第 3 章 控制程序流程

“就象任何有感知的生物一样，程序必须能操纵自己的世界，在执行过程中作出判断与选择。”

在 Java 里，我们利用运算符操纵对象和数据，并用执行控制语句作出选择。Java 是建立在 C++基础上的，所以对 C 和 C++程序员来说，对 Java 这方面的大多数语句和运算符都应是非常熟悉的。当然，Java 也进行了自己的一些改进与简化工作。

3.1 使用 Java 运算符

运算符以一个或多个自变量为基础，可生成一个新值。自变量采用与原始方法调用不同的一种形式，但效果是相同的。根据以前写程序的经验，运算符的常规概念应该不难理解。

加号（+）、减号和负号（-）、乘号（*）、除号（/）以及等号（=）的用法与其他所有编程语言都是类似的。

所有运算符都能根据自己的运算对象生成一个值。除此以外，一个运算符可改变运算对象的值，这叫作“副作用”（Side Effect）。运算符最常见的用途就是修改自己的运算对象，从而产生副作用。但要注意生成的值亦可由没有副作用的运算符生成。

几乎所有运算符都只能操作“主类型”（Primitives）。唯一的例外是“=”、“==”和“!=”，它们能操作所有对象（也是对象易令人混淆的一个地方）。除此以外，String 类支持“+”和“+=”。

3.1.1 优先级

运算符的优先级决定了存在多个运算符时一个表达式各部分的计算顺序。

Java 对计算顺序作出了特别的规定。其中，最简单的规则就是乘法和除法在加法和减法之前完成。程序员经常都会忘记其他优先级规则，所以应该用括号明确规定计算顺序。例如：

```
A = X + Y - 2/2 + Z;
```

为上述表达式加上括号后，就有了一个不同的含义。

```
A = X + (Y - 2)/(2 + Z);
```

3.1.2 赋值

赋值是用等号运算符（=）进行的。它的意思是“取得右边的值，把它复制到左边”。右边的值可以是任何常数、变量或者表达式，只要能产生一个值就行。但左边的值必须是一个明确的、已命名的变量。也就是说，它必须有一个物理性的空间来保存右边的值。举个例子来说，可将一个常数赋给一个变量（A=4;），但不可将任何东西赋给一个常数（比如不能 4=A）。

对主数据类型的赋值是非常直接的。由于主类型容纳了实际的值，而且并非指向一个对象的句柄，所以在为其赋值的时候，可将来自一个地方的内容复制到另一个地方。例如，假设为主类型使用“A=B”，那么 B 处的内容就复制到 A。若接着又修改了 A，那么 B 根本不会受这种修改的影响。作为一名程序员，这应成为自己的常识。

但在为对象“赋值”的时候，情况却发生了变化。对一个对象进行操作时，我们真正操作的是它的句柄。所以倘若“从一个对象到另一个对象”赋值，实际就是将句柄从一个地方复制到另一个地方。这意味着假若为对象使用“C=D”，那么 C 和 D 最终都会指向最初只有 D 才指向的那个对象。下面这个例子将向大家阐示这一点。

这里有一些题外话。在后面，大家在代码示例里看到的第一个语句将是“package 03”使用的“package”语句，它代表本书第 3 章。本书每一章的第一个代码清单都会包含象这样的一个“package”（封装、打包、包裹）语句，它的作用是为那一章剩余的代码建立章节编号。在第 17 章，大家会看到第 3 章的所有代码清单（除那些有不同封装名称的以外）都会自动置入一个名为 c03 的子目录里；第 4 章的代码置入 c04；以此类推。所有这些都是通过第 17 章展示的 CodePackage.java 程序实现的;“封装”的基本概念会在第 5 章进行详尽的解释。就目前来说，大家只需记住象“package 03”这样的形式只是用于为某一章的代码清单建立相应的子目录。

为运行程序，必须保证在 classpath 里包含了我们安装本书源码文件的根目录（那个目录里包含了 c02，c03c，c04 等等子目录）。

对于 Java 后续的版本（1.1.4 和更高版本），如果您的 main()用 package 语句封装到一个文件里，那么必须在程序名前面指定完整的包裹名称，否则不能运行程序。在这种情况下，命令行是：

```
java c03.Assignment
```

运行位于一个“包裹”里的程序时，随时都要注意这方面的问题。

下面是例子：

```
//: c03:Assignment.java
// Assignment with objects is a bit tricky.
```

```
class Number {
  int i;
}

public class Assignment {
  public static void main(String[] args) {
    Number n1 = new Number();
    Number n2 = new Number();
    n1.i = 9;
    n2.i = 47;
    System.out.println("1: n1.i: " + n1.i +
      ", n2.i: " + n2.i);
    n1 = n2;
    System.out.println("2: n1.i: " + n1.i +
      ", n2.i: " + n2.i);
    n1.i = 27;
    System.out.println("3: n1.i: " + n1.i +
      ", n2.i: " + n2.i);
  }
} ///:~
```

97-98 页程序

Number 类非常简单，它的两个实例（n1 和 n2）是在 main()里创建的。每个 Number 中的 i 值都赋予了一个不同的值。随后，将 n2 赋给 n1，而且 n1 发生改变。在许多程序设计语言中，我们都希望 n1 和 n2 任何时候都相互独立。但由于我们已赋予了一个句柄，所以下面才是真实的输出：

```
1: n1.i: 9, n2.i: 47
2: n1.i: 47, n2.i: 47
3: n1.i: 27, n2.i: 27
```

看来改变 n1 的同时也改变了 n2！这是由于无论 n1 还是 n2 都包含了相同的句柄，它指向相同的对象（最初的句柄位于 n1 内部，指向容纳了值 9 的一个对象。在赋值过程中，那个句柄实际已经丢失；它的对象会由“垃圾收集器”自动清除）。

这种特殊的现象通常也叫作“别名”，是 Java 操作对象的一种基本方式。但假若不愿意在这种情况下出现别名，又该怎么操作呢？可放弃赋值，并写入下述代码：

```
n1.i = n2.i;
```

这样便可保留两个独立的对象，而不是将 n1 和 n2 绑定到相同的对象。但您很快就会意识到，这样做会使对象内部的字段处理发生混乱，并与标准的面向对象设计准则相悖。由于这并非一个简单的话题，所以留待第 12 章详细论述，那一章是专门讨论别名的。其时，大家也会注意到对象的赋值会产生一些令人震惊的效果。

1. 方法调用中的别名处理

将一个对象传递到方法内部时，也会产生别名现象。

```
//: c03:PassObject.java
// Passing objects to methods may not be what
// you're used to.

class Letter {
  char c;
}

public class PassObject {
  static void f(Letter y) {
    y.c = 'z';
  }
  public static void main(String[] args) {
    Letter x = new Letter();
    x.c = 'a';
    System.out.println("1: x.c: " + x.c);
    f(x);
    System.out.println("2: x.c: " + x.c);
  }
} ///:~
```

99 页上程序

在许多程序设计语言中，f()方法表面上似乎要在方法的作用域内制作自己的自变量 Letter y 的一个副本。但同样地，实际传递的是一个句柄。所以下面这个程序行：

y.c = 'z';

实际改变的是 f()之外的对象。输出结果如下：

1: x.c: a

2: x.c: z

别名和它的对策是非常复杂的一个问题。尽管必须等至第 12 章才可获得所有答案，但从现在开始就应加以重视，以便提早发现它的缺点。

3.1.3 算术运算符

Java 的基本算术运算符与其他大多数程序设计语言是相同的。其中包括加号（+）、减号（-）、除号（/）、乘号（*）以及模数（%，从整数除法中获得余数）。整数除法会直接砍掉小数，而不是进位。

Java 也用一种简写形式进行运算，并同时进行赋值操作。这是由等号前的一个运算符标记的，而且对于语言中的所有运算符都是固定的。例如，为了将 4 加到变量 x，并将结果赋给 x，可用：x+=4。

下面这个例子展示了算术运算符的各种用法：

```
//: c03:MathOps.java
```

```
// Demonstrates the mathematical operators.
import java.util.*;
public class MathOps {
// Create a shorthand to save typing:
  static void prt(String s) {
System.out.println(s);
  }
  // shorthand to print a string and an int:
  static void pInt(String s, int i) {
prt(s + " = " + i);
  }
  // shorthand to print a string and a float:
  static void pFlt(String s, float f) {    prt(s + " = " + f);  }
  public static void main(String[] args) {
    // Create a random number generator,
// seeds with current time by default:
    Random rand = new Random();
    int i, j, k;    // '%' limits maximum value to 99:
j = rand.nextInt() % 100;
k = rand.nextInt() % 100;
pInt("j",j);
pInt("k",k);
i = j + k;
pInt("j + k", i);
i = j - k;
pInt("j - k", i);
i = k / j;
pInt("k / j", i);
i = k * j;
pInt("k * j", i);
i = k % j;
pInt("k % j", i);
j %= k;
pInt("j %= k", j);    // Floating-point number tests:
float u,v,w;  // applies to doubles, too
v = rand.nextFloat();
w = rand.nextFloat();
pFlt("v", v); pFlt("w", w);
u = v + w; pFlt("v + w", u);
u = v - w; pFlt("v - w", u);
u = v * w; pFlt("v * w", u);
u = v / w; pFlt("v / w", u);
    // the following also works for    // char, byte, short, int,
long,
```

```
// and double:
u += v; pFlt("u += v", u);
u -= v; pFlt("u -= v", u);
u *= v; pFlt("u *= v", u);
u /= v; pFlt("u /= v", u);
  }
} ///:~
```

