致读者：

我从 2002 年 7 月开始翻译这本书，当时还是第二版。但是翻完前言和介绍部分后，chinapub 就登出广告，说要出版侯捷的译本。于是我中止了翻译，等着侯先生的作品。

我是第一时间买的 这本书，但是我失望了。比起第一版，我终于能看懂这本书了，但是相比我的预期，它还是差一点。所以当 Bruce Eckel 在他的网站上公开本书的第三版的时候，我决定把它翻译出来。

说说容易，做做难。一本 1000 多页的书不是那么容易翻的。期间我也曾打过退堂鼓，但最终还是全部翻译出来了。从今年的两月初起，到 7 月底，我几乎放弃了所有的业余时间，全身心地投入本书的翻译之中。应该说，这项工作的难度超出了我的想像。

首先，读一本书和翻译一本书完全是两码事。英语与中文是两种不同的语言，用英语说得很畅的句子，翻成中文之后就完全破了相。有时我得花好几分钟，用中文重述一句我能用几秒钟读懂的句子。更何况作为读者，一两句话没搞懂，并不影响你理解整本书，但对译者来说，这就不一样了。

其次，这是一本讲英语的人写给讲英语的人的书，所以同很多要照顾非英语读者的技术文档不同，它在用词，句式方面非常随意。英语读者会很欣赏这一点，但是对外国读者来说，这就是负担了。

再有，Bruce Eckel 这样的大牛人，写了 1000 多页，如果都让你读懂，他岂不是太没面子？所以，书里还有一些很有“禅意”的句子。比如那句著名的“The genesis of the computer revolution was in a machine. The genesis of our programming languages thus tends to look like that machine.”我就一直没吃准该怎么翻译。我想大概没人能吃准，说不定 Bruce 要的就是这个效果。

这是一本公认的名著，作者在技术上的造诣无可挑剔。而作为译者，我的编程能力差了很多。再加上上面讲的这些原因，使得我不得不格外的谨慎。当我重读初稿的时候，我发现需要修改的地方实在太多了。因此，我不能现在就公开全部译稿，我只能公开已经修改过的部分。不过这不是最终的版本，我还会继续修订的。

本来，我准备到 10 月份，等我修改完前 7 章之后再公开。但是，我发现我又有点要放弃了，因此我决定给自己一点压力，现在就公开。以后，我将修改完一章就公开一章，请关注 www.wgqqh.com/shhgs/tij.html。

如果你觉得好，请给告诉我，你的鼓励是我工作的动力；如果你觉得不好，那就更应该告诉我了，我会参考你的意见作修改的。我希望能通过这种方法，译出一本配得上原著的书。

shhgs

2003 年 9 月 8 日

3: 控制程序流程

就像有感知力的生物那样，程序应该有能力操控它的世界，并且在执行过程中作决定。

Java 让你用运算符(operator)来控制数据，用执行控制语句来作决定。Java 继承了 C++，因此 C 和 C++程序员会对它的语句和运算符感到很亲切。不过 Java 也作了一些改进和简化。

如果你觉得这一章还是有点难，那么先去看看本书附送的多媒体 CD ROM：*Foundations for Java*。它里边包括了讲课的录音，幻灯片，练习以及答案。设计这个课程的目的就是要让你尽快地掌握学习 Java 所需的基础知识。

使用 Java 运算符

运算符需要一到两个参数，并且会产生一个新的值。参数的使用方法与普通方法的调用不同，但是结果是相同的。加(**+**)，减(**-**)，乘(*****)，除(**/**)，以及等号(**=**)的用法同其它编程语言完全相同。

运算符作用于操作数(operand)，并且会产生一个值。此外，运算符还被用于修改操作数的值。这被称为副作用(*side effect*)。我们经常看到的，用运算符来修改操作数的操作就是利用了这种副作用，但是你应该记住，这种做法所产生的值也是可以用的，它同那些没有副作用的运算符是一样。

几乎所有的运算符都只能作用于 primitive。但是‘=’，‘==’，以及‘**!=**’是例外，它们可以作用于任何对象(由此也是对象方面一个很让人头晕的问题。)此外，**String** 类也支持‘**+**’和‘+=’。

优先级

当一个表达式包含多个运算符的时候，运算符的优先顺序会决定该怎样计算这个表达式的值。Java 有一套判断计算顺序的特殊规则。其中最简单的一条就是“先乘除后加减”。程序员们经常会把其余的优先顺序都给忘了，所以你应该使用括号来明确的指明计算的顺序。例如：

```
a = x + y - 2/2 + z;
```

同一个表达式，加上两组括号之后就会有截然不同的意思：

```
a = x + (y - 2)/(2 + z);
```

赋值

赋值用的是 = 运算符。它的意思是“算出等号右边的值(通常称为 *rvalue*)，然后拷贝到等号的左边(通常称为 *lvalue*)”。rvalue 可以是任何常量，变量或者是能产生值的表达式，而 lvalue 则只能是一个明确的，有名字的变量(named variable)。(也就是说，必须要有一个物理空间来存储这个值。)比如，你可以将常量赋给变量：

```
a = 4;
```

但是你不能将值赋给常量——也就是说，常量是不能当 lvalue 的。(你不能说 **4 = a;**)

给 primitive 赋值还是比较简单的。由于 primitive 持有的是实实在在的值，而不是指向对象的 reference，因此为 primitive 赋值的时候，你是将内容从一个地方直接拷贝到另一个地方。假设 **a** 和 **b** 都是 primitive，如果你说 **a=b** ，那么 **b** 的内容会被拷贝到 **a**。如果接下来你又修改了 **a**，那么很显然 **b** 是不会受到影响的。对于程序员来说，在绝大多数情况下，这正是他们所需要的。

但是，给对象赋值的时候，情况就有所不同了。只要你想操控对象，你就得通过 reference 来进行操作，所以当你“在对象之间”进行赋值的时候，实际上你是在拷贝它的 reference。也就是说，如果 **c** 和 **d** 都是对象，而你说 **c = d**，结果就成了 **c** 和 **d** 都指向原先只有 **d** 指着的那个对象了。下面的例程演示了这种行为：

```
//: c03:Assignment.java
// Assignment with objects is a bit tricky.
import com.bruceeckel.simpletest.*;

class Number {
  int i;
}

public class Assignment {
  static Test monitor = new Test();
  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);
    monitor.expect(new String[] {
      "1: n1.i: 9, n2.i: 47",
      "2: n1.i: 47, n2.i: 47",
      "3: n1.i: 27, n2.i: 27"
    });
  }
} ///:~
```

首先，提请大家注意一下新加的这一行：

```
import com.bruceeckel.simpletest.*;
```

它引入了专门用于测试本书代码的“**simpletest**”类库，我们会到第 15 章再详细讲解这个类库。在 **Assignment** 类的开始部分，有这么一行：

```
  static Test monitor = new Test();
```

它会创建一个名为 **monitor** 的的 **Test** 类的实例，而 **Test** 又是 **simpletest** 的一个类。在 **main( )** 的最后， 你还会看到：

```
    monitor.expect(new String[] {
      "1: n1.i: 9, n2.i: 47",
      "2: n1.i: 47, n2.i: 47",
      "3: n1.i: 27, n2.i: 27"
    });
```

这是用 **String** 数组表示的，程序的预期输出。程序运行的时候，不但会把输出打印出来，而且会拿它同数组作比较，以检验是不是正确。于是你会发现，如果程序用到了 **simpletest**，那它肯定会调用显示程序输出的 **expect( )**。这样你就能从源代码里看到正确的输出了。

Number 类很简单。**main( )**创建了两个 **Number** 类的实例(**n1** 和 **n2**)。它们的 **i** 被分别赋上了不同的值，然后把 **n2** 赋给 **n1**，再修改 **n1**。在很多别的编程语言里，你可以设想 **n1** 和 **n2** 自始至终都是两个独立的对象。但在 Java 里面，由于是对 reference 进行赋值，因此从 **expect( )**语句可以看出，对 **n1** 的修改会影响到 **n2**！这是因为，**n1** 和 **n2** 都是 reference，而这两个 reference 又指向同一个对象。(原先那个在 **n1** 里面的 reference，指向的是保存 9 这个值的对象。在赋值过

程中，这个 reference 被覆盖了，实际上就是被扔掉了；而它所指的那个对象会被垃圾回收器清理掉。)

这种现象通常被称为 *aliasing* (别名效应)，这是 Java 操控对象的基本方法。但是如果你想避免 aliasing 又应该怎么作呢？你可以进一步对具体的数据成员进行赋值：

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

与丢掉一个对象，再将 **n1** 和 **n2** 都绑到同一个对象相比，这样就能保留两个相互独立的对象了。不过你很快就会知道直接操控对象内部的数据是很麻烦的，而且有违 OO 设计的原则。这个课题不是一两句话就能讲清楚的，所以我们把它放到附录 A 里，我们会在那里专门探讨 aliasing 的。在此期间，你只要知道为对象赋值可能会与你的初衷大相径庭就行了。

