

第3章 Java基本的程序设计结构

一个简单的Java应用程序

注释

数据类型

变量

运算符

字符串

输入输出

控制流程

大数值

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。本章主要讲述程序设计相关的基本概念（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、添加文本框和按钮等。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，它们仅仅通过shell窗口输入输出。

最后需要说明，对于一个具有C++编程经验的程序员来说，本章的内容只需要浏览一下，应该重点阅读分布在正文中的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研究一下。首先，Java对大小写敏感。如果出现了大小写拼写错误（例如，将main拼写成Main），那程序将无法运行。

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），它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）。

从类名FirstSample可以看出，标准的命名规范为：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有类的名字相同，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
```

运行编译程序时，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://java.sun.com/docs/books/jls>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.sun.com/bugdatabase/index.jsp>上输入bug号码4252539查看一下。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://java.sun.com/docs/books/vmspec>）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及以后的版本中将强制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一件很头痛的事情，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的很清楚。另一方面，Sun公司把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。“bug展示”对程序员来说是一种十分有用的资源，甚至程序员可以对感兴趣的bug进行“投票”。得票多的bug在下一个将要发布的JDK版本中得到解决的可能性就大。

需要注意源代码中的括号{ }。在Java中，像在C/C++中一样，用花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。

花括号的使用风格曾经引发过许多无谓的争论。我们的习惯是把匹配的花括号上下对齐。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任意风格。在下面讲述各种循环语句时，我们还会详细地介绍花括号的使用。

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方法没有给操作系统返回“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!'");
}
```

一对花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，以便每次调用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变元）。对于一个方法，即使没有参数也需要书写圆括号。例如，不带参数的println方法只打印一个空行。使用下面的语句：

```
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符‘ 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，有三种书写注释的方式。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。请参见例3-1。

例3-1 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. }
```

第三种注释可以用来自动地生成文档。这种注释以/**开始，以*/结束。有关这种注释的详细信息和自动生成文档的具体方法请参见第4章。

 警告：在Java中，/* */注释不能嵌套。也就是说，如果代码本身包含了一个*/，就不能用/*和*/将注释括起来。

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
int	4字节	- 2 147 483 648 ~ 2 147 483 647 (正好超过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有效的整型，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，所以每一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（如4000000000L）。十六进制数值有一个前缀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。



C++注释：在C和C++中，int表示的整型与目标机器相关。在8086这样的16位处理器上整型数值占2字节；在Sun SPARC这样的32位处理器上，整型数值占4字节；而在Intel Pentium处理器上，C和C++整型依赖于具体的操作系统，对于DOS和Windows 3.1，整型数值占2字节。当Windows程序使用32位模式时，整型数值占4字节。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没有任何无符号类型（unsigned type）。

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。例如，用7位有

效数字足以精确地表示普通雇员的年薪，但表示公司总裁的年薪可能就不够用了。实际上，只有很少的情况适合使用float类型，例如，需要快速地处理单精度数据，或者需要存储大量数据。

float类型的数值有一个后缀F（例如，3.402F）。没有后缀F的浮点数值（如3.402）默认为double类型。当然，也可以在浮点数值后面添加后缀D（例如，3.402D）。

☑ 注释：在JDK 5.0中，可以使用十六进制表示浮点数值。例如，0.125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。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（与相应的Float类型的常量一样）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：浮点数值不适用于禁止出现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，而不是人们想像的0.9。其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的表示分数1/10。这就好像十进制无法精确地表示1/3一样。如果需要在数值计算中不含有任何舍入误差，就应该使用BigDecimal类，本章稍后将介绍这个类。

3.3.3 char类型

char类型用于表示单个字符。通常用来表示字符常量。例如：'A'是编码为65所对应的字符常量。与"A"不同，"A"是一个包含字符A的字符串。Unicode编码单元可以表示为十六进制值，其范围从\u0000到\Uffff。例如：\u2122表示注册商标，\u03C0表示希腊字母π。

除了可以采用转义序列符\u表示Unicode代码单元的编码之外，还有一些用于表示特殊字符的转义序列符，请参看表3-3。所有这些转义序列符都可以出现在字符常量或字符串的引号内。例如，"\u2122"或"Hello\n"。转义序列符\u还可以出现在字符常量或字符串的引号之外（而其他所有转义序列不可以）。例如：

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

这种形式完全符合语法规则，\u005B和\u005D是（和）的编码。

表3-3 特殊字符的转义序列符

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			

要想弄清char类型，就必须了解Unicode编码表。Unicode打破了传统字符编码方法的限制。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、俄国的KOI-8、中国的GB118030和BIG-5等等。这样就产生了下面两个问题：一个是对于任意给定的代码值，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能够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、日语和韩国语言中的表意文字。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JDK 5.0开始。代码点（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，代码点采用十六进制书写，并加上前缀U+，例如U+0041就是字母A的代码点。Unicode的代码点可以分成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），代码点从U+0000到U+FFFF，其中包括了经典的Unicode代码；其余的16个附加级别，代码点从U+10000到U+10FFFF，其中包括了一些辅助字符（supplementary character）。