100-101 页程序

我们注意到的第一件事情就是用于打印（显示）的一些快捷方法：prt()方法打印一个 String；pInt()先打印一个 String，再打印一个 int；而 pFlt()先打印一个 String，再打印一个 float。当然，它们最终都要用 System.out.println()结尾。

为生成数字，程序首先会创建一个 Random（随机）对象。由于自变量是在创建过程中传递的，所以 Java 将当前时间作为一个“种子值”，由随机数生成器利用。通过 Random 对象，程序可生成许多不同类型的随机数字。做法很简单，只需调用不同的方法即可：nextInt()，nextLong()，nextFloat()或者 nextDouble()。

若随同随机数生成器的结果使用，模数运算符（%）可将结果限制到运算对象减 1 的上限（本例是 99）之下。

1. 一元加、减运算符

一元减号（-）和一元加号（+）与二元加号和减号都是相同的运算符。根据表达式的书写形式，编译器会自动判断使用哪一种。例如下述语句：

```
x = -a;
```

它的含义是显然的。编译器能正确识别下述语句：

```
x = a * -b;
```

但读者会被搞糊涂，所以最好更明确地写成：

```
x = a * (-b);
```

一元减号得到的运算对象的负值。一元加号的含义与一元减号相反，虽然它实际并不做任何事情。

3.1.4 自动递增和递减

和 C 类似，Java 提供了丰富的快捷运算方式。这些快捷运算可使代码更清爽，更易录入，也更易读者辨读。

两种很不错的快捷运算方式是递增和递减运算符（常称作“自动递增”和“自动递减”运算符）。其中，递减运算符是“--”，意为“减少一个单位”；递增运算符是“++”，意为“增加一个单位”。举个例子来说，假设 A 是一个 int（整数）值，则表达式++A 就等价于（A = A + 1）。递增和递减运算符结果生成的是变量的值。

对每种类型的运算符，都有两个版本可供选用；通常将其称为“前缀版”和“后缀版”。“前递增”表示++运算符位于变量或表达式的前面；而“后递增”表示++运算符位于变量或表达式的后面。类似地，“前递减”意味着--运算符位于变量或表达式的前面；而“后递减”意味着--运算符位于变量或表达式的后面。

对于前递增和前递减（如++A 或--A），会先执行运算，再生成值。而对于后递增和后递减（如 A++或 A--），会先生成值，再执行运算。下面是一个例子：

102-103 页程序

```
//: c03:AutoInc.java
// Demonstrates the ++ and -- operators.

public class AutoInc {
  public static void main(String[] args) {
    int i = 1;
    prt("i : " + i);
    prt("++i : " + ++i); // Pre-increment
    prt("i++ : " + i++); // Post-increment
    prt("i : " + i);
    prt("--i : " + --i); // Pre-decrement
    prt("i-- : " + i--); // Post-decrement
    prt("i : " + i);
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

该程序的输出如下：

```
i : 1
++i : 2
i++ : 2
i : 3
--i : 2
i-- : 2
i : 1
```

103 页下程序

从中可以看到，对于前缀形式，我们在执行完运算后才得到值。但对于后缀形式，则是在运算执行之前就得到值。它们是唯一具有“副作用”的运算符（除那些涉及赋值的以外）。也就是说，它们会改变运算对象，而不仅仅是使用自己的值。

递增运算符正是对“C++”这个名字的一种解释，暗示着“超载 C 的一步”。在早期的一次 Java 演讲中，Bill Joy（始创人之一）声称“Java=C++--”（C 加加减减），意味着 Java 已去除了 C++一些没来由折磨人的地方，形成一种更精简的语言。正如大家会在这本书中学到的那样，Java 的许多地方都得到了简化，所以 Java 的学习比 C++更容易。

3.1.5 关系运算符

关系运算符生成的是一个“布尔”（Boolean）结果。它们评价的是运算对象值之间的关系。若关系是真实的，关系表达式会生成 true（真）；若关系不真实，则生成 false（假）。关系运算符包括小于（<）、大于（>）、小于或等于（<=）、大于或等于（>=）、等于（==）以及不等于（!=）。等于和不等于适用于所有内建的数据类型，但其他比较不适用于 boolean 类型。

1. 检查对象是否相等

关系运算符==和!=也适用于所有对象，但它们的含义通常会使初涉 Java 领域的人找不到北。下面是一个例子：

104 页上程序

```
//: c03:Equivalence.java

public class Equivalence {
  public static void main(String[] args) {
    Integer n1 = new Integer(47);
    Integer n2 = new Integer(47);
    System.out.println(n1 == n2);
    System.out.println(n1 != n2);
  }
} ///:~
```

其中，表达式 System.out.println(n1 == n2)可打印出内部的布尔比较结果。一般人都会认为输出结果肯定先是 true，再是 false，因为两个 Integer 对象都是相同的。但尽管对象的内容相同，句柄却是不同的，而==和!=比较的正好就是对象句柄。所以输出结果实际上先是 false，再是 true。这自然会使第一次接触的人感到惊奇。

若想对比两个对象的实际内容是否相同，又该如何操作呢？此时，必须使用所有对象都适用的特殊方法 equals()。但这个方法不适用于“主类型”，那些类型直接使用==和!=即可。下面举例说明如何使用：

104 页下程序

```
//: c03:EqualsMethod.java

public class EqualsMethod {
  public static void main(String[] args) {
    Integer n1 = new Integer(47);
    Integer n2 = new Integer(47);
    System.out.println(n1.equals(n2));
  }
} ///:~
```

正如我们预计的那样，此时得到的结果是 true。但事情并未到此结束！假设您创建了自己的类，就象下面这样：

104-105 页程序

```
//: c03:EqualsMethod2.java

class Value {
  int i;
}

public class EqualsMethod2 {
  public static void main(String[] args) {
    Value v1 = new Value();
    Value v2 = new Value();
    v1.i = v2.i = 100;
    System.out.println(v1.equals(v2));
  }
} ///:~
```

此时的结果又变回了 false！这是由于 equals()的默认行为是比较句柄。所以除非在自己的新类中改变了 equals()，否则不可能表现出我们希望的行为。不幸的是，要到第 7 章才会学习如何改变行为。但要注意 equals()的这种行为方式同时或许能够避免一些“灾难”性的事件。

大多数 Java 类库都实现了 equals()，所以它实际比较的是对象的内容，而非它们的句柄。

3.1.6 逻辑运算符

逻辑运算符 AND（&&）、OR（||）以及 NOT（!）能生成一个布尔值（true 或 false）——以自变量的逻辑关系为基础。下面这个例子向大家展示了如何使用关系和逻辑运算符。

105-106 页程序

```
import java.util.*;
public class Bool {
public static void main(String[] args) {
Random rand = new Random();
int i = rand.nextInt() % 100;
int j = rand.nextInt() % 100;
prt("i = " + i);
prt("j = " + j);
prt("i > j is " + (i > j));
prt("i < j is " + (i < j));
prt("i >= j is " + (i >= j));
prt("i <= j is " + (i <= j));
```

```
prt("i == j is " + (i == j));
prt("i != j is " + (i != j));
// Treating an int as a boolean is
// not legal Java
//! prt("i && j is " + (i && j));
//! prt("i || j is " + (i || j));
//! prt("!i is " + !i);
prt("(i < 10) && (j < 10) is "
+ ((i < 10) && (j < 10)) );
prt("(i < 10) || (j < 10) is "
+ ((i < 10) || (j < 10)) );
}
static void prt(String s) {
System.out.println(s);
}
}
```

只可将 AND，OR 或 NOT 应用于布尔值。与在 C 及 C++中不同，不可将一个非布尔值当作布尔值在逻辑表达式中使用。若这样做，就会发现尝试失败，并用一个“//!”标出。然而，后续的表达式利用关系比较生成布尔值，然后对结果进行逻辑运算。

输出列表看起来象下面这个样子：

106 页下程序

```
//: c03:Bool.java
// Relational and logical operators.
import java.util.*;

public class Bool {
  public static void main(String[] args) {
    Random rand = new Random();
    int i = rand.nextInt() % 100;
    int j = rand.nextInt() % 100;
    prt("i = " + i);
    prt("j = " + j);
    prt("i > j is " + (i > j));
    prt("i < j is " + (i < j));
    prt("i >= j is " + (i >= j));
    prt("i <= j is " + (i <= j));
    prt("i == j is " + (i == j));
    prt("i != j is " + (i != j));

    // Treating an int as a boolean is
    // not legal Java
//! prt("i && j is " + (i && j));
```

```
//! prt("i || j is " + (i || j));
//! prt("!i is " + !i);

    prt("(i < 10) && (j < 10) is "
       + ((i < 10) && (j < 10)) );
    prt("(i < 10) || (j < 10) is "
       + ((i < 10) || (j < 10)) );
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
i = 85
j = 4
i > j is true
i < j is false
i >= j is true
i <= j is false
i == j is false
i != j is true
(i < 10) && (j < 10) is false
(i < 10) || (j < 10) is true
```

注意若在预计为 String 值的地方使用，布尔值会自动转换成适当的文本形式。

在上述程序中，可将对 int 的定义替换成除 boolean 以外的其他任何主数据类型。但要注意，对浮点数字的比较是非常严格的。即使一个数字仅在小数部分与另一个数字存在极微小的差异，仍然认为它们是“不相等”的。即使一个数字只比零大一点点（例如 2 不停地开平方根），它仍然属于“非零”值。

1. 短路

操作逻辑运算符时，我们会遇到一种名为“短路”的情况。这意味着只有明确得出整个表达式真或假的结论，才会对表达式进行逻辑求值。因此，一个逻辑表达式的所有部分都有可能不进行求值：

107 页程序