方法调用期间的 aliasing

把对象传给方法的时候，也会有 aliasing 的问题：

```
//: c03:PassObject.java
// Passing objects to methods may not be what
// you're used to.
import com.bruceeckel.simpletest.*;

class Letter {
  char c;
}

public class PassObject {
  static Test monitor = new Test();
  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);
    monitor.expect(new String[] {
      "1: x.c: a",
      "2: x.c: z"
    });
  }
} ///:~
```

在很多编程语言里，方法 **f( )**会在它的作用域里对参数 **Letter y** 作一份拷贝。但这里传的还是 reference，所以下面这行

```
y.c = 'z';
```

实际上会修改 **f( )**外面的对象。**expect( )**里面的输出可以印证这一点。

Aliasing 及其解决方案是个很复杂的课题。虽然我们要到附录 A 才会作全面讲解，但是你现在就得知道有这个问题，这样才会注意到一些“陷阱”。

数学运算符

在绝大多数编程语言里面，基本的数学运算符都是相同的：加(**+**)，减(**-**)，除(**/**)，乘(*****)，取模(**%**，它会返回整数相除之后的余数)。整数相除，会把运算结果的小数点后面全部截掉，而不是作四舍五入。

Java 也有可以同时进行计算和赋值的简写符号。它用运算符加一个等号来表示，在 Java 语言中，这种表示方法能一致地用于所有的运算符 (只要是有意义就行了)。比如，你要为变量 **x** 加 4，然后再赋给 **x**，就写：**x += 4** 。

下面这个例程演示了数学运算符的用法：

```
//: c03:MathOps.java
// Demonstrates the mathematical operators.
import com.bruceeckel.simpletest.*;
import java.util.*;

public class MathOps {
  static Test monitor = new Test();
  // Shorthand to print a string and an int:
  static void printInt(String s, int i) {
    System.out.println(s + " = " + i);
  }
  // Shorthand to print a string and a float:
  static void printFloat(String s, float f) {
    System.out.println(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;
    // Choose value from 1 to 100:
    j = rand.nextInt(100) + 1;
    k = rand.nextInt(100) + 1;
    printInt("j", j);  printInt("k", k);
    i = j + k; printInt("j + k", i);
    i = j - k; printInt("j - k", i);
    i = k / j; printInt("k / j", i);
    i = k * j; printInt("k * j", i);
    i = k % j; printInt("k % j", i);
    j %= k; printInt("j %= k", j);
```

```
    // Floating-point number tests:
    float u,v,w;  // applies to doubles, too
    v = rand.nextFloat();
    w = rand.nextFloat();
    printFloat("v", v); printFloat("w", w);
    u = v + w; printFloat("v + w", u);
    u = v - w; printFloat("v - w", u);
    u = v * w; printFloat("v * w", u);
    u = v / w; printFloat("v / w", u);
    // the following also works for
    // char, byte, short, int, long,
    // and double:
    u += v; printFloat("u += v", u);
    u -= v; printFloat("u -= v", u);
    u *= v; printFloat("u *= v", u);
    u /= v; printFloat("u /= v", u);
    monitor.expect(new String[] {
      "%% j = -?\\d+",
      "%% k = -?\\d+",
      "%% j \\+ k = -?\\d+",
      "%% j - k = -?\\d+",
      "%% k / j = -?\\d+",
      "%% k \\* j = -?\\d+",
      "%% k % j = -?\\d+",
      "%% j %= k = -?\\d+",
      "%% v = -?\\d+\\.\\d+(E-?\\d)?",
      "%% w = -?\\d+\\.\\d+(E-?\\d)?",
      "%% v \\+ w = -?\\d+\\.\\d+(E-?\\d)??",
      "%% v - w = -?\\d+\\.\\d+(E-?\\d)??",
      "%% v \\* w = -?\\d+\\.\\d+(E-?\\d)??",
      "%% v / w = -?\\d+\\.\\d+(E-?\\d)??",
      "%% u \\+= v = -?\\d+\\.\\d+(E-?\\d)??",
      "%% u -= v = -?\\d+\\.\\d+(E-?\\d)??",
      "%% u \\*= v = -?\\d+\\.\\d+(E-?\\d)??",
      "%% u /= v = -?\\d+\\.\\d+(E-?\\d)??"
    });
  }
} ///:~
```

首先看到的是两个为节省敲键盘而定义的打印方法：**printInt( )**会先打印一个 **String**，然后再接一个 **int**；而 **printFloat( )**会先打印一个 **String**，再打印一个 **float**。

程序先创建一个 **Random** 对象，以产生数字。由于创建过程中没有使用参数，因此 Java 会用当前的时间作随机数生成器的种子。程序用 **Random** 对象的 **nextInt( )**和 **nextFloat( )**方法生成了很多不同种类的随机数 (你也可以用 **nextLong( )**或 **nextDouble( )**)。

取模运算符(modulus operator)会将随机数生成器产生的数字限制在操作数减一的范围之内(这里就是 99)。

正则表达式 (Regular expressions)

由于程序用到了随机数，因此输出的结果会一次和一次不一样，所以**expect( )**就不能再像过去那样，一字不错地告诉你会输出些什么了。要解决这个问题，**expect( )**语句就得用到一种 Java JDK 1.4 所引入的，被称为正则表达式的新的特性(但是在 Perl 和 Python 之类的语言里，这已经是老特性了)。尽管我们要到第 12 章才会全面介绍这一无比强大的工具，但如果这里不作介绍的话，你就没法看下去了。目前，你只要能读懂 **expect( )**就行了，要想全面了解这部分内容，请查阅 JDK 文档的 **java.util.regex.Pattern** 部分。

正则表达式是一种用通用术语描述字符串的方法，这样你就可以说：“如果字符串里有这些东西，那么它同我正在找的东西相匹配。”比如，要表示一个可能有也可能没有的负号，你可以在负号后面加一个问号，就像这样：

```
-?
```

表示整数，就是表示一个或多个阿拉伯数字。在正则表达式中，阿拉伯数字用‘**\d**’表示的，但是在 Java 的 **String** 里，你必须再加一个反斜杠才能把它“转义”成一个反斜杠：‘**\\d**’。要在正则表达式里表示“一个或多个前述的表达式”，你就得用‘**+**’。所以要表示“前面可能有个减号，后边跟着一串阿拉伯数字”，你得写：

```
-?\\d+
```

这就是上面那段程序的 **expect( )**语句的第一行。

expect( )语句的各行中，开头部分的‘**%%**’(注意一下，空格是为了增强可读性而加的)，都不属于正则表达式。这是 **simpletest** 所使用的标记，它表示这行剩下来的部分是一个正则表达式。所以仅从 **simpletest** 的 **expect( )**语句，你是看不到普通的正则表达式的。

其它字符，只要它不属于正则表达式特有的专用字符，都要求完全匹配。所以第一行：

```
%% j = -?\\d+
```

‘j = ’会做精确匹配。但是在第三行，‘j + k’里面的‘+’必须进行转义，因为跟‘*’一样，在正则表达式里它也是一个特殊字符。作完这些介绍之后，剩下几行你就应该能看懂了。如果本书后面的 **expect( )** 里面又用到了正则表达式的新特性的话，我们会再作解释。

单元的加号和减号运算符

单元的减号(-)和加号(+)与双元减号和加号完全相同。编译器会通过的表达式来判断这个运算符到底是单元的还是双元的。例如，表达式

```
x = -a;
```

的意思就很明显。此外编译器也可以认出：

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

但是读代码的人就惨了，所以还是这样写更清楚：

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

单元减号表示负数。有单元减号就得有单元的正号，只是不起任何作用。

自动递增与递减

同 C 一样，Java 里面到处都是简写。简写可以让程序写起来简单，至于读取来怎么样，那得看情况。

递增和递减运算符(通常也被称作“自动递增 auto-increment”和“自动递减 auto-decrement”运算符) 是两种比较好的简写。递减的运算符是 **--** ，它的意思是“减小一个单位”。而递增得运算符是 **++** ，它的意思是“增加一个单位”。比方说 **a** 是一个 **int**，那么**++a** 实际上同(**a = a +1**)是等效的。递增和递减不但修改变量的值，而且会把这个值返回给变量。

这两个运算符还各有两个版本，也就是常提到的前置(prefix)和后置(postfix)。“前置递增 (*pre-increment*)”的意思是 **++** 运算符出现在变量或者表达式的前面，而“后置递增 (*post-increment*)”的意思是 **++** 运算符出现在变量或者表达式的后面。同样，“前置递减(*pre-decrement*)”是指 **--** 运算符出现在变量或者表达式的前面，“后置递减(*post-decrement*)”的意思是 **--** 运算符出现在变量或者表达式的后面。对于前置递增和前置递减，(也就是 **++a** 和 **--a**)，操作过程是会先计算再返回。对于后置递增和后置递减，(也就是 **a++** 和 **a--**)，操作步骤是先返回再计算。下面就是举例：

```
//: c03:AutoInc.java
// Demonstrates the ++ and -- operators.
import com.bruceeckel.simpletest.*;
```

```
public class AutoInc {
  static Test monitor = new Test();
  public static void main(String[] args) {
    int i = 1;
    System.out.println("i : " + i);
    System.out.println("++i : " + ++i); // Pre-
increment
    System.out.println("i++ : " + i++); // Post-
increment
    System.out.println("i : " + i);
    System.out.println("--i : " + --i); // Pre-
decrement
    System.out.println("i-- : " + i--); // Post-
decrement
    System.out.println("i : " + i);
    monitor.expect(new String[] {
      "i : 1",
      "++i : 2",
      "i++ : 2",
      "i : 3",
      "--i : 2",
      "i-- : 2",
      "i : 1"
    });
  }
} ///:~
```