UTF-16编码采用不同长度的编码表示所有Unicode代码点。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。这样构成的编码值一定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，对于整数集合的数学符号，它的代码点是U+1D56B，并且是用两个代码单元U+D835和U+DD6B编码的（有关编码算法的描述请参看<http://en.wikipe-dia.org/wiki/UTF-16>）。

在Java中，char类型用UTF-16编码描述一个代码单元。

我们强烈建议不要在程序中使用char类型，除非确实需要对UTF-16代码单元进行操作。最好将需要处理的字符串用抽象数据类型表示（有关这方面的内容将在稍后讨论）。

3.3.4 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，数值或指针可以代替boolean值。整数0相当于布尔值false，非0值相当于布尔值true。在Java中则不行。因此，Java应用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，每一个变量属于一种类型 (type)。在声明变量时，变量所属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语句，所以必须以分号结束。

变量名必须是一个以字母开头的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”的范围要大。字母包括'A'~'Z'、'a'~'z'、'_'或在某种语言中代表字母的任何Unicode字符。例如，德国的用户可以在变量名中使用字母'ä'；希腊人可以使用 π 。同样，数字包括'0'~'9'和在某种语言中代表数字的任何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方法进行检测。

另外，不能将变量名命名为Java保留字（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; // ok--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，千万不要使用未被初始化的变量。例如，Java编译器认为下面语句序列是错误的：

```
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;
```

是定义一个变量，而

```
extern int i;
```

是声明一个变量。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声明常量。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 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);
    }

    public static final double CM_PER_INCH = 2.54;
}
```

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。

可以在赋值语句中采用一种简化的格式书写二元算术运算符。

例如，

```
x += 4;
```

等价于

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

（通常，将运算符放在赋值号的左侧，如*= 或 %=。）



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double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，下列运算：

```
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，虚拟机设计者允许将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产生理想的结果。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。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，这两种方式的差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 自增运算符与自减运算符

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++的实现方式，也使用了自增、自减运算符：n++将变量n的当前值加1；n--将n的值减1。例如：

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

n的值将变为13。因为这些运算符改变了变量的值，所以它的操作数不能是数值。例如，4++就是一条非法的语句。

实际上，这两个运算符有两种形式。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。两种方式都是对变量值加1。但在表达式中，这两种形式就有区别了。前缀方式先进行加1运算；后缀方式则使用变量原来的值。

```
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
```

我们建议不要在其他表达式的内部使用++，这样编写的代码很容易令人带来迷惑的感觉，并会产生烦人的bug。

当然，++运算符作为C++语言名称的一部分，也引发了有关程序设计语言的第一个笑话。C++的反对者认为这种语言的名称也存在着bug，他们说：“因为只有对它改进之后，我们才有可能使用它，所以它的名字应该命名为++C。”

3.5.2 关系运算符与boolean运算符

Java包含各种关系运算符。其中，使用两个等号 == 检测是否相等。例如，3 == 7的值为false。

使用 != 检测是否不相等。例如，3 != 7的值为true。

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

Java沿用了C++的习惯，用&&表示逻辑“与”、用||表示逻辑“或”。从!=运算符很容易看出，!表示逻辑“非”。&&和||是按照“短路”方式求值的。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&&对两个表达式进行计算：

$expression_1 \ \&\& \ expression_2$

并且第一个表达式值为false，结果不可能为真。因此，第二个表达式的值就没有必要计算了。这种方式可以避免一些错误的发生。例如，表达式：

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

当x为0时，不会计算第二部分。因此，若x为0，1/x不被计算，也不会出现除以0的错误。

与之类似，对于 $expression_1 \ || \ expression_2$ ，当第一个表达式为true时，结果自动为true，不必再计算第二部分。

最后，Java支持三元操作?:。在很多时候，这个操作非常有用。表达式

$condition \ ? \ expression_1 : expression_2$

当条件condition为真时计算第1个表达式，否则计算第2个表达式。例如：

```
x < y ? x : y
```

返回x和y中较小的那个值。

3.5.3 位运算符

在处理整型数值时，可以直接对组成整型数值的各个位进行操作。这意味着可以使用屏蔽技术获得整数中的各个位。位运算符包括：

& (“与”) | (“或”) ^ (“异或”) ~ (“非”)

这些运算符在位模式下工作。例如，如果n是一个整型变量，并且用二进制表示的n从右数第4位为1，那么

```
int fourthBitFromRight = (n & 8) / 8;
```

返回1；否则返回0。通过运用2的幂次方的&运算可以将其他位屏蔽掉，而只保留其中的某一位。



注释：&和|运算符应用于布尔值，得到的结果也是布尔值。这两个运算符与&&和||的运算非常类似，只是不按“短路”方式计算。即在得到计算结果之前，一定要计算两个操作数的值。