```
//: c03:ShortCircuit.java
// Demonstrates short-circuiting behavior.
// with logical operators.
public class ShortCircuit {
  static boolean test1(int val) {
    System.out.println("test1(" + val + ")");
System.out.println("result: " + (val < 1));
return val < 1;
}
```

```
  static boolean test2(int val) {
    System.out.println("test2(" + val + ")");
System.out.println("result: " + (val < 2));
return val < 2;  }
  static boolean test3(int val) {
    System.out.println("test3(" + val + ")");
System.out.println("result: " + (val < 3));
return val < 3;
}
  public static void main(String[] args) {
    if(test1(0) && test2(2) && test3(2))
      System.out.println("expression is true");
    else
      System.out.println("expression is false");
   }} ///:~
```

每次测试都会比较自变量，并返回真或假。它不会显示与准备调用什么有关的资料。测试在下面这个表达式中进行：

```
if(test1(0)) && test2(2) && test3(2))
```

很自然地，你也许认为所有这三个测试都会得以执行。但希望输出结果不至于使你大吃一惊：

```
108 页程序
test1(0)
result: true
test2(2)
result: false
expression is false
```

第一个测试生成一个 true 结果，所以表达式求值会继续下去。然而，第二个测试产生了一个 false 结果。由于这意味着整个表达式肯定为 false，所以为什么还要继续剩余的表达式呢？这样做只会徒劳无益。事实上，“短路”一词的由来正种因于此。如果一个逻辑表达式的所有部分都不必执行下去，那么潜在的性能提升将是相当可观的。

3.1.7 按位运算符

按位运算符允许我们操作一个整数主数据类型中的单个“比特”，即二进制位。按位运算符会对两个自变量中对应的位执行布尔代数，并最终生成一个结果。按位运算来源于 C 语言的低级操作。我们经常都要直接操纵硬件，需要频繁设置硬件寄存器内的二进制位。Java 的设计初衷是嵌入电视顶置盒内，所以这种低级操作仍被保留下来了。然而，由于操作系统的进步，现在也许不必过于频繁地进行按位运算。

若两个输入位都是 1，则按位 AND 运算符（&）在输出位里生成一个 1；否则生成 0。若两个输入位里至少有一个是 1，则按位 OR 运算符（|）在输出位里生成一个 1；只有在两个输入位都是 0 的情况下，它才会生成一个 0。若两个输入位的某一个是 1，但不全都是 1，那么按位 XOR（^，异或）在输出位里生成一个

1。按位 NOT（~，也叫作“非”运算符）属于一元运算符；它只对一个自变量进行操作（其他所有运算符都是二元运算符）。按位 NOT 生成与输入位的相反的值——若输入 0，则输出 1；输入 1，则输出 0。

按位运算符和逻辑运算符都使用了同样的字符，只是数量不同。因此，我们能方便地记忆各自的含义：由于“位”是非常“小”的，所以按位运算符仅使用了一个字符。

按位运算符可与等号（=）联合使用，以便合并运算及赋值：&=，|=和^=都是合法的（由于~是一元运算符，所以不可与=联合使用）。

我们将 boolean（布尔）类型当作一种“单位”或“单比特”值对待，所以它多少有些独特的地方。我们可执行按位 AND，OR 和 XOR，但不能执行按位 NOT（大概是为了避免与逻辑 NOT 混淆）。对于布尔值，按位运算符具有与逻辑运算符相同的效果，只是它们不会中途“短路”。此外，针对布尔值进行的按位运算为我们新增了一个 XOR 逻辑运算符，它并未包括在“逻辑”运算符的列表中。在移位表达式中，我们被禁止使用布尔运算，原因将在下面解释。

3.1.8 移位运算符

移位运算符面向的运算对象也是二进制的“位”。可单独用它们处理整数类型（主类型的一种）。左移位运算符（<<）能将运算符左边的运算对象向左移动运算符右侧指定的位数（在低位补 0）。“有符号”右移位运算符（>>）则将运算符左边的运算对象向右移动运算符右侧指定的位数。“有符号”右移位运算符使用了“符号扩展”：若值为正，则在高位插入 0；若值为负，则在高位插入 1。Java 也添加了一种“无符号”右移位运算符（>>>），它使用了“零扩展”：无论正负，都在高位插入 0。这一运算符是 C 或 C++没有的。

若对 char，byte 或者 short 进行移位处理，那么在移位进行之前，它们会自动转换成一个 int。只有右侧的 5 个低位才会用到。这样可防止我们在一个 int 数里移动不切实际的位数。若对一个 long 值进行处理，最后得到的结果也是 long。此时只会用到右侧的 6 个低位，防止移动超过 long 值里现成的位数。但在进行“无符号”右移位时，也可能遇到一个问题。若对 byte 或 short 值进行右移位运算，得到的可能不是正确的结果（Java 1.0 和 Java 1.1 特别突出）。它们会自动转换成 int 类型，并进行右移位。但“零扩展”不会发生，所以在那些情况下会得到-1 的结果。可用下面这个例子检测自己的实现方案：

109-110 页程序

```
//: c03:URShift.java
// Test of unsigned right shift.

public class URShift {
  public static void main(String[] args) {
    int i = -1;
    i >>>= 10;
    System.out.println(i);
    long l = -1;
    l >>>= 10;
    System.out.println(l);
```

```
    short s = -1;
    s >>>= 10;
    System.out.println(s);
    byte b = -1;
    b >>>= 10;
    System.out.println(b);
  }
} ///:~
```

移位可与等号（<<=或>>=或>>>=）组合使用。此时，运算符左边的值会移动由右边的值指定的位数，再将得到的结果赋回左边的值。

下面这个例子向大家阐示了如何应用涉及“按位”操作的所有运算符，以及它们的效果：

110-112 页程序

```
//: c03:BitManipulation.java
// Using the bitwise operators.
import java.util.*;

public class BitManipulation {
  public static void main(String[] args) {
    Random rand = new Random();
    int i = rand.nextInt();
    int j = rand.nextInt();
    pBinInt("-1", -1);
    pBinInt("+1", +1);
    int maxpos = 2147483647;
    pBinInt("maxpos", maxpos);
    int maxneg = -2147483648;
    pBinInt("maxneg", maxneg);
    pBinInt("i", i);
    pBinInt("~i", ~i);
    pBinInt("-i", -i);
    pBinInt("j", j);
    pBinInt("i & j", i & j);
    pBinInt("i | j", i | j);
    pBinInt("i ^ j", i ^ j);
    pBinInt("i << 5", i << 5);
    pBinInt("i >> 5", i >> 5);
    pBinInt("(~i) >> 5", (~i) >> 5);
    pBinInt("i >>> 5", i >>> 5);
    pBinInt("(~i) >>> 5", (~i) >>> 5);

    long l = rand.nextLong();
```

```
    long m = rand.nextLong();
    pBinLong("-1L", -1L);
    pBinLong("+1L", +1L);
    long ll = 9223372036854775807L;
    pBinLong("maxpos", ll);
    long lln = -9223372036854775808L;
    pBinLong("maxneg", lln);
    pBinLong("l", l);
    pBinLong("~l", ~l);
    pBinLong("-l", -l);
    pBinLong("m", m);
    pBinLong("l & m", l & m);
    pBinLong("l | m", l | m);
    pBinLong("l ^ m", l ^ m);
    pBinLong("l << 5", l << 5);
    pBinLong("l >> 5", l >> 5);
    pBinLong("(~l) >> 5", (~l) >> 5);
    pBinLong("l >>> 5", l >>> 5);
    pBinLong("(~l) >>> 5", (~l) >>> 5);
  }
  static void pBinInt(String s, int i) {
    System.out.println(
      s + ", int: " + i + ", binary: ");
    System.out.print("   ");
    for(int j = 31; j >=0; j--)
      if(((1 << j) &  i) != 0)
        System.out.print("1");
      else
        System.out.print("0");
    System.out.println();
  }
  static void pBinLong(String s, long l) {
    System.out.println(
      s + ", long: " + l + ", binary: ");
    System.out.print("   ");
    for(int i = 63; i >=0; i--)
      if(((1L << i) & l) != 0)
        System.out.print("1");
      else
        System.out.print("0");
    System.out.println();
  }
} ///:~
```

程序末尾调用了两个方法：pBinInt()和 pBinLong()。它们分别操作一个 int 和 long 值，并用一种二进制格式输出，同时附有简要的说明文字。目前，可暂时忽略它们具体的实现方案。

大家要注意的是 System.out.print() 的使用，而不是 System.out.println()。print()方法不会产生一个新行，以便在同一行里罗列多种信息。

除展示所有按位运算符针对 int 和 long 的效果之外，本例也展示了 int 和 long 的最小值、最大值、+1 和-1 值，使大家能体会它们的情况。注意高位代表正负号：0 为正，1 为负。下面列出 int 部分的输出：

112-113 页程序