你可以看到，运算符前置的时候，你所得到的是经过计算的值，但是运算符后置的情况下，你所得到的是还未进行过处理的值。这是唯一一种 (除了赋值之外)还有其它附带用途的操作符。(也就是说，不是要用它来进行计算，而是要用它来修改操作数。)

递增运算符也是 C++命名的原因之一，它的寓意是“比 C 更进一步”。在早期的 Java 访谈中，Bill Joy (Sun 的创建者之一)说，“ Java = C++--”(C++的递减)。他的意思是，Java 是一种去除了 C++中多余的复杂性的语言，因此它是一种更为简单的语言。随着本书的进展，你会看到很多地方都变得更简单了，但是 Java 还不到“比 C++简单得多”的地步。

关系运算符

关系运算符会产生 **boolean** 类型的结果。它们会判断操作数的值的大小关系。如果这种关系为真，则关系表达式返回 **true**，如果这个关系为假，则返回 **false**。关系运算符有小于(<)，大于(>)，小于等于(<=)，大于等于(>=)，相等(==)以及不等(!=)。所有的内置数据类型都能比较相等(equivalence)和不等(nonequivalence)，但是对 **boolean** 类型，除了相等和不等的之外，其它比较是没意义的。

测试对象的相等性

关系运算符 **==** 和 **!=** 能用于所有对象，但是它们的意思经常会把新手们给搞胡涂。下面就是一个例子：

```
//: c03:Equivalence.java
import com.bruceeckel.simpletest.*;

public class Equivalence {
  static Test monitor = new Test();
  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);
    monitor.expect(new String[] {
      "false",
      "true"
    });
  }
} ///:~
```

表达式 **System.out.println(n1 == n2)** 会把比较结果的 **boolean** 值打印出来。很显然，由于这两个 **Integer** 对象是完全相同的，因此输出结果应该是先 **true** 后 **false**。然而，虽然这对象的“内容”是相同，但它们的 reference 不同，而 **==** 和 **!=** 比较的是对象的 reference。所以实际的输出应该是先 **false** 后 **true**。对此，新手们当然会吓一跳的。

要想比较这两个对象的实际内容，又该怎么办呢？必须使用每个类都有的，专门的 **equals( )**方法 (primitives 没有，因为**==** 和 **!=** 就用得很好)。下面就是该如何使用：

```
//: c03:EqualsMethod.java
import com.bruceeckel.simpletest.*;

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

现在的结果就是你预料中的 **true** 了。嗯，但是没那么简单，如果你创建了自己的类，比方说下面这个：

```
//: c03:EqualsMethod2.java
```

```
import com.bruceeckel.simpletest.*;

class Value {
  int i;
}

public class EqualsMethod2 {
  static Test monitor = new Test();
  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));
    monitor.expect(new String[] {
      "false"
    });
  }
} ///:~
```

兜了一圈又回来了：结果还是 **false**。这是因为 **equals( )** 的缺省行为是比较 reference。所以，除非你在新类里覆写(*override*) **equals( )**，否则你就没法获得预期的效果。很遗憾，我们要到第 7 章才会讲到覆写，要到第 11 章才会讲如何正确地定义 **equals( )**。但是在此期间，请留意 **equals( )**，这样能为你省下不少时间。

绝大多数 Java 类库的类都实现了 **equals( )**，所以它们会比较对象的内容而不是 reference。

逻辑运算符

逻辑运算符与 (&&) ，或 (||)，非 (!)会根据参数的逻辑关系产生一个 **true** 或 **false** 的 **boolean** 值。下面就是使用关系和逻辑运算符的例子：

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

public class Bool {
  static Test monitor = new Test();
  public static void main(String[] args) {
    Random rand = new Random();
    int i = rand.nextInt(100);
    int j = rand.nextInt(100);
    System.out.println("i = " + i);
    System.out.println("j = " + j);
    System.out.println("i > j is " + (i > j));
    System.out.println("i < j is " + (i < j));
    System.out.println("i >= j is " + (i >= j));
    System.out.println("i <= j is " + (i <= j));
    System.out.println("i == j is " + (i == j));
    System.out.println("i != j is " + (i != j));
```

```
    // Treating an int as a boolean is not legal
Java:
//! System.out.println("i && j is " + (i && j));
//! System.out.println("i || j is " + (i || j));
//! System.out.println("!i is " + !i);
    System.out.println("(i < 10) && (j < 10) is "
       + ((i < 10) && (j < 10)) );
    System.out.println("(i < 10) || (j < 10) is "
       + ((i < 10) || (j < 10)) );
    monitor.expect(new String[] {
      "%% i = -?\\d+",
      "%% j = -?\\d+",
      "%% i > j is (true|false)",
      "%% i < j is (true|false)",
      "%% i >= j is (true|false)",
      "%% i <= j is (true|false)",
      "%% i == j is (true|false)",
      "%% i != j is (true|false)",
      "%% \\(i < 10\\) && \\(j < 10\\) is
(true|false)",
      "%% \\(i < 10\\) \\|\\| \\(j < 10\\) is
(true|false)"
    });
  }
} ///:~
```

在 **expect( )** 语句的正则表达式里，括号是用来为表达式分组的，而‘**|**’表示逻辑“与”。所以：

```
(true|false)
```

表示这部分的字符串可能是‘**true**’或‘**false**’。由于有些字符在正则表达式里面有特殊意义，所以要让它们在正则表达式中表示其本来的字符，就必须用‘****’对它们进行转义。

逻辑与“AND”，或“OR”，非“NOT”只能用于 **boolean** 值。你不能像 C 或者 C++那样，在逻辑表达式里面把非 **boolean** 的值当作 **boolean** 值处理。也许你已经注意到了，这种失败的尝试已经被 **//!** 记号注释掉了。因而最后这个表达式，会先用关系运算符进行比较，再用逻辑运算符计算比较结果的值。

要知道，如果把 **boolean** 值当作 **String** 来用的话，它会被自动转换成相应的文本。

在上述程序里，你可以把 **int** 换成除 **boolean** 之外的任何 primitive 数据。但是，请记住浮点数的比较是十分严格的。两个数之间哪怕只有表示最小值的那一位有差别，也会认为是“不等”的。一个数只要有一位不是零，那它就不是零。

短接

处理逻辑运算符的时候，会碰到一种叫“短接 (short circuiting)”的现象。也就是说，只要能准确地知道整个表达式是真还是假，就立刻作出判断。这样就有可能会出现，无需计算逻辑表达式的后半部分就能作出判断的情况。下面就是一段演示短接的例子：

```
//: c03:ShortCircuit.java
// Demonstrates short-circuiting behavior.
// with logical operators.
import com.bruceeckel.simpletest.*;

public class ShortCircuit {
  static Test monitor = new Test();
  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");
    monitor.expect(new String[] {
      "test1(0)",
      "result: true",
      "test2(2)",
      "result: false",
      "expression is false"
    });
  }
} ///:~
```

每个测试都拿参数同常量进行比较，然后返回 true 或 false。此外，它还会打印一些东西，以证明被调用了。接着用下面的表达式进行测试：

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

可能，你会想当然地认为三个测试都要执行一遍，但是输出的结果却表明，事实并非如此。第一个测试返回了 **true**，所以表达式还要进行下

去。第二个测试返回了 **false**。这就是说整个表达式的结果肯定是 **false**，那么为什么还要判断下去呢？这么作太浪费了。实际上短接的初衷就是，如果能省掉一部分逻辑判断，性能就会得到提升。

位运算符

位运算符能让你逐位操控 primitive 类型的数据。两个参数的相对应的各位分别进行布尔运算，其结果就是位运算的结果。

位运算符是从 C 语言带过来的。C 有一些低级语言的特点，你可以用它去直接控制硬件，这样就要设置硬件的寄存器了。最早，Java 是被当作要嵌入电视机的机顶盒来设计的，所以保留一些底层功能还是情有可源的。不过你大概没多少机会来用位运算符。

如果输入位都是一，则“与运算符(**&**)”会返回一，否则就是零。输入的两位当中只要有一个一，则“或运算符(**|**)”会返回一，只有两个都是零的时候，它才返回零。如果输入的两位当中有，且只有一个一，则“异或运算符(**^**)”会返回一。非运算符(**~**，也被称为 *ones complement* 运算符)是一个单元运算符；它只需要一个参数。(其它的位运算符都是二元运算符。)非运算符会对输入位取反——零变成一，一变成零。