另外，“>>”和“<<”运算符将二进制位进行右移或左移操作。当需要建立位模式屏蔽某些位时，使用这两个运算符十分方便：

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

最后，>>>运算符将用0填充高位；>>运算符用符号位填充高位。没有<<<运算符。



警告：对移位运算符右侧的参数需要进行模32的运算（除非左边的操作数是long类型，在这种情况下需对右侧操作数模64）。例如， $1 \ll 35$ 与 $1 \ll 3$ 或8是相同的。



C++注释：在C或C++中无法确定>>操作执行的是算术移位（扩展符号位），还是逻辑移位（高位填0）。在执行中将会选择效率较高的一种。这就是说，在C/C++中，>>运算符实际上只是为非负数定义的。Java消除了这种含糊性。

3.5.4 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方法操作一个定义在System类中的System.out对象。但是，Math类中的sqrt方法操作的不是对象，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。

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：

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

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：从JDK 5.0开始，不必在数学方法名和常量名前添加前缀“Math.”，而只要在源文件的顶部加上下列内容就可以了。

例如：

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

在第4章中将讨论静态导入。

```
System.out.println("The square root of \u03C0 is " + sqrt(PI));
```



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<http://www.netlib.org/fdlibm/index.html>（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）。

3.5.5 数值类型之间的转换

在程序运行时，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。

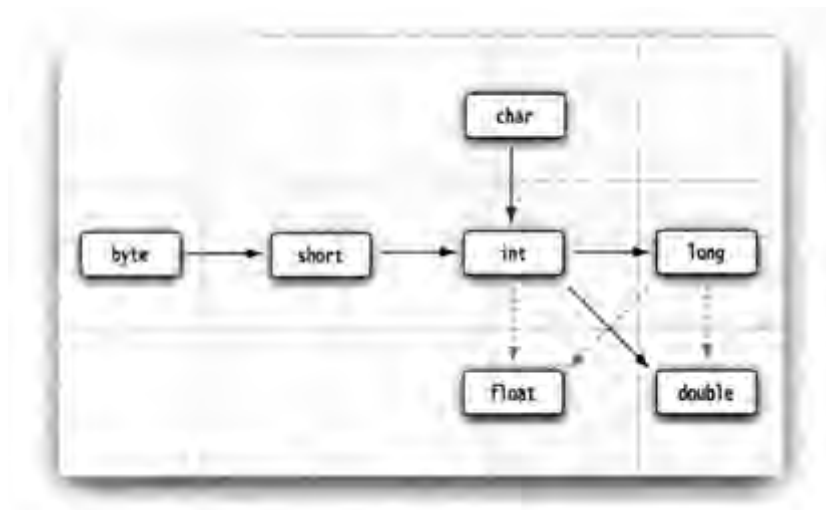


图3-1 数值类型之间的合法转换

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 1.23456792E8
```

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类型的，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。

3.5.6 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，希望使用这种操

作方式)，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。

3.5.7 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
! ~ ++ -- + (一元运算) - (一元运算) () (强制类型转换) new	从右向左
* / %	从左向右
+ -	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

(续)

运 算 符	结 合 性
	从左向右
&&	从左向右
	从左向右
?:	从右向左
= += -= *= /= %= &= = ^= <<= >>= >>>=	从右向左

3.5.8 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）。

从JDK 5.0开始，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，可以声明这样一种类型的变量：

```
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“Expletivedeleted”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得到“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（单词is后面加了一个空格，输出时也会加上这个空格）。

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，可以想像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，将源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“构建字符串”中将详细地给予介绍)。



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字符串更加像char*指针，

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

当采用另一个字符串替换greeting的时候，Java代码主要进行下列操作：

```
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，也可以是字符串常量。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")
```

一定不能使用 == 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 代码点与代码单元

Java字符串由char序列组成。从前面已经看到，字符数据类型是一个采用UTF-16编码表示Unicode代码点的代码单元。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，即代码点数量，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`个代码点，应该使用下列语句

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



注释：Java以独特的风格对字符串中的代码单元计数：字符串中的第一个代码单元位置为0。这种习惯起源于C，这样处理主要出于技术上的原因。具体理由似乎已经淡忘，而麻烦却保留了下来。但是，许多程序员习惯于这种风格，因而Java设计者也就将其保留了下来。

为什么会对代码单元如此大惊小怪？请考虑下列语句：

```
 $\mathbb{Z}$  is the set of integers
```

使用UTF-16编码表示 \mathbb{Z} 需要两个代码单元。调用

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

返回的不是空格，而是第二个代码单元 \mathbb{Z} 。为了避免这种情况的发生，请不要使用char类型。这太低级了。

如果想要遍历一个字符串，并且依次查看每一个代码点，可以使用下列语句：

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

非常幸运，codePointAt方法能够辨别一个代码单元是辅助字符的第一部分还是第二部分，并能够返回正确的结果。也就是说，可以使用下列语句实现回退操作：

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

3.6.6 字符串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章给予解释。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，而是选择一些使用最频繁的方法，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Reading the On-Line API Documentation）。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。