```
-1, int: -1, binary:
   11111111111111111111111111111111
+1, int: 1, binary:
   00000000000000000000000000000001
maxpos, int: 2147483647, binary:
   01111111111111111111111111111111
maxneg, int: -2147483648, binary:
   10000000000000000000000000000000
i, int: 59081716, binary:
   00000011100001011000001111110100
~i, int: -59081717, binary:
   11111100011110100111110000001011
-i, int: -59081716, binary:
   11111100011110100111110000001100
j, int: 198850956, binary:
   00001011110110100011100110001100
i & j, int: 58720644, binary:
   00000011100000000000000110000100
i | j, int: 199212028, binary:
   00001011110111111011101111111100
i ^ j, int: 140491384, binary:
   00001000010111111011101001111000
i << 5, int: 1890614912, binary:
   01110000101100000111111010000000
i >> 5, int: 1846303, binary:
   00000000000111000010110000011111
(~i) >> 5, int: -1846304, binary:
   11111111111000111101001111100000
i >>> 5, int: 1846303, binary:
   00000000000111000010110000011111
(~i) >>> 5, int: 132371424, binary:
   00000111111000111101001111100000
```

数字的二进制形式表现为“有符号 2 的补值”。

3.1.9 三元 if-else 运算符

这种运算符比较罕见，因为它有三个运算对象。但它确实属于运算符的一种，因为它最终也会生成一个值。这与本章后一节要讲述的普通 if-else 语句是不同的。表达式采取下述形式：

布尔表达式 ？ 值 0:值 1

若“布尔表达式”的结果为 true，就计算“值 0”，而且它的结果成为最终由运算符产生的值。但若“布尔表达式”的结果为 false，计算的就是“值 1”，而且它的结果成为最终由运算符产生的值。

当然，也可以换用普通的 if-else 语句（在后面介绍），但三元运算符更加简洁。尽管 C 引以为傲的就是它是一种简练的语言，而且三元运算符的引入多半就是为了体现这种高效率的编程，但假若您打算频繁用它，还是要先多作一些思量——它很容易就会产生可读性极差的代码。

可将条件运算符用于自己的“副作用”，或用于它生成的值。但通常都应将其用于值，因为那样做可将运算符与 if-else 明确区别开。下面便是一个例子：

```
static int ternary(int i) {
return i < 10 ? i * 100 : i * 10;
}
```

可以看出，假设用普通的 if-else 结构写上述代码，代码量会比上面多出许多。如下所示：

```
static int alternative(int i) {
if (i < 10)
return i * 100;
else
return i * 10;
}
```

但第二种形式更易理解，而且不要求更多的录入。所以在挑选三元运算符时，请务必权衡一下利弊。

3.1.10 逗号运算符

在 C 和 C++里，逗号不仅作为函数自变量列表的分隔符使用，也作为进行后续计算的一个运算符使用。在 Java 里需要用到逗号的唯一场所就是 for 循环，本章稍后会对此详加解释。

3.1.11 字串运算符+

这个运算符在 Java 里有一项特殊用途：连接不同的字串。这一点已在前面的例子中展示过了。尽管与+的传统意义不符，但用+来做这件事情仍然是非常自

然的。在 C++里，这一功能看起来非常不错，所以引入了一项“运算符过载”机制，以便 C++程序员为几乎所有运算符增加特殊的含义。但非常不幸，与 C++的另外一些限制结合，运算符过载成为一种非常复杂的特性，程序员在设计自己的类时必须对此有周到的考虑。与 C++相比，尽管运算符过载在 Java 里更易实现，但迄今为止仍然认为这一特性过于复杂。所以 Java 程序员不能象 C++程序员那样设计自己的过载运算符。

我们注意到运用“String +”时一些有趣的现象。若表达式以一个 String 起头，那么后续所有运算对象都必须是字串。如下所示：

```
int x = 0, y = 1, z = 2;
String sString = "x, y, z ";
System.out.println(sString + x + y + z);
```

在这里，Java 编译程序会将 x，y 和 z 转换成它们的字串形式，而不是先把它们加到一起。然而，如果使用下述语句：

```
System.out.println(x + sString);
```

那么早期版本的 Java 就会提示出错（以后的版本能将 x 转换成一个字串）。因此，如果想通过“加号”连接字串（使用 Java 的早期版本），请务必保证第一个元素是字串（或加上引号的一系列字符，编译能将其识别成一个字串）。

3.1.12 运算符常规操作规则

使用运算符的一个缺点是括号的运用经常容易搞错。即使对一个表达式如何计算有丝毫不确定的因素，都容易混淆括号的用法。这个问题在 Java 里仍然存在。

在 C 和 C++中，一个特别常见的错误如下：

```
while(x = y) {
//...
}
```

程序的意图是测试是否“相等”(==)，而不是进行赋值操作。在 C 和 C++中，若 y 是一个非零值，那么这种赋值的结果肯定是 true。这样使可能得到一个无限循环。在 Java 里，这个表达式的结果并不是布尔值，而编译器期望的是一个布尔值，而且不会从一个 int 数值中转换得来。所以在编译时，系统就会提示出现错误，有效地阻止我们进一步运行程序。所以这个缺点在 Java 里永远不会造成更严重的后果。唯一不会得到编译错误的时候是 x 和 y 都为布尔值。在这种情况下，x = y 属于合法表达式。而在上述情况下，则可能是一个错误。

在 C 和 C++里，类似的一个问题是使用按位 AND 和 OR，而不是逻辑 AND 和 OR。按位 AND 和 OR 使用两个字符之一（&或|），而逻辑 AND 和 OR 使用两个相同的字符（&&或||）。就象“=”和“==”一样，键入一个字符当然要比键入两个简单。在 Java 里，编译器同样可防止这一点，因为它不允许我们强行使用一种并不属于的类型。

3.1.13 造型运算符

“造型”（Cast）的作用是“与一个模型匹配”。在适当的时候，Java 会将一种数据类型自动转换成另一种。例如，假设我们为浮点变量分配一个整数值，计算机会将 int 自动转换成 float。通过造型，我们可明确设置这种类型的转换，或者在一般没有可能进行的时候强迫它进行。

为进行一次造型，要将括号中希望的数据类型（包括所有修改符）置于其他任何值的左侧。下面是一个例子：

```
void casts() {
int i = 200;
long l = (long)i;
long l2 = (long)200;
}
```

正如您看到的那样，既可对一个数值进行造型处理，亦可对一个变量进行造型处理。但在这儿展示的两种情况下，造型均是多余的，因为编译器在必要的时候会自动进行 int 值到 long 值的转换。当然，仍然可以设置一个造型，提醒自己留意，也使程序更清楚。在其他情况下，造型只有在代码编译时才显出重要性。

在 C 和 C++中，造型有时会让人头痛。在 Java 里，造型则是一种比较安全的操作。但是，若进行一种名为“缩小转换”（Narrowing Conversion）的操作（也就是说，脚本是能容纳更多信息的数据类型，将其转换成容量较小的类型），此时就可能面临信息丢失的危险。此时，编译器会强迫我们进行造型，就好象说：“这可能是一件危险的事情——如果您想让我不顾一切地做，那么对不起，请明确造型。”而对于“放大转换”（Widening conversion），则不必进行明确造型，因为新类型肯定能容纳原来类型的信息，不会造成任何信息的丢失。

Java 允许我们将任何主类型“造型”为其他任何一种主类型，但布尔值（bollean）要除外，后者根本不允许进行任何造型处理。“类”不允许进行造型。为了将一种类转换成另一种，必须采用特殊的方法（字串是一种特殊的情况，本书后面会讲到将对象造型到一个类型“家族”里；例如，“橡树”可造型为“树”；反之亦然。但对于其他外来类型，如“岩石”，则不能造型为“树”）。

1. 字面值

最开始的时候，若在一个程序里插入“字面值”（Literal），编译器通常能准确知道要生成什么样的类型。但在有些时候，对于类型却是暧昧不清的。若发生这种情况，必须对编译器加以适当的“指导”。方法是用与字面值关联的字符形式加入一些额外的信息。下面这段代码向大家展示了这些字符。

116-117 页程序

```
//: c03:Literals.java

class Literals {
  char c = 0xffff; // max char hex value
  byte b = 0x7f; // max byte hex value
```