位运算符与逻辑运算符使用相同的符号，所以我们得造条理由来帮助记忆：因为位比较“小”，所以位运算符只用一个字符。

位运算符可以与 **=** 连起来用，以表示同时进行计算和赋值：**&=**，**|=** 和 **^=** 都是合法的。(至于**~**，它既然是一个单元运算符，也就不需要同 **=** 绑在一起了。)

Boolean 类型会被当作只有一位的值，所以它多少有点不同。你可以进行与，或，以及异或运算，但是不能进行非运算(可能是为了防止同逻辑非相混淆)。对于 **boolean**，位运算用逻辑运算的在功能上是相同的，只是它没有短接。此外 **boolean** 的位运算里有一个逻辑运算所没有的异或。**boolean** 值不能移位，至于什么是移位，我们马上就讲。

移位运算符

移位也是一种位运算。它只能用于整数型的 primitive 数据。左移位会把运算符(**<<**)左边那个操作数向左移它右边那个数所表示的位数(低位用零填充)。带符号的右移位会把运算符(**>>**)左边那个操作数向右移它右边那个数所表示的位数。带符号的右移位 **>>** 使用 “根据正负号来扩展(*sign extension*)”的规则：如果这个值是正的，则高位一律填零；如果这个值是负的，则高位全部填一。Java 也有不带正负号的右移位 **>>>** ，它用的是“一律用零来扩展(*zero extension*)”的规则：不论正负号，高位都填零。这是一个 C 和 C++都是没有的运算符。

如果对 **char**，**byte**，或者 **short** 进行了移位，它们会被先转换成 **int**，因此运算的结果也就成了 **int**。运算符的右边那这个数，只有低五位有效。这是为了防止移位的位数超过 **int** 的位数。如果操作数是 **long**，那么结果也是 **long**。这时运算符右边那个数的低六位有效，这样移位的位数就不会超过 **long** 的位数。

移位可以同等号连起来用(**<<=** 或 **>>=** 或 **>>=**)。lvalue 移了 rvalue 位之后，会被再存进 lvalue。但是不带符号的右移位和等号连用的时候会有一个问题。如果操作数是 **byte** 或 **short** 的话，结果就不对了。它们会被先提升到 **int** 再进行右移，但是重新赋值给原先的变量时，这个值就得被截短，所以移来移去结果都是 **-1**。下面这段程序就演示了这个效果：

```
//: c03:URShift.java
// Test of unsigned right shift.
import com.bruceeckel.simpletest.*;

public class URShift {
  static Test monitor = new Test();
  public static void main(String[] args) {
    int i = -1;
    System.out.println(i >>>= 10);
    long l = -1;
    System.out.println(l >>>= 10);
    short s = -1;
    System.out.println(s >>>= 10);
    byte b = -1;
    System.out.println(b >>>= 10);
    b = -1;
    System.out.println(b>>>10);
    monitor.expect(new String[] {
      "4194303",
      "18014398509481983",
      "-1",
      "-1",
      "4194303"
    });
  }
} ///:~
```

最后一次移位，没有把运算结果赋还给 **b** 而是直接打印出来了，所以它才是正确的。

下面这段例程演示了所有与位操作相关的运算符的用法：

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

public class BitManipulation {
  static Test monitor = new Test();
```

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

  long l = rand.nextLong();
  long m = rand.nextLong();
  printBinaryLong("-1L", -1L);
  printBinaryLong("+1L", +1L);
  long ll = 9223372036854775807L;
  printBinaryLong("maxpos", ll);
  long lln = -9223372036854775808L;
  printBinaryLong("maxneg", lln);
  printBinaryLong("l", l);
  printBinaryLong("~l", ~l);
  printBinaryLong("-l", -l);
  printBinaryLong("m", m);
  printBinaryLong("l & m", l & m);
  printBinaryLong("l | m", l | m);
  printBinaryLong("l ^ m", l ^ m);
  printBinaryLong("l << 5", l << 5);
  printBinaryLong("l >> 5", l >> 5);
  printBinaryLong("(~l) >> 5", (~l) >> 5);
  printBinaryLong("l >>> 5", l >>> 5);
  printBinaryLong("(~l) >>> 5", (~l) >>> 5);
  monitor.expect("BitManipulation.out");
}
static void printBinaryInt(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 printBinaryLong(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();
    }
  } ///:~
```

程序结尾定义的两个方法，**printBinaryInt( )**和**printBinaryLong( )**，它们分别拿 **int** 和 **long** 做参数，然后先打印一段说明性的文字，再以二进制的方式打印出这个数字。现在你可以不用关心它是怎么实现的。

你会注意到，这里用的是 **System.out.print( )**而不是 **System.out.println( )**。**print( )**方法不会加换行符，所以它能让你分几次输出一行。

这里的 **expect( )** 拿一个文件名作参数，它会从这个文件读取预期的结果(其中可以有，也可以没有正则表达式)。对于那些结果太长，不适宜列在书里的程序，这种做法很实用。这个文件的扩展名是“.out”，它也算是源代码，可以从 *www.BruceEckel.com* 下载。有兴趣的话，可以打开这个文件，看看到底输出些什么(或者直接运行这个程序)。

除了演示了位运算符对 **int** 和 **long** 的运算效果，这段程序还显示了 **int** 和 **long** 的最小值，最大值，+1 和-1 的两进制表达形式。注意高位表示符号：0 表示正号，1 表示负号，**int** 部分的输出是这样的：

```
-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:
   00001011111011010001110011001100
i & j, int: 58720644, binary:
   00000011100000000000000110000100
i | j, int: 199212028, binary:
   00001011111011111101101111111100
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
```

两进制表达方式被称为 *signed two's complement*。

三元 if-else 运算符

这个运算符有些与众不同，它有三个操作数。但它确实是一个返回值的运算符，这一点与下一节会讲的 if-else 语句不同。这个表达式的形式是这样的：

```
boolean-exp ? value0 : value1
```

如果 *boolean-exp* 判断是 **true**，则计算 *value0*，然后这个值就是运算的结果。如果 *boolean-exp* 是 **false**，则计算 *value1* 的值，然后拿它当运算的结果。

当然，你可以使用普通的 **if-else** 语句(后面会讲)，但是这个三元运算符更为简洁。尽管 C (发明这个运算符的语言) 以它是一种简洁的语言而自豪，而且 Java 引入这个三元操作符就有提高效率的意思，但是在平常使用的时候还是要谨慎——它有可能会破坏程序的可读性。

你既可以为它的副作用，也可以为要得到一个值而使用这个运算符，但是一般来说还是这个值，因为有没有值才是它与 if-else 相区别的地方。下面就是一个例子：

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

你能看到这段代码要比不用三元运算符的紧凑了许多：

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

第二种写法读起来更简单，而且也多敲不了几下键盘。所以，使用三元运算符的时候一定要三思而后行——一般来说，在两选一的情况下，使用这个运算符是比较合理的。

逗号运算符

在 C 和 C++里面，逗号不但被用于分隔函数的各个参数，而且还是一种连续赋值 (sequential evaluation)的运算符。Java 里面，唯一一个把逗号当运算符用的地方是本章后面会讲的 **for** 循环。

String 的+运算符

在 Java 里，加号(+)是一个有特殊用途的运算符：正如你已经知道的，它还能连接字符串。虽然与习惯上的用法有些不一样，但 + 的这种用法还是比较自然的。设计 C++的时候，大家都觉得能重载运算符(*operator overloading*)应该是一个不错的主意，因此 C++提供了这种特性，它让程序员能为几乎所有的运算符都加上特别的意思。不幸的是，当程序员要设计自己的类库的时候，运算符重载以及 C++的其它一些限制，就会使它变得过于复杂了。如果 Java 想提供运算符重载的话，会比用 C++简单许多，但是设计者们还是觉得这个特性太复杂了，因此 Java 的程序员不能像 C++的程序员那样去重载运算符。

加号 (+)用在 **String** 上的时候会很有趣。如果表达式以 **String** 开始的，那么所有后面跟着的操作数都必须是 **String** (记住，编译器会把用引号括起来的字符转换成 **String** 的)：

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

这里 Java 编译器会把 **x**，**y**，和 **z** 都转换成它们的 **String** 形式而不是先把它们加起来再转换。如果程序是：

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

那么 Java 就会把 **x** 转换成 **String**。

常见的使用运算符方面的错误

最常见的错误是自己还不能完全肯定表达式会如何运算，就把括号给省了。Java 里面也一样。

C 和 C++里面还有一个出现频率极高的错误：

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

很明显程序员是想作相等性的测试 (**==**) 而不是赋值。在 C 和 C++里面，只要 **y** 不等于零，这个赋值结果就永远是 **true**，因此它很可能会形成一个无限循环。在 Java 里面，编译器在等一个 **boolean** 值，而这个表达式的结果不是 **boolean**，而且编译器不会把 **int** 转换成 **boolean**，因此编译的时候它就会直接报错，所以这种错误会在程序还没运行的时候就被纠出来。因此 Java 永远也不会发生此类错误。(唯一一种不会得到编译时错误的情形是，**x** 和 **y** 都是 **boolean**，这样 **x=y** 就是一个合法的表达式，在上述代码中，这很有可能是个错误。)