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或结束的代码点。
- int offsetByCodePoints(int startIndex, int cpCount) 5.0
返回从startIndex代码点开始，位移cpCount后的代码点索引。
- 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。
- boolean endsWith(String suffix)
如果字符串以suffix结尾，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。
- 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 lastIndexOf(int cp)`
- `int lastIndexOf(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。
- `boolean startsWith(String prefix)`
如果字符串以`prefix`字符串开始，返回`true`。
- `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()`
返回一个新字符串。这个字符串将原始字符串中的所有大写字母改成了小写字母。
- `String toUpperCase()`
返回一个新字符串。这个字符串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。

3.6.7 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。

可以看到，屏幕被分成三个窗框。在左上方的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。



图3-2 API文档的三个窗格



图3-3 String类的描述

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。



提示：马上在浏览器中将docs/api/index.html页面做一个书签。

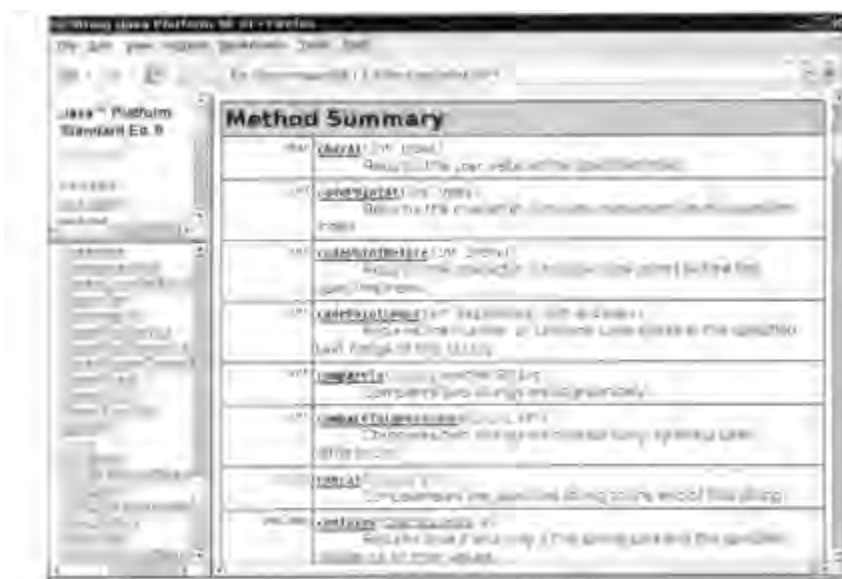


图3-4 String类方法的小结



图3-5 String方法的详细描述

3.6.8 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();
```

(有关构造器和new操作符的内容将在第4章详细地介绍)。

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，其效率略微有些低，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（通常都是这样）编辑，则应该用StringBuilder替代它。这两个类的API是相同的。

下面的API注解包含了StringBuilder类中的重要方法。



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7章～第9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);
```

（构造器和`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。

在例3-2的程序中，询问用户姓名和年龄，然后打印一条如下格式的消息：

```
Hello, Cay. Next year, you'll be 46
```

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。

例3-2 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（数组处理将在本章稍后介绍）。

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。



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()
- 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()
检测是否还有表示整数或浮点数的下一个字符序列。



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

```
3333.3333333333335
```

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("%.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	日期时间	见表3-7
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);
```

尽管在第4章之前，没有对Date类型进行过详细地描述，但基于完整性的考虑，还是简略地介绍一下printf方法中日期与时间的格式化选项。在这里，使用以t开始，以表3-7中任意字母结束的两个字母格式。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 2004

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 2004
F	ISO 8601日期	2004-02-09
D	美国格式的日期（月/日/年）	02/09/2004
T	24小时时间	18:05:19
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）	2004

(续)

转 换 符	类 型	举 例
y	年的后两位数字 (前面补0)	04
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	两位数字的月 (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	上午或下午的大写标志	PM
p	上午或下午的小写标志	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, 2004
```

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。



注释: 许多格式化规则是本地环境特有的。例如, 在德国, 十进制的分隔符是句号而不是逗号, Monday被格式化为Montag。在卷II中将介绍如何控制应用程序与地域有关的行为。

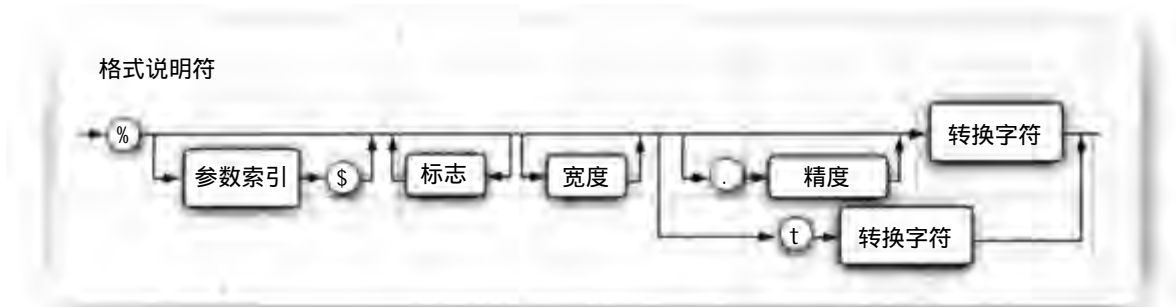


图3-6 格式说明符语法

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(new File("myfile.txt"));
```

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”。

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");
```