```
    short s = 0x7fff; // max short hex value
    int i1 = 0x2f; // Hexadecimal (lowercase)
    int i2 = 0X2F; // Hexadecimal (uppercase)
    int i3 = 0177; // Octal (leading zero)
    // Hex and Oct also work with long.
    long n1 = 200L; // long suffix
    long n2 = 200l; // long suffix
    long n3 = 200;
    //! long l6(200); // not allowed
    float f1 = 1;
    float f2 = 1F; // float suffix
    float f3 = 1f; // float suffix
    float f4 = 1e-45f; // 10 to the power
    float f5 = 1e+9f; // float suffix
    double d1 = 1d; // double suffix
    double d2 = 1D; // double suffix
    double d3 = 47e47d; // 10 to the power
} ///:~
```

十六进制（Base 16）——它适用于所有整数数据类型——用一个前置的 0x 或 0X 指示。并在后面跟随采用大写或小写形式的 0-9 以及 a-f。若试图将一个变量初始化成超出自身能力的一个值（无论这个值的数值形式如何），编译器就会向我们报告一条出错消息。注意在上述代码中，最大的十六进制值只会在 char，byte 以及 short 身上出现。若超出这一限制，编译器会将值自动变成一个 int，并告诉我们需要对这一次赋值进行“缩小造型”。这样一来，我们就可清楚获知自己已超载了边界。

八进制（Base 8）是用数字中的一个前置 0 以及 0-7 的数位指示的。在 C，C++或者 Java 中，对二进制数字没有相应的“字面”表示方法。

字面值后的尾随字符标志着它的类型。若为大写或小写的 L，代表 long；大写或小写的 F，代表 float；大写或小写的 D，则代表 double。

指数总是采用一种我们认为很不直观的记号方法：1.39e-47f。在科学与工程学领域，“e”代表自然对数的基数，约等于 2.718（Java 一种更精确的 double 值采用 Math.E 的形式）。它在象“1.39×e 的-47 次方”这样的指数表达式中使用，意味着“1.39×2.718 的-47 次方”。然而，自 FORTRAN 语言发明后，人们自然而然地觉得 e 代表“10 多少次幂”。这种做法显得颇为古怪，因为 FORTRAN 最初面向的是科学与工程设计领域。理所当然，它的设计者应对这样的混淆概念持谨慎态度（注释①）。但不管怎样，这种特别的表达方法在 C，C++以及现在的 Java 中顽固地保留下来了。所以倘若您习惯将 e 作为自然对数的基数使用，那么在 Java 中看到象“1.39e-47f”这样的表达式时，请转换您的思维，从程序设计的角度思考它；它真正的含义是“1.39×10 的-47 次方”。

①：John Kirkham 这样写道：“我最早于 1962 年在一部 IBM 1620 机器上使用 FORTRAN II。那时——包括 60 年代以及 70 年代的早期，FORTRAN 一直都是使用大写字母。之所以会出现这一情况，可能是由于早期的输入设备大多是老式电

传打字机，使用 5 位 Baudot 码，那种码并不具备小写能力。乘幂表达式中的‘E’也肯定是大写的，所以不会与自然对数的基数‘e’发生冲突，后者必然是小写的。‘E’这个字母的含义其实很简单，就是‘Exponential’的意思，即‘指数’或‘幂数’，代表计算系统的基数——一般都是 10。当时，八进制也在程序员中广泛使用。尽管我自己未看到它的使用，但假若我在乘幂表达式中看到一个八进制数字，就会把它认作 Base 8。我记得第一次看到用小写‘e’表示指数是在 70 年代末期。我当时也觉得它极易产生混淆。所以说，这个问题完全是自己‘潜入’FORTRAN 里去的，并非一开始就有。如果你真的想使用自然对数的基数，实际有现成的函数可供利用，但它们都是大写的。”

注意如果编译器能够正确地识别类型，就不必使用尾随字符。对于下述语句：

```
long n3 = 200;
```

它并不存在含混不清的地方，所以 200 后面的一个 L 大可省去。然而，对于下述语句：

```
float f4 = 1e-47f; //10 的幂数
```

编译器通常会将指数作为双精度数（double）处理，所以假如没有这个尾随的 f，就会收到一条出错提示，告诉我们须用一个“造型”将 double 转换成 float。

2. 转型

大家会发现假若对主数据类型执行任何算术或按位运算，只要它们“比 int 小”（即 char，byte 或者 short），那么在正式执行运算之前，那些值会自动转换成 int。这样一来，最终生成的值就是 int 类型。所以只要把一个值赋回较小的类型，就必须使用“造型”。此外，由于是将值赋回给较小的类型，所以可能出现信息丢失的情况）。通常，表达式中最大的数据类型是决定了表达式最终结果大小的那个类型。若将一个 float 值与一个 double 值相乘，结果就是 double；如将一个 int 和一个 long 值相加，则结果为 long。

3.1.14 Java 没有“sizeof”

在 C 和 C++中，sizeof()运算符能满足我们的一项特殊需要：获知为数据项目分配的字符数量。在 C 和 C++中，size()最常见的一种应用就是“移植”。不同的数据在不同的机器上可能有不同的大小，所以在进行一些对大小敏感的运算时，程序员必须对那些类型有多大做到心中有数。例如，一台计算机可用 32 位来保存整数，而另一台只用 16 位保存。显然，在第一台机器中，程序可保存更大的值。正如您可能已经想到的那样，移植是令 C 和 C++程序员颇为头痛的一个问题。

Java 不需要 sizeof()运算符来满足这方面的需要，因为所有数据类型在所有机器的大小都是相同的。我们不必考虑移植问题——Java 本身就是一种“与平台无关”的语言。

3.1.15 复习计算顺序

在我举办的一次培训班中，有人抱怨运算符的优先顺序太难记了。一名学生推荐用一句话来帮助记忆：“Ulcer Addicts Really Like C A lot”，即“溃疡患者特别喜欢（维生素）C”。

助记词 运算符类型 运算符

Ulcer（溃疡） Unary：一元 + - + + - [[其余的]]
Addicts（患者） Arithmetic(shift)；算术（和移位） * / % + - << >>
Really（特别） Relational：关系 > < >= <= == !=
Like（喜欢） Logical(bitwise)：逻辑（和按位） && || & | ^
C Conditional(ternary)：条件（三元） A>B ? X:Y
A Lot Assignment：赋值 =（以及复合赋值，如*=）

当然，对于移位和按位运算符，上表并不是完美的助记方法；但对于其他运算来说，它确实很管用。

3.1.16 运算符总结

下面这个例子向大家展示了如何随同特定的运算符使用主数据类型。从根本上说，它是同一个例子反反复复地执行，只是使用了不同的主数据类型。文件编译时不会报错，因为那些会导致错误的行已用//!变成了注释内容。

120-129 页程序

```
//: c03:AllOps.java
// Tests all the operators on all the
// primitive data types to show which
// ones are accepted by the Java compiler.

class AllOps {
  // To accept the results of a boolean test:
  void f(boolean b) {}
  void boolTest(boolean x, boolean y) {
    // Arithmetic operators:
    //! x = x * y;
    //! x = x / y;
    //! x = x % y;
    //! x = x + y;
    //! x = x - y;
    //! x++;
    //! x--;
    //! x = +y;
    //! x = -y;
    // Relational and logical:
    //! f(x > y);
    //! f(x >= y);
    //! f(x < y);
    //! f(x <= y);
    f(x == y);
    f(x != y);
```

```
  f(!y);
  x = x && y;
  x = x || y;
  // Bitwise operators:
  //! x = ~y;
  x = x & y;
  x = x | y;
  x = x ^ y;
  //! x = x << 1;
  //! x = x >> 1;
  //! x = x >>> 1;
  // Compound assignment:
  //! x += y;
  //! x -= y;
  //! x *= y;
  //! x /= y;
  //! x %= y;
  //! x <<= 1;
  //! x >>= 1;
  //! x >>>= 1;
  x &= y;
  x ^= y;
  x |= y;
  // Casting:
  //! char c = (char)x;
  //! byte B = (byte)x;
  //! short s = (short)x;
  //! int i = (int)x;
  //! long l = (long)x;
  //! float f = (float)x;
  //! double d = (double)x;
}
void charTest(char x, char y) {
  // Arithmetic operators:
  x = (char)(x * y);
  x = (char)(x / y);
  x = (char)(x % y);
  x = (char)(x + y);
  x = (char)(x - y);
  x++;
  x--;
  x = (char)+y;
  x = (char)-y;
  // Relational and logical:
```

```
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Bitwise operators:
    x= (char)~y;
    x = (char)(x & y);
    x  = (char)(x | y);
    x = (char)(x ^ y);
    x = (char)(x << 1);
    x = (char)(x >> 1);
    x = (char)(x >>> 1);
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //! boolean b = (boolean)x;
    byte B = (byte)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
  }
  void byteTest(byte x, byte y) {
    // Arithmetic operators:
    x = (byte)(x* y);
    x = (byte)(x / y);
    x = (byte)(x % y);
    x = (byte)(x + y);
```

```
    x = (byte)(x - y);
    x++;
    x--;
    x = (byte)+ y;
    x = (byte)- y;
    // Relational and logical:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Bitwise operators:
    x = (byte)~y;
    x = (byte)(x & y);
    x = (byte)(x | y);
    x = (byte)(x ^ y);
    x = (byte)(x << 1);
    x = (byte)(x >> 1);
    x = (byte)(x >>> 1);
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //! boolean b = (boolean)x;
    char c = (char)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
}
```

```
void shortTest(short x, short y) {
  // Arithmetic operators:
  x = (short)(x * y);
  x = (short)(x / y);
  x = (short)(x % y);
  x = (short)(x + y);
  x = (short)(x - y);
  x++;
  x--;
  x = (short)+y;
  x = (short)-y;
  // Relational and logical:
  f(x > y);
  f(x >= y);
  f(x < y);
  f(x <= y);
  f(x == y);
  f(x != y);
  //! f(!x);
  //! f(x && y);
  //! f(x || y);
  // Bitwise operators:
  x = (short)~y;
  x = (short)(x & y);
  x = (short)(x | y);
  x = (short)(x ^ y);
  x = (short)(x << 1);
  x = (short)(x >> 1);
  x = (short)(x >>> 1);
  // Compound assignment:
  x += y;
  x -= y;
  x *= y;
  x /= y;
  x %= y;
  x <<= 1;
  x >>= 1;
  x >>>= 1;
  x &= y;
  x ^= y;
  x |= y;
  // Casting:
  //! boolean b = (boolean)x;
  char c = (char)x;
```