C 和 C++还有一个类似的问题，那就是位运算的“与”和“或”同逻辑运算的“与”和“或”。位运算符只有一个字符 (**&** 或 **|**)，而逻辑运算符要有两个字符 (**&&** 或 **||**)。这就跟 **=** 和 **==** 一样，只是敲一下键和敲两下的区别，因此很容易就搞混了。Java 编译器再一次挺身而出，它不会让你傲慢地将一种类型当作另一种类型来用。

类型转换运算符

cast 的意思是“把东西浇到模子里”。只要情况合适，Java 会自动的将一种数据类型的转换成另一种数据类型。比方说，把整数值赋给浮点变量的时候，编译器会自动将这个 **int** 转换成 **float**。但是碰到一些不会自动进行这种转换的情况的时候，你也可以明确地告诉它进行转换，或者强制进行类型转换。

要进行这种转换，你就得把数据的目标类型(包括所有的修饰符)放进变量左边的括号里。下面就是举例：

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

正如你所看到的，你既可以对数值，也可以对变量进行转换。但是，现在演示的这两种转换都是多余的，因为一旦有必要，编译器就会自动将 **int** 提升为 **long**。不过用了这种多余的转换之后，代码会变得更清晰。但是在有些情况下，为了要让代码能编译通过，转换就可能是必须的了。

在 C 和 C++里面，类型转换是一件让人头疼的事。而 Java 的类型转换则是安全的。因为碰到所谓的 *narrowing conversion* 的时侯(也就是说，当你将一个能保存较多信息的数据类型，转换成一个保存不了这么多

信息的数据类型时)，它会产生一个异常。编译器会强制你明确地进行转换，实际上它的潜台词是“这么做可能很危险——如果你一定要做，那么请明确地告诉我”。对于 *widening conversion*，就不必进行显式转换了，因为新类型可以存储比旧类型更多的信息，所以什么都丢不了。

Java 允许你对除 **boolean** 之外的其它 primitive 类型的数据进行任意的转换，而 **boolean** 则根本不能转换。类是不允许转换的。要让它们相互转换，必须要有特殊的方法。(**String** 是一个特例。此外，本书后面会讲，所有对象都可以在它的类系(*family of types*)里面上下传递；**Oak**(橡树)可以被传到 **Tree**，反之也可以，但是它不能传给另一个类系的类型，比如 **Rock**。)

常量 (Literals)

通常情况下，往程序插常量值的时候，编译器能准确地知道这是什么类型的。但是在有些情况下，这种类型是模棱两可的。碰到这种情况的时候，你就只能加入与常量值相关的字符来提供额外信息，以此来引导编译器。下面这段程序就演示了这种字符：

```
//: c03:Literals.java

public 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 (but can be
confusing)
  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
} ///:~
```

表示 16 进制数的时候，要用 **0x** 或者 **0X** 打头，后面跟上 **0-9** 或者大小写都可以的 **a-f**。所有的整数类型都能以 16 进制的形式加以处理。要是你用一个比这种类型的最大值更大的数来初始化变量的话(不管这个数字是以哪种形式表示的)，编译器就会报一个错。注意一下，在前面的程序里，我们已经给出了 **char**，**byte** 以及 **short** 所能表示的最大的值。如

果超出了这个范围，编译器会自动地将这个值转换成 **int**，然后告诉你，赋值得时候还得进行 *narrowing cast*。于是你就知道过界了。

8 进制数是由 0 后面跟着一串 0-7 的数字组成的。C，C++或者 Java 里面都没有两进制常量值的表示方法。

跟在常量值后面的那个数字标明了它的类型。大写或者小写的 **L** 表示 **long**，大写或者小写的 **F** 表示 **float**，而大写或者小写的 **D** 表示 **double**。

讲到指数的时候，我总觉得有些郁闷。在科研和工程领域，‘e’表示自然对数的底数，大概是 2.718。(Java 的 **Math.E** 会给出一个更为精确的 **double** 值。) 像 1.39 x e^{-47} 这样的指数表示形式，实际上就是 1.39 x 2.718^{-47}。但是设计 FORTRAN 的时候，设计师们用 e 来表示我们通常认为的“10 的幂”。这是个很奇怪的决定，因为 FORTRAN 是为科研和工程计算设计的，更何况设计人员在引入这种两义性的时候应该很敏感才对。[17]但是不管怎么说，这种传统带到了 C 和 C++以及现在的 Java。所以，如果你已经习惯于把 **e** 当作自然对数的底数的话，那么看到 Java 程序里的 **1.39 e-47f** 时就得转一下脑筋了；实际上它表示 1.39 x 10^{-47}。

提醒一下，如果编译器能够确定数据的类型，那你就不必再加上后缀字符了。比如：

```
long n3 = 200;
```

这里没有会引发两义性的地方，所以 200 后面的 L 是多余的。但是如果是

```
float f4 = 1e-47f; // 10 to the power
```

这个 f 就是必须的了。因为编译器通常会把用指数表示的数字当成 double，所以如果没有后缀的 f 的话，它就会给你一个错误信息，告诉你你必须将 **double** 转换成 **float**。

提升

你会发现，如果是对比 **int** 小的数据(也就是 **char**，**byte**，或 **short** 型的数据)进行数学或位运算的话，这些值会先被提升到 **int**，再进行运算，所以运算的结果仍然是 **int**。所以当你把这个值赋还给那个较小的类型时，你就必须进行类型转换。(而且，由于是把值赋给一个较小的类型，因此会丢失部分信息。)总之，表达式里面最大的那个数据的类型会

决定表达式的结果的数据类型；如果你求一个 **float** 和一个 **double** 的积，得到的结果也是 **double**；如果你求一个 **int** 和一个 **long** 的和，结果也是 **long**。

Java 没有 “sizeof”

C 和 C++的 **sizeof( )**运算符是有特殊用途的：它会告诉你，数据要占用多少字节的内存。C 和 C++要有 **sizeof( )**的最主要的原因是为了移植。相同的数据类型在不同的机器上可能会不一样，所以程序员在进行与数据大小相关的操作之前必须先了解一下这个类型有多大。比如，一台计算机用 32 位保存整数，而另一台用 16 位。程序员可以在前面那台里面存储更大的数字。可能你也猜到了，对 C 和 C++的程序员来说，移植性是一件很头疼的事。

Java 没有移植的问题，因此不需要 **sizeof( )**，所有数据类型在所有的机器上都是相同的。你不必为这种级别的移植性操心——它已经做进语言里面了。

重访优先级

一次课上，有个学生听到我在抱怨运算符的优先级有多难记的时候，不加思索地给出了一段助记词“Ulcer Addicts Really Like C A lot.”

Mnemonic	Operator type	Operators
Ulcer	单元 Unary	**+ - ++--**
Addicts	算术 Arithmetic (以及移位)	*** / % + - << >>**
Really	关系 Relational	**> < >= <= == !=**
Like	逻辑 Logical (以及位运算)	**&& || & | ^**
C	条件 Conditional (三元的)	**A > B ? X : Y**
A Lot	赋值 Assignment	**=** (以及像 ***=*** 形式的赋值)

当然，如果把移位和位运算算上的话，这就不是一个完美的助记词了，但是如果不考虑位运算的话，它还不错。

运算符的的总结

下面这段程序演示了各种 primitive 数据都可以使用哪些的运算符。基本上是用不同类型的数据重复同一个例子。由于有问题的代码都用 **//!** 给注释掉了，因此编译这个文件是不会有问题的。

```
//: c03:AllOps.java
// Tests all the operators on all the primitive data
types
```

```
// to show which ones are accepted by the Java
compiler.

public 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，**boolean** 的限制性是很强的。你只能赋给它 **true** 和 **false** 这两个值，而且只能测试它的真或假，你不能把 **boolean** 值加起来，或者进行其它什么操作。

你也看到了，进行数学运算的时候，**char**，**byte**，以及 **short**，都会先进行提升，而运算结果也是 **int**，因此如果想把它赋还给原先那个变量的话，就必须明确地进行类型转换 (属于会丢失信息的 *narrowing conversion*)。但是对于 **int**，你就无需再作转换了，因为它已经是 **int** 了。不过也别太大意了。如果你把两个足够大的 **int** 乘起来，结果就会溢出。下面这段程序就演示了这种情况：