如果文件不存在，则可以像输出到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“被零整除”异常更严重。在第11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“找不到文件”的异

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 FileNotFoundException
{
    Scanner in = new Scanner(new File("myfile.txt"));
    . . .
}
```

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、读取目录以及编写压缩文件，请参看卷II第1章。



注释：当采用命令行方式启动一个程序时，可以利用重定向将任意文件捆绑到System.in和System.out：

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

这样，就不必担心处理FileNotFoundException异常了。



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。



java.io.PrintWriter 1.1

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



java.io.File 1.0

- File(String fileName)
用给定文件名，构造一个描述文件的File对象。注意这个文件当前不必存在。

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，Java SE 5.0还添加了一种变形的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）是指由一对花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），格式为

```
{
    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）。

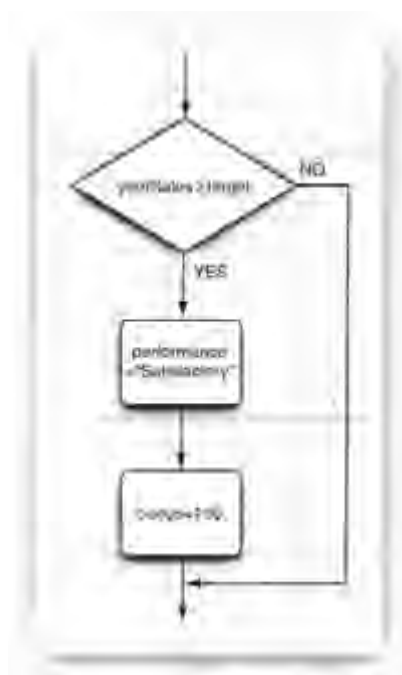


图3-7 if语句的流程图



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，比较常见的条件语句格式如下所示（请参看图3-8）：

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

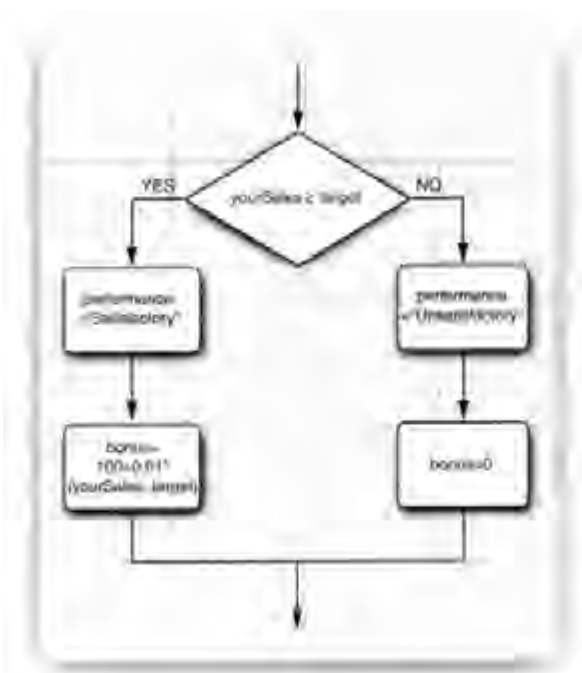


图3-8 if/else语句的流程图

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：

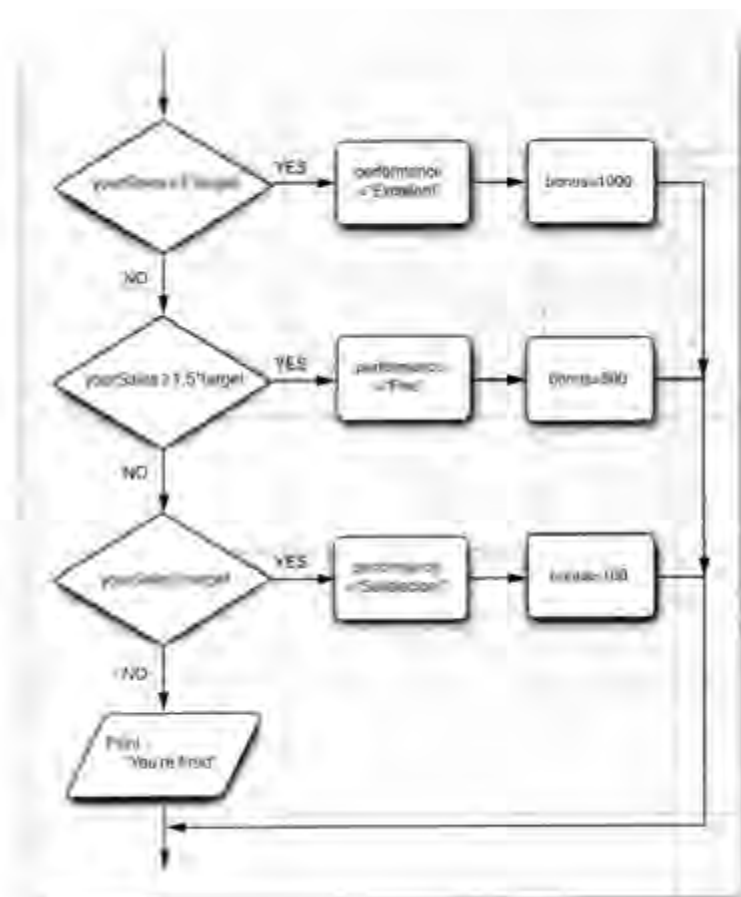


图3-9 if/else if（多分支）的流程图