```
  byte B = (byte)x;
  int i = (int)x;
  long l = (long)x;
  float f = (float)x;
  double d = (double)x;
}
void intTest(int x, int y) {
  // Arithmetic operators:
  x = x * y;
  x = x / y;
  x = x % y;
  x = x + y;
  x = x - y;
  x++;
  x--;
  x = +y;
  x = -y;
  // Relational and logical:
  f(x > y);
  f(x >= y);
  f(x < y);
  f(x <= y);
  f(x == y);
  f(x != y);
  //! f(!x);
  //! f(x && y);
  //! f(x || y);
  // Bitwise operators:
  x = ~y;
  x = x & y;
  x = x | y;
  x = x ^ y;
  x = x << 1;
  x = x >> 1;
  x = x >>> 1;
  // Compound assignment:
  x += y;
  x -= y;
  x *= y;
  x /= y;
  x %= y;
  x <<= 1;
  x >>= 1;
  x >>>= 1;
```

```
  x &= y;
  x ^= y;
  x |= y;
  // Casting:
  //! boolean b = (boolean)x;
  char c = (char)x;
  byte B = (byte)x;
  short s = (short)x;
  long l = (long)x;
  float f = (float)x;
  double d = (double)x;
}
void longTest(long x, long y) {
  // Arithmetic operators:
  x = x * y;
  x = x / y;
  x = x % y;
  x = x + y;
  x = x - y;
  x++;
  x--;
  x = +y;
  x = -y;
  // Relational and logical:
  f(x > y);
  f(x >= y);
  f(x < y);
  f(x <= y);
  f(x == y);
  f(x != y);
  //! f(!x);
  //! f(x && y);
  //! f(x || y);
  // Bitwise operators:
  x = ~y;
  x = x & y;
  x = x | y;
  x = x ^ y;
  x = x << 1;
  x = x >> 1;
  x = x >>> 1;
  // Compound assignment:
  x += y;
  x -= y;
```

```
  x *= y;
  x /= y;
  x %= y;
  x <<= 1;
  x >>= 1;
  x >>>= 1;
  x &= y;
  x ^= y;
  x |= y;
  // Casting:
  //! boolean b = (boolean)x;
  char c = (char)x;
  byte B = (byte)x;
  short s = (short)x;
  int i = (int)x;
  float f = (float)x;
  double d = (double)x;
}
void floatTest(float x, float y) {
  // Arithmetic operators:
  x = x * y;
  x = x / y;
  x = x % y;
  x = x + y;
  x = x - y;
  x++;
  x--;
  x = +y;
  x = -y;
  // Relational and logical:
  f(x > y);
  f(x >= y);
  f(x < y);
  f(x <= y);
  f(x == y);
  f(x != y);
  //! f(!x);
  //! f(x && y);
  //! f(x || y);
  // Bitwise operators:
  //! x = ~y;
  //! x = x & y;
  //! x = x | y;
  //! x = x ^ y;
```

```
  //! x = x << 1;
  //! x = x >> 1;
  //! x = x >>> 1;
  // Compound assignment:
  x += y;
  x -= y;
  x *= y;
  x /= y;
  x %= y;
  //! x <<= 1;
  //! x >>= 1;
  //! x >>>= 1;
  //! x &= y;
  //! x ^= y;
  //! x |= y;
  // Casting:
  //! boolean b = (boolean)x;
  char c = (char)x;
  byte B = (byte)x;
  short s = (short)x;
  int i = (int)x;
  long l = (long)x;
  double d = (double)x;
}
void doubleTest(double x, double y) {
  // Arithmetic operators:
  x = x * y;
  x = x / y;
  x = x % y;
  x = x + y;
  x = x - y;
  x++;
  x--;
  x = +y;
  x = -y;
  // Relational and logical:
  f(x > y);
  f(x >= y);
  f(x < y);
  f(x <= y);
  f(x == y);
  f(x != y);
  //! f(!x);
  //! f(x && y);
```

```
    //! f(x || y);
    // Bitwise operators:
    //! x = ~y;
    //! x = x & y;
    //! x = x | y;
    //! x = x ^ y;
    //! x = x << 1;
    //! x = x >> 1;
    //! x = x >>> 1;
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    //! x <<= 1;
    //! x >>= 1;
    //! x >>>= 1;
    //! x &= y;
    //! x ^= y;
    //! x |= y;
    // Casting:
    //! boolean b = (boolean)x;
    char c = (char)x;
    byte B = (byte)x;
    short s = (short)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
  }
} ///:~
```

注意布尔值（boolean）的能力非常有限。我们只能为其赋予 true 和 false 值。而且可测试它为真还是为假，但不可为它们再添加布尔值，或进行其他其他任何类型运算。

在 char，byte 和 short 中，我们可看到算术运算符的“转型”效果。对这些类型的任何一个进行算术运算，都会获得一个 int 结果。必须将其明确“造型”回原来的类型（缩小转换会造成信息的丢失），以便将值赋回那个类型。但对于 int 值，却不必进行造型处理，因为所有数据都已经属于 int 类型。然而，不要放松警惕，认为一切事情都是安全的。如果对两个足够大的 int 值执行乘法运算，结果值就会溢出。下面这个例子向大家展示了这一点：

129 页下程序

```
//: c03:Overflow.java
```

```
// Surprise! Java lets you overflow.

public class Overflow {
  public static void main(String[] args) {
    int big = 0x7fffffff; // max int value
    prt("big = " + big);
    int bigger = big * 4;
    prt("bigger = " + bigger);
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

输出结果如下：

```
big = 2147483647
bigger = -4
```

而且不会从编译器那里收到出错提示，运行时也不会出现异常反应。爪哇咖啡（Java）确实是很好的东西，但却没有“那么”好！

对于 char，byte 或者 short，混合赋值并不需要造型。即使它们执行转型操作，也会获得与直接算术运算相同的结果。而在另一方面，将造型略去可使代码显得更加简练。

大家可以看到，除 boolean 以外，任何一种主类型都可通过造型变为其他主类型。同样地，当造型成一种较小的类型时，必须留意“缩小转换”的后果。否则会在造型过程中不知不觉地丢失信息。

3.2 执行控制

Java 使用了 C 的全部控制语句，所以假期您以前用 C 或 C++编程，其中大多数都应是非常熟悉的。大多数程序化的编程语言都提供了某种形式的控制语句，这在语言间通常是共通的。在 Java 里，涉及的关键字包括 if-else、while、do-while、for 以及一个名为 switch 的选择语句。然而，Java 并不支持非常有害的 goto（它仍是解决某些特殊问题的权宜之计）。仍然可以进行象 goto 那样的跳转，但比典型的 goto 要局限多了。

3.2.1 真和假

所有条件语句都利用条件表达式的真或假来决定执行流程。条件表达式的一个例子是 A==B。它用条件运算符“==”来判断 A 值是否等于 B 值。该表达式返回 true 或 false。本章早些时候接触到的所有关系运算符都可拿来构造一个条件语句。注意 Java 不允许我们将一个数字作为布尔值使用，即使它在 C 和 C++里是允许的（**真是非零，而假是零**）。若想在一次布尔测试中使用一个非布尔值——比如在 if(a)里，那么首先必须用一个条件表达式将其转换成一个布尔值，例如 if(a!=0)。

3.2.2 if-else

if-else 语句或许是控制程序流程最基本的形式。其中的 else 是可选的，所以可按下述两种形式来使用 if：

if(布尔表达式)
语句

或者

if(布尔表达式)
语句
else
语句

条件必须产生一个布尔结果。“语句”要么是用分号结尾的一个简单语句，要么是一个复合语句——封闭在括号内的一组简单语句。在本书任何地方，只要提及“语句”这个词，就有可能包括简单或复合语句。

作为 if-else 的一个例子，下面这个 test()方法可告诉我们猜测的一个数字位于目标数字之上、之下还是相等：

131 页程序

```
static int test(int testval) {
   int result = 0;
   if(testval > target)
      result = -1;
   else if(testval < target)
           result = +1;
        else
           result = 0; // match
   return result;
}
```

最好将流程控制语句缩进排列，使读者能方便地看出起点与终点。

1. return

return 关键字有两方面的用途：指定一个方法返回什么值（假设它没有 void 返回值），并立即返回那个值。可据此改写上面的 test()方法，使其利用这些特点：

131-132 页程序

```
static int test(int testval) {
  int result = 0;
  if(testval > target)
    result = -1;
```

```
  else if(testval < target)
    result = +1;
  else
    result = 0; // match
  return result;
}
```

不必加上 else，因为方法在遇到 return 后便不再继续。

3.2.3 反复

while，do-while 和 for 控制着循环，有时将其划分为“反复语句”。除非用于控制反复的布尔表达式得到“假”的结果，否则语句会重复执行下去。while 循环的格式如下：

```
static int test2(int testval) {
  if(testval > target)
    return -1;
  if(testval < target)
    return +1;
  return 0; // match
}
```

```
while(布尔表达式)
语句
```