```
//: c03:Overflow.java
// Surprise! Java lets you overflow.
```

```
import com.bruceeckel.simpletest.*;

public class Overflow {
  static Test monitor = new Test();
  public static void main(String[] args) {
    int big = 0x7fffffff; // max int value
    System.out.println("big = " + big);
    int bigger = big * 4;
    System.out.println("bigger = " + bigger);
    monitor.expect(new String[] {
      "big = 2147483647",
      "bigger = -4"
    });
  }
} ///:~
```

编译器没报错，也没给警告信息，运行的时候也没抛出异常。Java 是很强大，但还没有强大到那种程度。

同算术运算一样，**char**，**byte**，和 **short** 在进行混合赋值(compound assignment)的时候也会被提升，但是却不需要进行类型转换。另一方面，不作类型转换也会使代码变得更简单。

你能看到，除了 **boolean** 之外，所有的 primitive 类型都能被转换成其它的 primitive 类型。还有，在进行 narrowing conversion 的时候，你得对它的效果有准备，否则信息就会在转换过程中不知不觉地丢了。

执行控制

Java 全盘照抄 C 的执行控制语句，所以如果你有 C 或 C++的编程经验，那么你应该对这里讲的东西很熟悉了。绝大多数过程语言都有一些控制语句，而且语言和语言之间也有很多重复的地方。Java 的关键词里就有 **if-else**，**while**，**do-while**，**for**，以及名为 **switch** 的选择语句。但是 Java 没有提供饱受非议的 **goto**(其实用得得法，它会是解决某些问题的捷径)。你还可以像用 **goto** 那样，在程序里到处跳转，但是这种跳转的限制这要比原先的 **goto** 多得多。

true 和 false

所有的条件语句都使用条件表达式所返回的结果来判断应该执行哪条路径。举个例子，**A == B** 就是一个条件表达式。这个表达式用了条件运算符 == 来判断 **A** 的值是否与 **B** 的值相等。这个表达式会返回 **true** 或 **false**。你在本章前面部分看到过的所有关系运算符都能用于条件语句。注意，Java 不会允许你把数字当作 **boolean** 用的，尽管 C 和 C++里面允许这么做(非零值表示 **true**，零表示 **false**)。如果你要对非

boolean 的值进行 **boolean** 测试，比如 **if(a)**，你就必须通过像 **if(a! = 0)**这样的条件表达式先将它转换成 **boolean** 值。

if-else

if-else 也许是最基本的程序流程控制语句。**else** 是可选的，所以 **if** 实际上有两种用法：

```
if(Boolean-expression)
  statement
```

或者

```
if(Boolean-expression)
  statement
else
  statement
```

“条件判断(conditional)”必须要能产生一个 **boolean** 结果。*statement* 部分既可以是以分号表示结束的简单语句，也可以是用花括号括起来的复合语句。只要用到“*statement*”，就表示它既可以是简单语句，也可以是复合语句。

举一个 **if-else** 的例子，下面的 **test( )**方法会告诉你，你猜的数字是大于，小于，还是等于这个数字：

```
//: c03:IfElse.java
import com.bruceeckel.simpletest.*;

public class IfElse {
  static Test monitor = new Test();
  static int test(int testval, int target) {
    int result = 0;
    if(testval > target)
      result = +1;
    else if(testval < target)
      result = -1;
    else
      result = 0; // Match
    return result;
  }
  public static void main(String[] args) {
    System.out.println(test(10, 5));
    System.out.println(test(5, 10));
    System.out.println(test(5, 5));
    monitor.expect(new String[] {
      "1",
      "-1",
      "0"
    });
  }
```

```
} ///:~
```

对流程控制语句使用缩进是一种约定俗成的做法，因为这样能让读者更容易地看出程序的开始和结束。

return

return 关键词有两个作用：首先，它会告诉你方法要返回哪个值(如果不是 **void** 返回型的)；其次，它会立即返回那个值。我们可以利用这个特性重写前面那个 **test( )**方法：

```
//: c03:IfElse2.java
import com.bruceeckel.simpletest.*;

public class IfElse2 {
  static Test monitor = new Test();
  static int test(int testval, int target) {
    if(testval > target)
      return +1;
    else if(testval < target)
      return -1;
    else
      return 0; // Match
  }
  public static void main(String[] args) {
    System.out.println(test(10, 5));
    System.out.println(test(5, 10));
    System.out.println(test(5, 5));
    monitor.expect(new String[] {
      "1",
      "-1",
      "0"
    });
  }
} ///:~
```

不必再用 **else** 了，因为执行完 **return** 之后，方法就结束了。

循环语句

循环是通过 **while**，**do-while** 以及 **for** 来控制的，它们被统称为循环语句(*iteration statements*)。*statement* 会不断执行下去，直到 **Boolean-expression** 为 false。**while** 循环的用法是

```
while(Boolean-expression)
  statement
```

每次循环之前都要判断 *Boolean-expression*，然后下次运行 *statemenet* 之前，还要再作判断。

下边这段简单的程序会不停地生成随机数，直到它满足某项条件：

```
//: c03:WhileTest.java
// Demonstrates the while loop.
import com.bruceeckel.simpletest.*;

public class WhileTest {
  static Test monitor = new Test();
  public static void main(String[] args) {
    double r = 0;
    while(r < 0.99d) {
      r = Math.random();
      System.out.println(r);
      monitor.expect(new String[] {
        "%% \\d\\.\\d+E?-?\\d*"
      }, Test.AT_LEAST);
    }
  }
} ///:~
```

它用 **Math** 类库的 **static** 的 **random( )**方法来生成 0 到 1 之间的 **double** 值。(包括 0 但不包括 1。)**while** 语句的条件表达式的意思是“只要这个值比 0.99 小，就给我一直作下去”。每次运行这段程序的时候，它的输出都是不同的。

你会看到，在 **expect( )** 语句的字符串清单的后面，还跟着一个 **Test.AT_LEAST** 标记。**expect( )**还有几个能修改其行为的标记；这就是其中之一，它表示这一行应该至少显示一次，当然还可以显示其它东西 (这些东西就忽略了)。这里，它的意思是“你应该至少能看到一个 double 值。”

do-while

do-while 的形式是

```
do
  statement
while(Boolean-expression);
```

while 与 **do-while** 的唯一区别就是，即使第一次判断的结果是 false，**do-while** 的 *statement* 也会至少运行一次。在 **while** 语句中，如果第一次判断是 false，那么 *statement* 就再也没机会执行了。实际编程的时候，**while** 会比 **do-while** 用得更多。

for

for 语句会在第一次进入循环之前先做初始化。再作条件判断，每次循环之后还要有一些“步进调整”。**for** 循环的用法是：

```
for(initialization; Boolean-expression; step)
  statement
```

initialization(初始化)，*Boolean-expression*(进行条件测试)以及 *step*(步进调整)都可以为空。每次循环之前，它会测试 *Boolean-expression*，只要一看到它是 **false**，它就会跳出 for 语句，去执行后面的命令。每次循环之后，它还要执行 *step*。

for 循环通常被用于“计数”的任务：

```
//: c03:ListCharacters.java
// Demonstrates "for" loop by listing
// all the lowercase ASCII letters.
import com.bruceeckel.simpletest.*;

public class ListCharacters {
  static Test monitor = new Test();
  public static void main(String[] args) {
    for(int i = 0; i < 128; i++)
      if(Character.isLowerCase((char)i))
        System.out.println("value: " + i +
          " character: " + (char)i);
    monitor.expect(new String[] {
      "value: 97 character: a",
      "value: 98 character: b",
      "value: 99 character: c",
      "value: 100 character: d",
      "value: 101 character: e",
      "value: 102 character: f",
      "value: 103 character: g",
      "value: 104 character: h",
      "value: 105 character: i",
      "value: 106 character: j",
      "value: 107 character: k",
      "value: 108 character: l",
      "value: 109 character: m",
      "value: 110 character: n",
      "value: 111 character: o",
      "value: 112 character: p",
      "value: 113 character: q",
      "value: 114 character: r",
      "value: 115 character: s",
      "value: 116 character: t",
      "value: 117 character: u",
      "value: 118 character: v",
      "value: 119 character: w",
      "value: 120 character: x",
      "value: 121 character: y",
      "value: 122 character: z"
```

```
    });
  }
} ///:~
```

注意，变量 **i** 是在程序要用到它的地方，在 for 循环的“控制表达式(control expression)”里面定义的；而不是在程序刚开始地方，在表示开始的花括号后面定义的。**i** 的作用域只有 **for** 语句这么大。

这段程序还用到了 **java.lang.Character** 这个“wrapper”类，它不但会把 primitive 的 **char** 包进对象，而且还提供了一些别的的方法。这里用到了 **static** 的 **isLowerCase( )**方法来侦测这个字符是不是小写的。

C 之类的传统的过程语言会要求所有的变量都必须在一段程序的开头部分定义，这样编译器创建块的时候会为这些变量分配空间。在 Java 和 C++里面，你可以在任意地方声明变量。这样就能使编程风格显得更为自然，而代码的可读性也提高了。

你可以在 **for** 语句里定义多个变量，但必须是相同类型的。

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

在 **for** 语句里定义的 **int** 包括 **i** 和 **j** 。只有 **for** 语句才有在“控制表达式”里定义变量的能力。千万别在其它选择语句或循环语句里这么做。

逗号运算符

在本章的前面部分，我曾经提到过，Java 只在一个地方使用“逗号运算符”(而不是“逗号分隔符(comma *seperator*)”，后者是用来分隔定义和方法参数中的变量的)：在 for 循环的“控制表达式”里。在“控制表达式”的初始化和步进调整部分里，你都可以写很多用逗号分隔开的语句，而这些语句会按顺序依次执行。前面这段代码就用到了这个特性。下面再举一个例子：