```
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）。常用的格式为

```
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）。

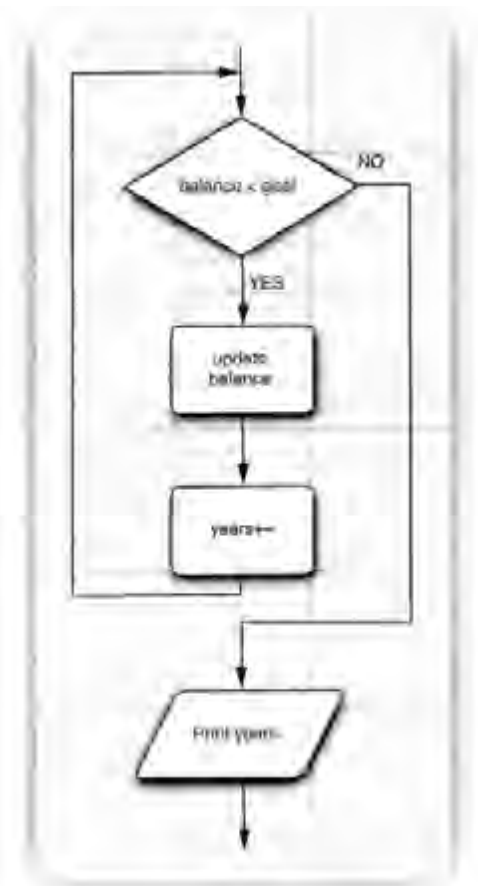


图3-10 while语句的流程图

例3-3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。

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。在例3-4的代码中，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。

例3-3 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. }
```

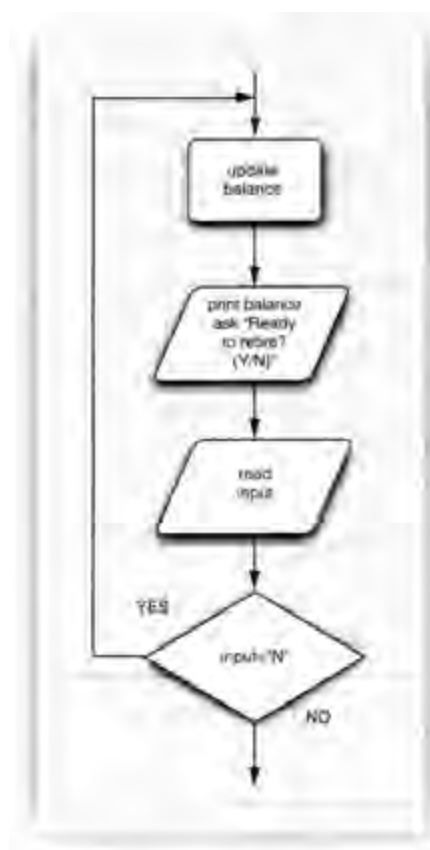


图3-11 do/while语句的流程图

例3-4 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);
```

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。

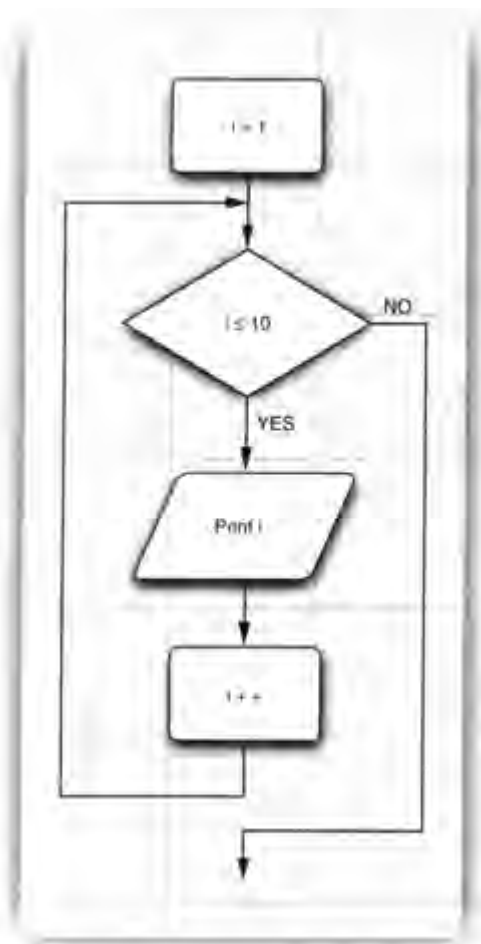


图3-12 for语句的流程图

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 98跳到10.099 999 999 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 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--;
}

```

例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！

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。

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

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：稍后将会介绍“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。

例3-5 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  $n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot (n-k+1) / (1 \cdot 2 \cdot 3 \cdot \dots \cdot k)$ 
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，就应该使用下列代码：