在循环刚开始时，会计算一次“布尔表达式”的值。而对于后来每一次额外的循环，都会在开始前重新计算一次。

下面这个简单的例子可产生随机数，直到符合特定的条件为止：

132 下程序

```
//: c03:WhileTest.java
// Demonstrates the while loop.

public class WhileTest {
  public static void main(String[] args) {
    double r = 0;
    while(r < 0.99d) {
      r = Math.random();
      System.out.println(r);
    }
  }
} ///:~
```

它用到了 Math 库里的 static（静态）方法 random()。该方法的作用是产生 0 和 1 之间（包括 0，但不包括 1）的一个 double 值。while 的条件表达式意思

是说：“一直循环下去，直到数字等于或大于 0.99”。由于它的随机性，每运行一次这个程序，都会获得大小不同的数字列表。

3.2.4 do-while

do-while 的格式如下：

```
do
语句
while(布尔表达式)
```

while 和 do-while 唯一的区别就是 do-while 肯定会至少执行一次；也就是说，至少会将其中的语句“过一遍”——即便表达式第一次便计算为 false。而在 while 循环结构中，若条件第一次就为 false，那么其中的语句根本不会执行。在实际应用中，while 比 do-while 更常用一些。

3.2.5 for

for 循环在第一次反复之前要进行初始化。随后，它会进行条件测试，而且在每一次反复的时候，进行某种形式的“步进”（Stepping）。for 循环的形式如下：

```
for(初始表达式; 布尔表达式; 步进)
语句
```

无论初始表达式，布尔表达式，还是步进，都可以置空。每次反复前，都要测试一下布尔表达式。若获得的结果是 false，就会继续执行紧跟在 for 语句后面的那行代码。在每次循环的末尾，会计算一次步进。

for 循环通常用于执行“计数”任务：

133 页下程序

```
//: c03:ListCharacters.java
// Demonstrates "for" loop by listing
// all the ASCII characters.

public class ListCharacters {
  public static void main(String[] args) {
  for( char c = 0; c < 128; c++)
    if (c != 26 )  // ANSI Clear screen
      System.out.println(
        "value: " + (int)c +
        " character: " + c);
  }
} ///:~
```

注意变量 c 是在需要用到它的时候定义的——在 for 循环的控制表达式内

部，而非在由起始花括号标记的代码块的最开头。c 的作用域是由 for 控制的表达式。

以于象 C 这样传统的程序化语言，要求所有变量都在一个块的开头定义。所以在编译器创建一个块的时候，它可以为那些变量分配空间。而在 Java 和 C++中，则可在整个块的范围内分散变量声明，在真正需要的地方才加以定义。这样便可形成更自然的编码风格，也更易理解。

可在 for 语句里定义多个变量，但它们必须具有同样的类型：

134 页上程序

```
for(int i = 0, j = 1;
    i < 10 && j != 11;
    i++, j++)
  /* body of for loop */;
```

其中，for 语句内的 int 定义同时覆盖了 i 和 j。只有 for 循环才具备在控制表达式里定义变量的能力。对于其他任何条件或循环语句，都不可采用这种方法。

1. 逗号运算符

早在第 1 章，我们已提到了逗号运算符——注意不是逗号分隔符；后者用于分隔函数的不同自变量。Java 里唯一用到逗号运算符的地方就是 for 循环的控制表达式。在控制表达式的初始化和步进控制部分，我们可使用一系列由逗号分隔的语句。而且那些语句均会独立执行。前面的例子已运用了这种能力，下面则是另一个例子：

134 页下程序

```
//: c03:CommaOperator.java
public class CommaOperator {
  public static void main(String[] args) {
    for(int i = 1, j = i + 10; i < 5;
        i++, j = i * 2) {
      System.out.println("i= " + i + " j= " + j);
    }
  }
} ///:~
```

输出如下：

```
i= 1
j= 11
i= 2
j= 4
i= 3
j= 6
i= 4
```

j= 8
135 页上程序

大家可以看到，无论在初始化还是在步进部分，语句都是顺序执行的。此外，尽管初始化部分可设置任意数量的定义，但都属于同一类型。

3.2.6 中断和继续

在任何循环语句的主体部分，亦可用 break 和 continue 控制循环的流程。其中，break 用于强行退出循环，不执行循环中剩余的语句。而 continue 则停止执行当前的反复，然后退回循环起始和，开始新的反复。

下面这个程序向大家展示了 break 和 continue 在 for 和 while 循环中的例子：

135 页下程序

```
//: c03:BreakAndContinue.java
// Demonstrates break and continue keywords.

public class BreakAndContinue {
  public static void main(String[] args) {
    for(int i = 0; i < 100; i++) {
      if(i == 74) break; // Out of for loop
      if(i % 9 != 0) continue; // Next iteration
      System.out.println(i);
    }
    int i = 0;
    // An "infinite loop":
    while(true) {
      i++;
      int j = i * 27;
      if(j == 1269) break; // Out of loop
      if(i % 10 != 0) continue; // Top of loop
      System.out.println(i);
    }
  }
} ///:~
```

在这个 for 循环中，i 的值永远不会到达 100。因为一旦 i 到达 74，break 语句就会中断循环。通常，只有在不知道中断条件何时满足时，才需象这样使用 break。只要 i 不能被 9 整除，continue 语句会使程序流程返回循环的最开头执行（所以使 i 值递增）。如果能够整除，则将值显示出来。

第二部分向大家揭示了一个“无限循环”的情况。然而，循环内部有一个 break 语句，可中止循环。除此以外，大家还会看到 continue 移回循环顶部，同时不完成剩余的内容（所以只有在 i 值能被 9 整除时才打印出值）。输出结果如下：

136 页程序
0
9
18
27
36
45
54
63
72
10
20
30
40

之所以显示 0，是由于 0%9 等于 0。

无限循环的第二种形式是 for(;;)。编译器将 while(true)与 for(;;)看作同一回事。所以具体选用哪个取决于自己的编程习惯。

1. 臭名昭著的“goto”

goto 关键字很早就在程序设计语言中出现。事实上，goto 是汇编语言的程序控制结构的始祖：“若条件 A，则跳到这里；否则跳到那里”。若阅读由几乎所有编译器生成的汇编代码，就会发现程序控制里包含了许多跳转。然而，goto 是在源码的级别跳转的，所以招致了不好的声誉。若程序总是从一个地方跳到另一个地方，还有什么办法能识别代码的流程呢？随着 Edsger Dijkstra 著名的“Goto 有害”论的问世，goto 便从此失宠。

事实上，真正的问题并不在于使用 goto，而在于 goto 的滥用。而且在一些少见的情况下，goto 是组织控制流程的最佳手段。

尽管 goto 仍是 Java 的一个保留字，但并未在语言中得到正式使用；Java 没有 goto。然而，在 break 和 continue 这两个关键字的身上，我们仍然能看出一些 goto 的影子。它并不属于一次跳转，而是中断循环语句的一种方法。之所以把它们纳入 goto 问题中一起讨论，是由于它们使用了相同的机制：标签。

“标签”是后面跟一个冒号的标识符，就象下面这样：

label1:

对 Java 来说，唯一用到标签的地方是在循环语句之前。进一步说，它实际需要紧靠在循环语句的前方——在标签和循环之间置入任何语句都是不明智的。而在循环之前设置标签的唯一理由是：我们希望在其中嵌套另一个循环或者一个开关。这是由于 break 和 continue 关键字通常只中断当前循环，但若随同标签使用，它们就会中断到存在标签的地方。如下所示：

```
label1:
外部循环{
内部循环{
//...
```

```
break; //1
//...
continue; //2
//...
continue label1; //3
//...
break label1; //4
}
}
```

在条件 1 中,break 中断内部循环,并在外部循环结束。在条件 2 中,continue 移回内部循环的起始处。但在条件 3 中，continue label1 却同时中断内部循环以及外部循环，并移至 label1 处。随后，它实际是继续循环，但却从外部循环开始。在条件 4 中，break label1 也会中断所有循环，并回到 label1 处，但并不重新进入循环。也就是说，它实际是完全中止了两个循环。

下面是 for 循环的一个例子：

138-139 页程序

```
//: c03:LabeledFor.java
// Java's "labeled for" loop.

public class LabeledFor {
  public static void main(String[] args) {
    int i = 0;
    outer: // Can't have statements here
    for(; true ;) { // infinite loop
      inner: // Can't have statements here
      for(; i < 10; i++) {
        prt("i = " + i);
        if(i == 2) {
          prt("continue");
          continue;
        }
        if(i == 3) {
          prt("break");
          i++; // Otherwise i never
               // gets incremented.
          break;
        }
        if(i == 7) {
          prt("continue outer");
          i++; // Otherwise i never
               // gets incremented.
          continue outer;
```

```
        }
        if(i == 8) {
          prt("break outer");
          break outer;
        }
        for(int k = 0; k < 5; k++) {
          if(k == 3) {
            prt("continue inner");
            continue inner;
          }
        }
      }
    }
    // Can't break or continue
    // to labels here
  }
  static void prt(String s) {
    System.out.println(s);
  }
} ///:~
```

这里用到了在其他例子中已经定义的 prt()方法。

注意 break 会中断 for 循环，而且在抵达 for 循环的末尾之前，递增表达式不会执行。由于 break 跳过了递增表达式，所以递增会在 i==3 的情况下直接执行。在 i==7 的情况下，continue outer 语句也会到达循环顶部，而且也会跳过递增，所以它也是直接递增的。