```
//: c03:CommaOperator.java
import com.bruceeckel.simpletest.*;

public class CommaOperator {
  static Test monitor = new Test();
  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);
    }
    monitor.expect(new String[] {
```

```
        "i= 1 j= 11",
        "i= 2 j= 4",
        "i= 3 j= 6",
        "i= 4 j= 8"
      });
  }
} ///:~
```

可以看到在初始化和步进调整部分，这些语句是按循序执行的。此外，初始化部分可以定义任意数量的同一种类型的变量。

break 和 continue

你也可以在循环语句的正文里，用 **break** 和 **continue** 控制循环的流程。**break** 会忽略尚未执行的循环代码，并且退出循环。而 **continue** 会跳过尚未执行的代码，中断本次循环，再进入下一周期的循环。

这段程序演示了如何在 **for** 和 **while** 循环中使用 **break** 和 **continue**：

```
//: c03:BreakAndContinue.java
// Demonstrates break and continue keywords.
import com.bruceeckel.simpletest.*;

public class BreakAndContinue {
  static Test monitor = new Test();
  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);
    }
    monitor.expect(new String[] {
      "0",
      "9",
      "18",
      "27",
      "36",
      "45",
      "54",
      "63",
      "72",
      "10",
      "20",
      "30",
```

```
        "40"
      });
    }
} ///:~
```

在 **for** 循环里，**i** 永远也到不了 100，因为等 **i** 到 74 的时候，**break** 就中止循环了。一般来说，只有在不能事先预知中断条件什么时候发生的情况下，才会这样使用 **break**。**continue** 语句的作用是，只要 **i** 不能整除 9，它就让程序重新开始下一轮循环(这样 **i** 就会 递增)。如果能整除 9，就把这个值打印出来。

第二部分演示了所谓的“无限循环”。从理论上讲，这个循环会永远进行下去，但是它里面有个能中断循环的 **break**。此外你还会看到 **continue** 会跳过尚未执行的部分，重新开始下一轮循环。(这样在第二个循环里，只有当 **i** 能整除 10 的时候才会有东西打印。)此外，0 也被打印出来了，这是因为 0%10 等于 0。

还有一种形式的无限循环，**for(;;)**。编译器对 **while(true)**和 **for(;;)** 是一视同仁的，所以选哪个只是编程风格的问题。

臭名昭著的“goto”

自从有了编程语言就有了 **goto**。实际上，在汇编语言时代，**goto** 是程序流程控制的雏形：“如果条件 A，则跳转到这里，否则跳转到那里”。实际上，编译器最终生成的都是汇编代码。所以如果你读过这种代码的话，你会看到程序里充斥着跳转(Java 编译器会产生它自己的“汇编代码”，但是这种代码是在 Java 虚拟机上运行而不是直接在 CPU 硬件上运行)。

goto 则是源代码级别上的跳转，而这才是给它带来坏名声的真正原因。如果程序总是从这里跳到那里，那是不是有什么办法能重新整理代码让它不要那么“跳”呢？随着 Edsger Dijkstra 那篇著名的名为“Goto considered harmful”的论文的发表，**goto** 真正到了山穷水尽的地步。从那时起，对 goto 的穷追猛打成了一种时尚，甚至有人动议要把它从关键词里驱逐出去。

其实碰到这种情况，中庸之道才是最可取。问题不在于用没用 **goto**，而在于是不是滥用了 **goto**；在一些极为特殊的情况下，实际上 **goto** 是最好的流程控制语句。

尽管 **goto** 是 Java 的保留词，但它并没把它作进去；Java 没有 **goto**。但是有了 **break** 和 **continue** 这两个关键词，它确实能做出一些类似跳

转的效果。这不是跳转，实际上这还是一种跳出循环的方法。之所以会把它们同 **goto** 相提并论，是因为它们使用了相同的机制：label(标签)。

标签是一个后面有冒号的标识符，就像这样：

```
label1:
```

在 Java 里，唯一能放标签的地方，就是在循环语句的外面。而且必须直接放——在循环语句和标签之间不能有任何东西。而这么做的唯一理由就是，你会嵌套多层循环或选择。因为通常情况下 **break** 和 **continue** 关键词只会中断当前循环，而用了标签之后，它就会退到 label 所在的地方：

```
label1:
outer-iteration {
  inner-iteration {
    //...
    break; // 1
    //...
    continue;  // 2
    //...
    continue label1; // 3
    //...
    break label1;  // 4
  }
}
```

在案例 1 中，**break** 中断了内部的循环，并且退到外部循环。案例 2 中，**continue** 会重新移到内部循环的最开始。但是在案例 3 中，**continue label1** 会在中断内部循环的同时中断外部循环，再一气退到 **label1**。然后从头开始执行循环，但是这次是从外部开始。再案例 4 中，**break label1** 也会退到 **label1**，但它不会再进入循环。实际上它同时退出两个循环。

下面是用 **for** 循环举例：

```
//: c03:LabeledFor.java
// Java's "labeled for" loop.
import com.bruceeckel.simpletest.*;

public class LabeledFor {
  static Test monitor = new Test();
  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++) {
        System.out.println("i = " + i);
```

```
        if(i == 2) {
          System.out.println("continue");
          continue;
        }
        if(i == 3) {
          System.out.println("break");
          i++; // Otherwise i never
               // gets incremented.
          break;
        }
        if(i == 7) {
          System.out.println("continue outer");
          i++; // Otherwise i never
               // gets incremented.
          continue outer;
        }
        if(i == 8) {
          System.out.println("break outer");
          break outer;
        }
        for(int k = 0; k < 5; k++) {
          if(k == 3) {
            System.out.println("continue inner");
            continue inner;
          }
        }
      }
    }
    // Can't break or continue to labels here
    monitor.expect(new String[] {
      "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"
    });
  }
} ///:~
```

提示一下，**break** 会退出 **for** 循环，而且不经一轮完整的 **for** 循环，递增操作就不会发生。由于 **break** 跳过了递增表达式，因此要在 **i==3** 的时候直接进行了递增操作。在 **i==7** 的时候的 **continue outer** 语句

也直接转入了最高层的循环，因此也跳过了递增，所以这时也要进行直接递增。

如果不是 **break outer** 语句，你就根本就没办法从 **inner** 循环退到 **outer** 循环的外面，因为 **break** 只能退出退出最内层的循环。(**continue** 也一样。)

当然，如果退出循环的同时还要退出方法，你可以直接使用 **return**。

下面演示的是带标签的 **break** 和 **continue** 同 **while** 一起运行的效果：

```
//: c03:LabeledWhile.java
// Java's "labeled while" loop.
import com.bruceeckel.simpletest.*;

public class LabeledWhile {
  static Test monitor = new Test();
  public static void main(String[] args) {
    int i = 0;
    outer:
    while(true) {
      System.out.println("Outer while loop");
      while(true) {
        i++;
        System.out.println("i = " + i);
        if(i == 1) {
          System.out.println("continue");
          continue;
        }
        if(i == 3) {
          System.out.println("continue outer");
          continue outer;
        }
        if(i == 5) {
          System.out.println("break");
          break;
        }
        if(i == 7) {
          System.out.println("break outer");
          break outer;
        }
      }
    }
    monitor.expect(new String[] {
      "Outer while loop",
      "i = 1",
      "continue",
      "i = 2",
      "i = 3",
      "continue outer",
      "Outer while loop",
      "i = 4",
      "i = 5",
      "break",
      "Outer while loop",
      "i = 6",
```

```
        "i = 7",
        "break outer"
      });
    }
  } ///:~
```

这些规则也适用于 **while**：

1. 普通的 **continue** 会退到内部循环的最开始，然后继续执行内部循环。
2. 带标签的 **continue** 会跳转到标签，并且重新进入直接跟在标签后面的循环。
3. **break** 会从循环的“底部溜出去”。
4. 带标签的 **break** 会从由这个标签标识的循环的“底部溜出去”。

记着，在 Java 里，能让你使用标签的唯一理由就是，在嵌套循环的同时还要用 **break** 和 **continue** 退出多层循环。

在 Dijkstra 那篇名为“goto considered harmful”的论文里，他明确反对的是标签，而不是 goto。他发现程序 bug 的数量似乎是同标签的数量同步增长的。由于标签和 goto 在程序的执行图(excecution graph)里引入了循环，因此程序的静态分析变得非常困难。但是 Java 标签却没有这种问题，因为它们的放置位置是有严格限制的，而且不能用于转移程序的控制权。此外，这里还有一个非常有趣的现象。我们限制了语言的某个特性，但是它居然变得更加有用了。

switch

有时，**switch** 会被归为选择语句。**switch** 语句会根据“整数表达式(integral expression)”的值，决定应该运行哪些代码。它的形式是：