```
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;  
}
```

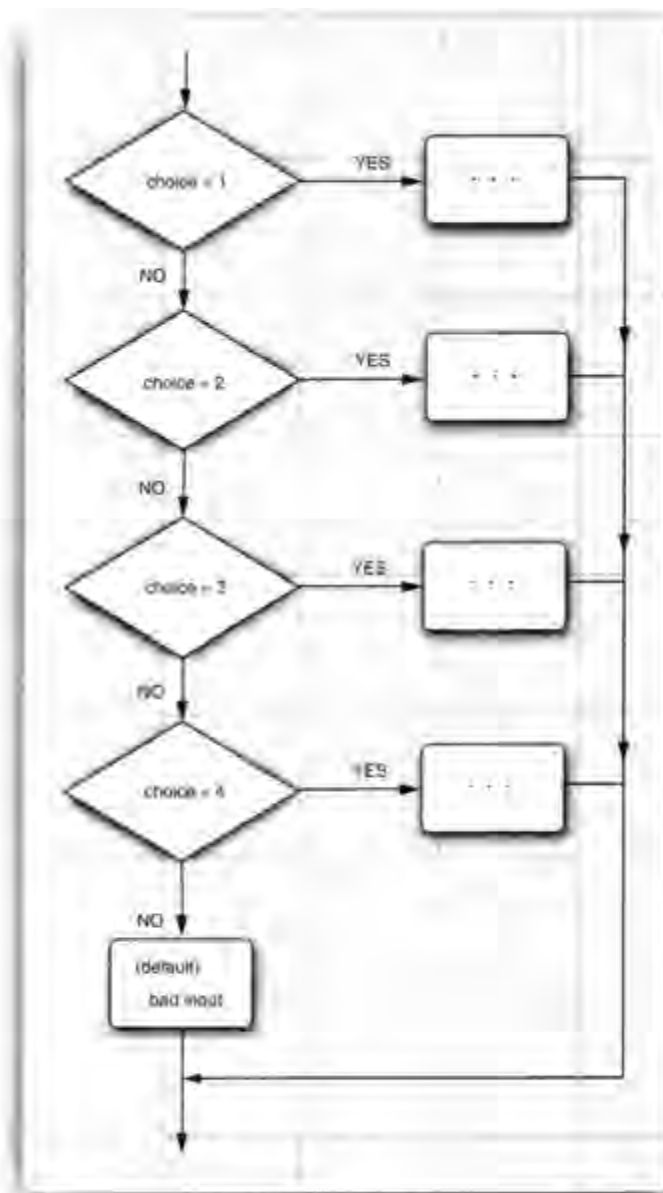


图3-13 switch语句的流程图

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。

case标签必须是整数或枚举常量，不能测试字符串。例如，下面这段代码就存在错误。

```
String input = ...;
switch (input) // ERROR
{
    case "A": // ERROR
        ...
        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，偶尔地使用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 > 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，一下这个循环：

```
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程序员自己重载运算符的权利。

例3-6是对例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1/716395843461995557415116222540092933411717612789263493493351013459481104668848。祝你好运！

在例3-5程序中，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));
```

例3-6 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. }
```

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。

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 $x / 10^{\text{scale}}$ 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;
```

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。



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 0 to 99
```

 **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 SE 5.0增加了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接口的内容将在卷II的第2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（关于null的详细论述请参看第4章）。

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。

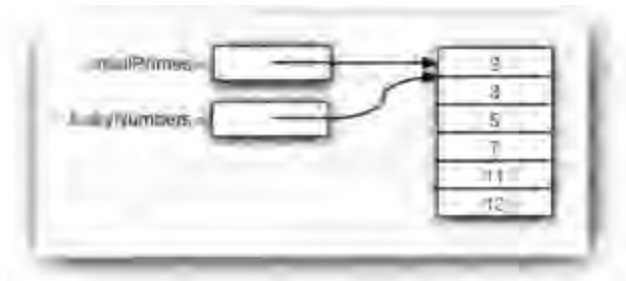


图3-14 拷贝一个数组变量

数组to必须有足够的空间存放拷贝的元素。

☑ 注释：在Java SE 6之前，用System类的arraycopy方法将一个数组的元素拷贝到另一个数组中。调用这个方法的语法格式为：

```
System.arraycopy(from, fromIndex, to, toIndex, count);
```

数组to必须有足够的空间存放拷贝的元素。

例如，下面这段语句所得到的结果如图3-15所示。它创建了两个数组，然后将第一个数组的后4个元素拷贝到第二个数组中。拷贝从原始数组的第2个位置开始，一共拷贝4个元素，目标数组的起始位置为3。