下面是输出结果：

139 页中程序

```
i = 0
continue inner
i = 1
continue inner
i = 2
continue
i = 3
break
i = 4
continue inner
i = 5
continue inner
i = 6
continue inner
i = 7
```

```
continue outer
i = 8
break outer
```

如果没有 break outer 语句，就没有办法在一个内部循环里找到出外部循环的路径。这是由于 break 本身只能中断最内层的循环(对于 continue 同样如此)。

当然，若想在中断循环的同时退出方法，简单地用一个 return 即可。

下面这个例子向大家展示了带标签的 break 以及 continue 语句在 while 循环中的用法：

140 页程序

```
//: c03:LabeledWhile.java
// Java's "labeled while" loop.
public class LabeledWhile {
     public static void main(String[] args) {
          int i = 0;
outer:    while(true) {
            prt("Outer while loop");
              while(true) {
                i++;
                prt("i = " + i);
               if(i == 1) {
                 prt("continue");
               continue;
              }
               if(i == 3) {
                  prt("continue outer");
                  continue outer;        }
            if(i   ==   5)   {                          prt("break");
break;        }
          if(i == 7) {          prt("break outer");          break
outer;        }
        }    }  }
         static void prt(String s) {    System.out.println(s);   }}
///:~
```

同样的规则亦适用于 while：

(1) 简单的一个 continue 会退回最内层循环的开头（顶部），并继续执行。

(2) 带有标签的 continue 会到达标签的位置，并重新进入紧接在那个标签后面的循环。

(3) break 会中断当前循环，并移离当前标签的末尾。

(4) 带标签的 break 会中断当前循环，并移离由那个标签指示的循环的末尾。

这个方法的输出结果是一目了然的：

141 页程序

```
Outer while loopi = 1continuei = 2i = 3continue outerOuter while loopi = 4i = 5
breakOuter while loopi = 6i = 7break outer
```

大家要记住的重点是：在 Java 里唯一需要用到标签的地方就是拥有嵌套循环，而且想中断或继续多个嵌套级别的时候。

在 Dijkstra 的“Goto 有害”论中，他最反对的就是标签，而非 goto。随着标签在一个程序里数量的增多，他发现产生错误的机会也越来越多。标签和 goto 使我们难于对程序作静态分析。这是由于它们在程序的执行流程中引入了许多“怪圈”。但幸运的是，Java 标签不会造成这方面的问题，因为它们的活动场所已被限死，不可通过特别的方式到处传递程序的控制权。由此也引出了一个有趣的问题：通过限制语句的能力，反而能使一项语言特性更加有用。

3.2.7 开关

“开关”（Switch）有时也被划分为一种“选择语句”。根据一个整数表达式的值，switch 语句可从一系列代码选出一段执行。它的格式如下：

```
switch(整数选择因子) {
case 整数值 1 : 语句; break;
case 整数值 2 : 语句; break;
case 整数值 3 : 语句; break;
case 整数值 4 : 语句; break;
case 整数值 5 : 语句; break;
//..
default:语句;
}
```

其中，“整数选择因子”是一个特殊的表达式，能产生整数值。switch 能将整数选择因子的结果与每个整数值比较。若发现相符的，就执行对应的语句（简单或复合语句）。若没有发现相符的，就执行 default 语句。

在上面的定义中，大家会注意到每个 case 均以一个 break 结尾。这样可使执行流程跳转至 switch 主体的末尾。这是构建 switch 语句的一种传统方式，但 break 是可选的。若省略 break，会继续执行后面的 case 语句的代码，直到遇到一个 break 为止。尽管通常不想出现这种情况，但对有经验的程序员来说，也许能够善加利用。注意最后的 default 语句没有 break，因为执行流程已到了 break 的跳转目的地。当然，如果考虑到编程风格方面的原因，完全可以在 default 语句的末尾放置一个 break，尽管它并没有任何实际的用处。

switch 语句是实现多路选择的一种易行方式（比如从一系列执行路径中挑选一个）。但它要求使用一个选择因子，并且必须是 int 或 char 那样的整数值。例如，假若将一个字串或者浮点数作为选择因子使用，那么它们在 switch 语句里是不会工作的。对于非整数类型，则必须使用一系列 if 语句。

下面这个例子可随机生成字母，并判断它们是元音还是辅音字母：

142-143 页程序

```
//: c03:VowelsAndConsonants.java
```

```
// Demonstrates the switch statement.

public class VowelsAndConsonants {
  public static void main(String[] args) {
    for(int i = 0; i < 100; i++) {
      char c = (char)(Math.random() * 26 + 'a');
      System.out.print(c + ": ");
      switch(c) {
      case 'a':
      case 'e':
      case 'i':
      case 'o':
      case 'u':
                System.out.println("vowel");
                break;
      case 'y':
      case 'w':
                System.out.println(
                  "Sometimes a vowel");
                break;
      default:
                System.out.println("consonant");
      }
    }
  }
} ///:~
```

由于 Math.random()会产生 0 到 1 之间的一个值，所以只需将其乘以想获得的最大随机数（对于英语字母，这个数字是 26），再加上一个偏移量，得到最小的随机数。

尽管我们在这儿表面上要处理的是字符，但 switch 语句实际使用的字符的整数值。在 case 语句中，用单引号封闭起来的字符也会产生整数值，以便我们进行比较。

请注意 case 语句相互间是如何聚合在一起的，它们依次排列，为一部分特定的代码提供了多种匹配模式。也应注意将 break 语句置于一个特定 case 的末尾，否则控制流程会简单地下移，并继续判断下一个条件是否相符。

1. 具体的计算

应特别留意下面这个语句：

```
char c = (char)(Math.random() * 26 + 'a');
```

Math.random()会产生一个 double 值，所以 26 会转换成 double 类型，以便执行乘法运算。这个运算也会产生一个 double 值。这意味着为了执行加法，必须无将'a'转换成一个 double。利用一个“造型”，double 结果会转换回 char。

我们的第一个问题是，造型会对 char 作什么样的处理呢？换言之，假设一

个值是 29.7，我们把它造型成一个 char，那么结果值到底是 30 还是 29 呢？答案可从下面这个例子中得到：

144 页上程序

```
//: c03:CastingNumbers.java
// What happens when you cast a float
// or double to an integral value?

public class CastingNumbers {
  public static void main(String[] args) {
    double
      above = 0.7,
      below = 0.4;
    System.out.println("above: " + above);
    System.out.println("below: " + below);
    System.out.println(
      "(int)above: " + (int)above);
    System.out.println(
      "(int)below: " + (int)below);
    System.out.println(
      "(char)('a' + above): " +
      (char)('a' + above));
    System.out.println(
      "(char)('a' + below): " +
      (char)('a' + below));
  }
} ///:~
```

输出结果如下：

144 页下程序

```
above: 0.7below: 0.4(int)above: 0(int)below: 0(char)('a' + above): a
(char)('a' + below): a
```

所以答案就是：将一个 float 或 double 值造型成整数值后，总是将小数部分“砍掉”，不作任何进位处理。

第二个问题与 Math.random()有关。它会产生 0 和 1 之间的值，但是否包括值'1'呢？用正统的数学语言表达，它到底是(0,1)，[0,1]，(0,1]，还是[0,1)呢（方括号表示“包括”，圆括号表示“不包括”）？同样地，一个示范程序向我们揭示了答案：

145 页程序

```
public class RandomBounds {  static void usage() {
    System.err.println("Usage:  \n\t"  +            "RandomBounds
```

```
lower\n\t" +
          "RandomBounds upper");    System.exit(1);  }
     public static void main(String[] args) {    if(args.length != 1)
usage();
       if(args[0].equals("lower")) {      while(Math.random() != 0.0)
           ;  //  Keep  trying          System.out.println("Produced
0.0!");    }
       else if(args[0].equals("upper")) {      while(Math.random() !=
1.0)
           ;  //  Keep  trying          System.out.println("Produced
1.0!");    }
       else       usage();  }} ///:~
```

为运行这个程序，只需在命令行键入下述命令即可：

```
java RandomBounds lower
```

或

```
java RandomBounds upper
```

在这两种情况下，我们都必须人工中断程序，所以会发现 Math.random()“似乎”永远都不会产生 0.0 或 1.0。但这只是一项实验而已。若想到 0 和 1 之间有 2 的 128 次方不同的双精度小数，所以如果全部产生这些数字，花费的时间会远远超过一个人的生命。当然，最后的结果是在 Math.random()的输出中包括了 0.0。或者用数字语言表达，**输出值范围是[0,1)。**

3.3 总结

本章总结了大多数程序设计语言都具有的基本特性：计算、运算符优先顺序、类型转换以及选择和循环等等。现在，我们作好了相应的准备，可继续向面向对象的程序设计领域迈进。在下一章里，我们将讨论对象的初始化与清除问题，再后面则讲述隐藏的基本实现方法。

3.4 练习

(1) 写一个程序，打印出 1 到 100 间的整数。

(2) 修改练习(1)，在值为 47 时用一个 break 退出程序。亦可换成 return 试试。

(3) 创建一个 switch 语句，为每一种 case 都显示一条消息。并将 switch 置入一个 for 循环里，令其尝试每一种 case。在每个 case 后面都放置一个 break，并对其进行测试。然后，删除 break，看看会有什么情况出现。