```
switch(integral-selector) {
  case integral-value1 : statement; break;
  case integral-value2 : statement; break;
  case integral-value3 : statement; break;
  case integral-value4 : statement; break;
  case integral-value5 : statement; break;
  // ...
  default: statement;
}
```

Intergal-selector 是一个会产生整数值的表达式。**switch** 会将这个值与下面的 *integral-value* 逐个作比较。如果找到了匹配的，它会执行相应的 *statement*(简单的或是复合的)。如果没有找到匹配的，则执行 **default** 的 *statement*。

你会发现，在上面那段程序里，每个 **case** 的最后都有 **break**。这样程序运行完相关指令之后，就会跳出 **switch**。这是 **switch** 语句的一般形式，不过 **break** 是可选的。如果没有 **break**，程序会继续执行下一个 **case** 的指令，直到它遇见 **break**。虽然你不会用这个功能，但是有经验的程序员可能会。注意最后那段跟在 **default** 后面的语句，它没有 **break**，这是因为程序运行到这里，接下来用不用 break 都会退到同一个地方。如果你想统一风格，也可以把 **break** 放到 **default** 的最后，这样一点坏处也没有。

switch 语句能很有条理地实现多路选择(就是从很多运行路径当中选一个)，然而它需要一个能产生像 **int** 或 **char** 这样的整数值的选择条件。假如你用字符串或是浮点数作选择条件，那么 **switch** 语句就不能正常工作了。对于非整数的类型，你只能使用一连串的 **if** 语句。

下面的程序会先随机生成一些字母，然后再判断他们是元音还是辅音：

```
//: c03:VowelsAndConsonants.java
// Demonstrates the switch statement.
import com.bruceeckel.simpletest.*;

public class VowelsAndConsonants {
  static Test monitor = new Test();
  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");
      }
      monitor.expect(new String[] {
        "%% [aeiou]: vowel|[yw]: Sometimes a vowel|"
+
          "[^aeiouyw]: consonant"
      }, Test.AT_LEAST);
    }
  }
} ///:~
```

由于 **Math.random( )**生成的是 0 到 1 之间的值，因此你得先乘以范围的大小 (字母表有 26 个字母)，然后再加上偏移量，也就是这些数字里面最小的那个。

虽然看上去我们是根据字符来决定跳转，但实际上 **switch** 语句用的还是这些字符的整数值。在 **case** 语句里，单引号括起来的字符在用于比较的时候，会返回整数值。

注意一下，为了让一段代码匹配多项选择，**case** 是如何“堆叠”的。你应该知道，**break** 放在哪个 **case** 的后面是很重要的；否则，程序会直接漏过去执行下一个 **case** 的程序。

expect( )语句的正则表达式用‘**|**’表示三种可能性。‘**[]**’框了的一“组”正则表达式的字符，所以第一种可能性是，“a，e，i，o，u 中间的一个，跟上一个冒号，然后是单词‘vowel’”。第二种可能性是，y 或者 w，然后是：“Sometimes a vowel.”，在最后一种可能性里，字符集是用‘^’开头，它表示“不在这个范围里面的字符，”所以它表示匹配非元音的字符。

计算细节

下面这条语句：

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

值得我们仔细看看。**Math.random( )**生成一个 **double**，所以 26 会先被转换成 **double** 再进行乘法运算，于是运算的结果也是 **double**。也就是说‘**a**’必须被转换成 **double** 才能进行加法运算。因此要用类型转换把这个 **double** 的结果转换成 **char**。

但是转换成 **char** 的过程中有作了些什么呢？也就是说，如果值是 29.7，那么你把它转化成 **char** 的时候，是把它当 30 还是 29 呢？下面这段程序会告诉你答案：

```
//: c03:CastingNumbers.java
// What happens when you cast a float
// or double to an integral value?
import com.bruceeckel.simpletest.*;

public class CastingNumbers {
  static Test monitor = new Test();
  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));
```

```
      monitor.expect(new String[] {
        "above: 0.7",
        "below: 0.4",
        "(int)above: 0",
        "(int)below: 0",
        "(char)('a' + above): a",
        "(char)('a' + below): a"
      });
    }
} ///:~
```

所以答案就是，将 **float** 或 **double** 转换成整数的时候，它总是将其后面的小数截去。

第二个问题与 **Math.random( )**有关。它会生成一个 0 到 1 之间的值，这个值包不包括‘1’？用数学术语，它是(0,1)，还是 [0,1]，还是 (0,1] 还是 [0,1)？(方括号表示包括，而括号表示“不包括。”)我们还是再用一个测试程序来寻找答案吧：

```
//: c03:RandomBounds.java
// Does Math.random() produce 0.0 and 1.0?
// {RunByHand}

public class RandomBounds {
  static void usage() {
    System.out.println("Usage: \n\t" +
      "RandomBounds lower\n\tRandomBounds 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();
  }
} ///:~
```

要运行这个程序，你得在命令行下输入：

```
java RandomBounds lower
```

或

```
java RandomBounds upper
```

无论你运行哪条命令，最后都不得不手动停下程序，所以看上去**Math.random( )**既不会产生 0.0 也不会生成 1.0。但是这是这个实验的欺骗性的一面。如果你知道，[18]0 到 1 之间有 2^{62} 个不同的值，所以要用试验来撞某个值，可能会耗尽计算机，甚至是作试验的人的寿命。告诉你，**Math.random( )**会生成 0.0，或者用数学术语，它的值域是[0,1)。

总结

这一章讲述了绝大多数的编程语言都有的基本特性：计算，操作符优先级，类型转换，选择与循环。现在你已经做好准备了，能够进一步探索面向对象编程的世界了。下一章，我们会讲解隐藏实现，然后是对象的初始化，和清理之类的重大问题。

练习

只要付很小一笔费用就能从 *www.BruceEckel.com* 下载名为 *The Thinking in Java Annotated Solution Guide* 的电子文档，这上面有一些习题的答案。

1. 在本章前面部分的“优先级”一节中有两条表达式。把这两条表达式放到程序里去，以印证他们会产生不同的结果。
2. 把 **ternary( )**和 **alternative( )**方法做成一个能运行的程序。
3. 修改“if-else”和“return”两节的两个 **test( )**方法，让它判断 **testval** 是不是在参数 **begin** 和 **end** 之间(包括)。
4. 写一个会打印从 1 到 100 的值的程序。
5. 用 **break** 语句修改练习 4 的程序，让它在 47 的时候退出。再试着用 **return** 来代替 **break**。
6. 写一个拿两个 **String** 做参数的方法，用各种的 **boolean** 方法比较这两个 **String**，然后把结果打印出来。做 == 和 **!=**比较的时候，还要用 **equals( )**作比较。在 **main( )**里面用几个不同的 **String** 调用这个方法。
7. 写一个会随机生成 25 个 **int** 值的程序。用 **if-else** 语句判断，这个值是大于，小于，还是等于下一个随机生成的值。
8. 修改练习 7 的程序，用“无限循环”的 while 语句来包裹这段程序。这样，你就只能用敲键盘来中断程序了(通常就是 Ctronl-C)。
9. 写一个嵌套两层 **for** 循环的程序。用模运算符(**%**)来检测质数，然后把它打印出来。(所谓质数是指只能被 1 和它自己整除的自然数)。

10. 写一段 **switch**，让它为每个 **case** 打印一个消息。然后把这个 **switch** 放进 **for** 循环，以此测试每个 **case**。先测试每个 **case** 后面都有 **break** 的程序，然后把 **break** 删了，再看看结果会怎样。

---

[17] John Kirkham 写道，“我从 1962 年开始在 IBM 1620 上用 FORTRAN II 做计算。那时，甚至是整个 1960 和 1970 年代，FORTRAN 都是一种全部大写的语言。可能这是因为早期的输入设备都是老式的电传设备，使用 5 位的 Baudot 编码，还没有使用小写字母的能力。表示指数的‘E’也是大写的，因此从来就没有同自然对数的‘e’混淆过。‘E’只是简单的表示指数，至于其底数就是我们的计数体系所用的 10。那时程序员也经常使用 8 进制。尽管我从来没看到有人这么用过，但是如果我看到指数里面有 8 进制的数的话，我会认为它是以 8 为底的。我记得第一次看到指数使用小写的‘e’是在 1970 年代，当时我觉得非常困惑。这个问题是因为 FORTRAN 里面有了小写字母才引起的，它不是一开始就有的。实际上如果要表示自然对数的话，我们都是用函数的，只不过也是大写字母的。”

[18] Chuck Allison 写道：浮点数系统能表示的精度是
2(M-m+1)b^(p-1) + 1
其中 **b** 是底数(通常是 2)，**p** 指精度(10 进制的尾数)，**M** 是最大的指数，而 **m** 是最小的指数。IEEE 754 使用：
M = 1023, m = -1022, p = 53, b = 2
所以这个小数可以有的位数是
2(1023+1022+1)2^52
= 2((2^10-1) + (2^10-1))2^52
= (2^10-1)2^54
= 2^64 - 2^54
这些数字的一半(用科学表示法表示，其指数范围在[-1022,0])，会比 1 小(既包括正数也包括负数)，所以这个表达式的 1/4，也就是 2^62 - 2^52 + 1(大约 2^62)，会在[0,1)之间。看看我在 http://www.freshsources.com/1995006a.htm 发表的文章(文章的最后)。