```
int[] smallPrimes = {2, 3, 5, 7, 11, 13};
int[] luckyNumbers = {1001, 1002, 1003, 1004, 1005, 1006, 1007};
System.arraycopy(smallPrimes, 2, luckyNumbers, 3, 4);
for (int i = 0; i < luckyNumbers.length; i++)
    System.out.println(i + ": " + luckyNumbers[i]);
```

输出结果为

```
0: 1001
1: 1002
2: 1003
3: 5
4: 7
5: 11
6: 13
```

☑ C++注释：Java数组与C++数组在堆栈上有很大的不同，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。

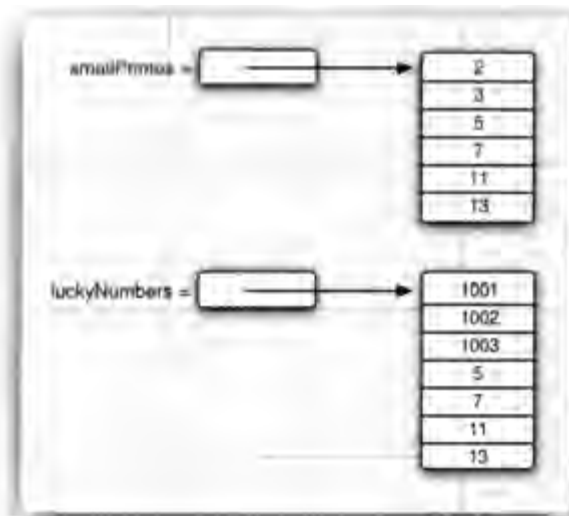


图3-15 数组之间拷贝元素值

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[]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[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。Array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。

例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，这样可以使输出效果更加清晰：

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


例3-7 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
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。

API java.lang.System 1.1

- `static void arraycopy(Object from, int fromIndex, Object to, int toIndex, int count)`
将第一个数组中的元素拷贝到第二个数组中。
- 参数：from 任意类型的数组（在第5章中将解释为什么这个参数的类型是Object）。
fromIndex 原始数组中待拷贝元素的起始下标。
to 与from同类型的数组。
toIndex 目标数组放置拷贝元素的起始下标。
count 拷贝的元素数量。

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。表3-8是相应的图示。

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balance。

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 < balance[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;
    }
}

```

例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]]
```

例3-8 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3-16）。

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

```

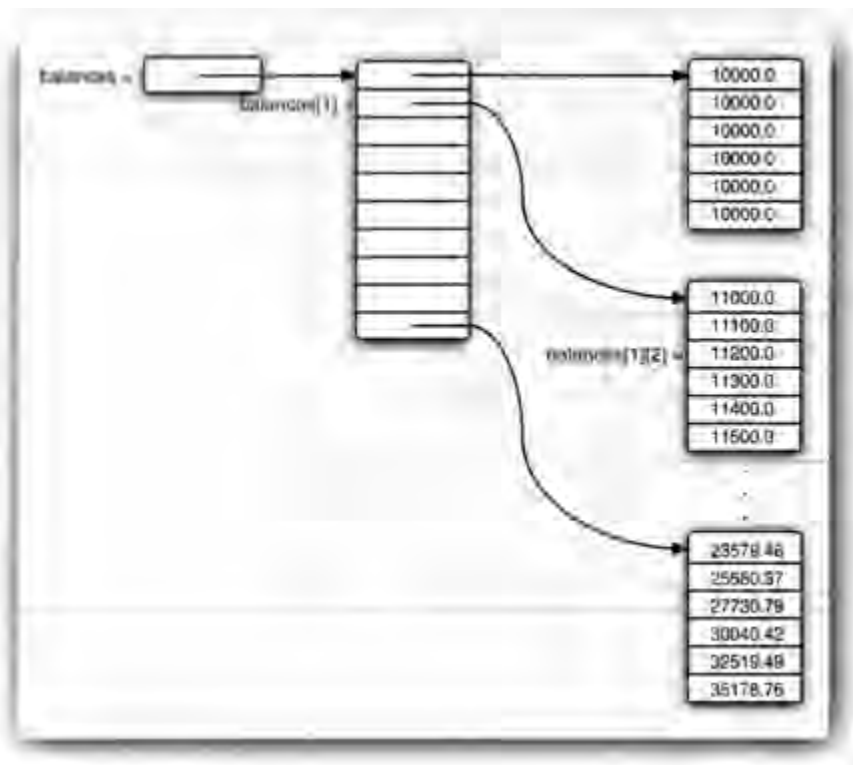


图3-16 一个二维数组

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;
    }
```

例3-9给出了完整的程序。



C++注释：在C++中，Java的声明

```
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++
```

而是创建了一个包含10个指针的一个数组：

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

然后，指针数组的每一个元素被分配了一个包含6个数值的数组：

```
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。

例3-9 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，下一章节将介绍Java中的面向对象的程序设计。