对应的元素都相同，则返回 true。

3.10.7 多维数组

多维数组使用多个索引访问数组元素，它适用于表格或其他更复杂的排列形式。你可以先跳过这一节的内容，等真正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格，用来显示在不同利率下投资 10 000 美元有多少收益，利息每年兑现并复投（见表 3-8）。

表 3-8 不同利率下的投资收益情况

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

(续)

10%	11%	12%	13%	14%	15%
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）来存储这些信息，名为 balances。

在 Java 中，声明一个二维数组相当简单。例如：

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

数组在进行初始化之前是不能使用的。在这里可以如下初始化：

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

其他情况下，如果知道数组元素，可以不调用 new，而直接使用一种简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}
};
```

一旦初始化数组，就可以利用两对中括号访问单个元素，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 和一个存储账户余额的二维数组 balances（对应每个年度和利率分别有一个余额），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;
```

然后计算其他行，如下所示：

```
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” 循环语句不会自动循环处理二维数组的所有元素。它会循环处理行，而这些行本身就是一维数组。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 }
```

```

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 }

```

3.10.8 不规则数组

到目前为止，我们看到的数组与其他程序设计语言中的数组没有多大区别。但在底层实际存在着一些细微的差异，有时你可以充分利用这一点：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（请参见图 3-15）。

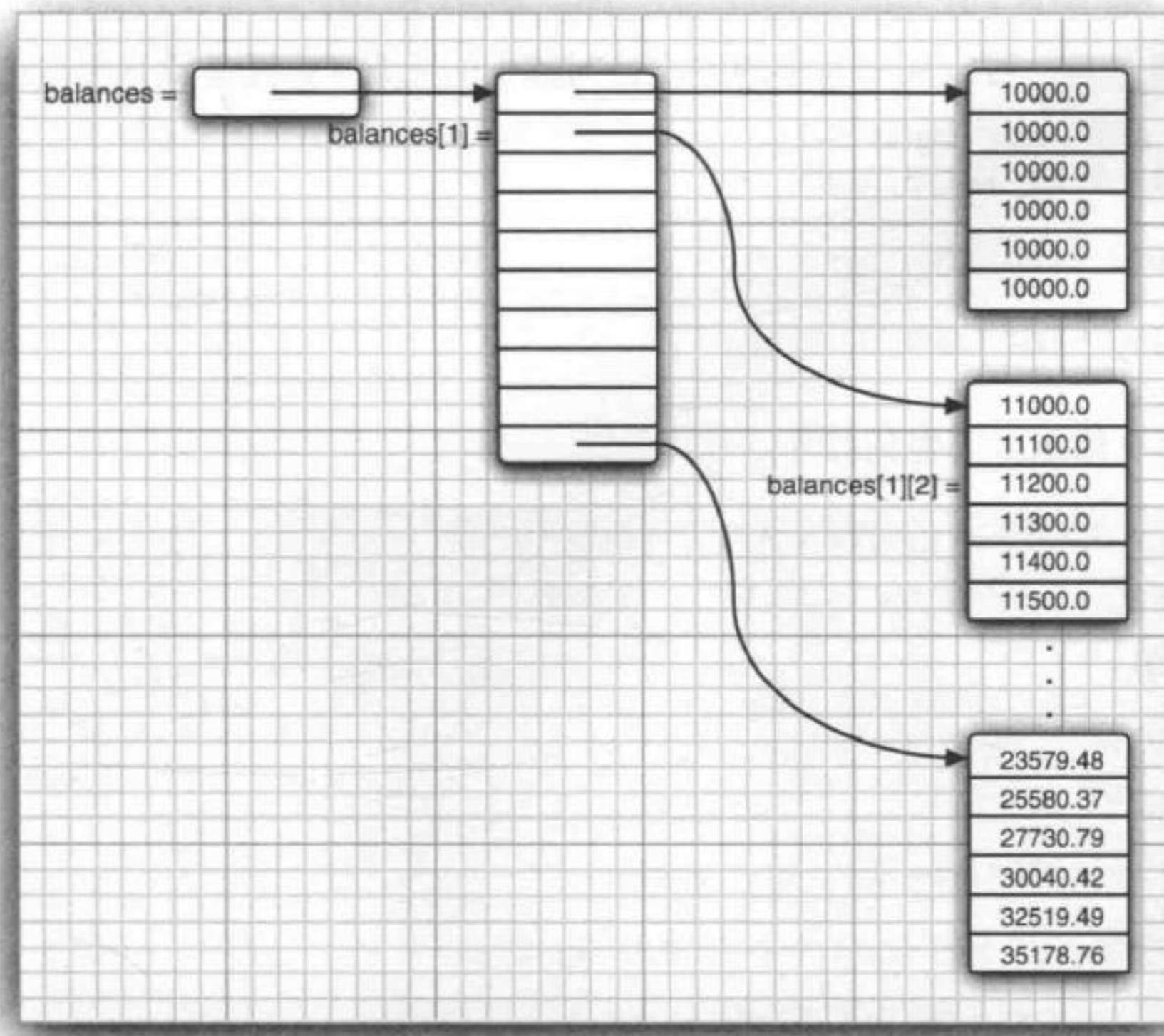


图 3-15 一个二维数组

表达式 `balances[i]` 指示第 i 个子数组，也就是表格的第 i 行。它本身也是一个数组，`balances[i][j]` 指示这个数组的第 j 个元素。

由于可以单独地访问数组的某一行，所以可以让两行交换。

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

还可以很容易地构造一个“不规则”数组，即数组的每一行有不同的长度。下面是一个标准的示例。我们将创建一个数组，第 i 行第 j 列的元素将存放“从 i 个数中抽取 j 个数”可能的结果数。

```
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，这种选择只有一种可能）。要想创建这样一个不规则的数组，首先需要分配一个数组包含这些行：

```
final int NMAX = 10;
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];
```

分配了数组之后，可以采用通常的方式访问其中的元素（前提是沒有超出边界）：

```
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，这个指针数组的每一个元素会填充一个包含 6 个数字的数组：

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

庆幸的是，当调用 `new double[10][6]` 时，这个循环是自动的。需要不规则的数组时，只能单独地分配行数组。

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，我们已经了解了 Java 语言的基本程序结构，下一章将介绍 Java 面向对象程序设计。