第2章

数据类型与运算符

关键技能与概念

- 了解 Java 的基本类型
- 使用字面值
- 初始化变量
- 了解方法中变量的作用域原则
- 使用算术运算符
- 使用关系运算符和逻辑运算符
- 理解赋值运算符
- 使用赋值速记符
- 理解赋值中的类型转换
- 不兼容类型的强制转换
- 理解表达式中的类型转换

任何程序设计语言的基础都是它的数据类型和运算符，Java 也不例外。这些元素定义了语言的限制，并且决定了何种任务可以使用它。Java 支持种类众多的数据类型与运算符，所以它适用于任何类型的程序设计。

数据类型和运算符是十分庞大的话题。我们将从 Java 的基本数据类型和最常用的运算符开始介绍。同时，还将详细介绍变量和表达式。

2.1 数据类型为什么重要

在 Java 中，数据类型特别重要的原因在于 Java 是一门强类型语言。这意味着所有操作都要被编译器进行类型检查以确保类型的兼容性。非法操作将不被编译。因此，强类型检查有助于防止错误发生，增强了可靠性。为实现强类型检查，所有的变量、表达式和值都必须有自己的类型。例如，没有“无类型”变量这样的概念。不仅如此，值的类型还可以决定允许对该值执行什么样的操作。允许对一种类型执行的操作可能不适用于另一种类型。

2.2 Java 的基本类型

Java 包含两类基本的内置数据类型：面向对象类型和非面向对象类型。Java 的面向对象类型由类定义。稍后将对类进行讨论。然而 Java 的核心是 8 种基本(也称为简单)数据类型，如表 2-1 所示。术语“基本”用在这里表示这些类型不是面向对象意义中的对象，而是普通的二进制值。这些基本类型之所以不是对象，是考虑到效率的原因。Java 的所有其他类型都从这些基本类型构造而来。

Java 严格指定了每种基本类型的范围与行为，所有 Java 虚拟机的实现都必须提供对这些类型的支持。由于 Java 可移植性的需要，这一点是绝不能妥协的。例如，在所有执行环境中 int 都必须一样。这就使程序是完全可移植的，也就无须为适应某个特定的平台而重写代码了。尽管在某些环境下，严格指定基本类型的大小会导致性能上小的损失，但这是实现可移植性所必需的。

表 2-1 Java 内置的基本数据类型

类 型	含 义	类 型	含 义
boolean	表示 true/false 值	float	单精度浮点数
byte	占用 8 个二进制位的整数	int	整数
char	字符	long	长整数
double	双精度浮点数	short	短整数

2.2.1 整数类型

Java 提供 4 种整数类型：byte、short、int 和 long，如表 2-2 所示。

表 2-2 4 种整数类型

类 型	占用的二进制位数	取值范围
byte	8	–128～127
short	16	–32768～32767
int	32	–2147483648～2147483647
long	64	–9223372036854775808～9223372036854775807

如表 2-2 所示，所有整数类型都有正负值之分。Java 不支持无符号(只为正的)整数。其他许多语言都支持有符号整数和无符号整数，然而，Java 的设计者感觉无符号整数是不必要的。

注意：

从技术角度看，Java 运行时系统可以使用任意大小的空间来存储基本类型，然而在任何情况下，类型都必须按照指定的规则工作。

最常用的整数类型是 int。int 类型的变量常应用于循环控制、数组索引，以及执行常规的整数数学运算。

当需要的整数范围超过 int 的取值范围时，就使用 long。例如，下面的程序使用 long 类型计算一立方英里的立方体中包含多少立方英寸：

```
/*
   Compute the number of cubic inches
   in 1 cubic mile.
*/
class Inches {
  public static void main(String args[]) {
    long ci;
    long im;

    im = 5280 * 12;

    ci = im * im * im;

    System.out.println("There are " + ci +
                       " cubic inches in cubic mile.");

  }
}
```

程序的输出如下：

```
There are 254358061056000 cubic inches in cubic mile.
```

很明显，结果已经超出 int 变量的取值范围。

最小的整数类型是 byte。在处理无法直接和 Java 的其他内置类型兼容的原始二进制数据时，byte 类型的变量特别有用。short 类型可以创建短整数。当不需要 int 那么大的取值范围时，可以使用 short 类型的变量。

专家解答

问： 你提到有 4 种整数类型：int、short、long 和 byte。但是，我听说在 Java 中，char 也可以归入整数类型。你能解释原因吗？

答： Java 的正式规范定义了名为整数类型的类别，其中包含 byte、short、int、long 和 char。之所以称为整数类型，是因为它们都保存整数的二进制值。但是，前 4 种类型的目的是表示整数的数字量，而 char 则用于表示字符。因此，char 的主要用途和其他整数类型的主要用途存在根本区别。正因为如此，本书中区别对待 char 类型。

2.2.2　浮点型

如第 1 章所述，浮点型可以表示有小数部分的数值。浮点型有两种：float 和 double，分别代表单精度和双精

度数值。float 类型为 32 位，double 类型为 64 位。

这两种类型中最常用的是 double，因为 Java 类库中的所有数学函数都使用 double 值。例如 sqrt( )方法(由标准 Math 类定义)返回一个 double 值，这个值是它的 double 实参的平方根。下面的程序给定直角三角形的两个直角边长度，使用 sqrt( )计算斜边的长度：

```
/*
  Use the Pythagorean theorem to
  find the length of the hypotenuse
  given the lengths of the two opposing
  sides.
*/
class Hypot {
  public static void main(String args[]) {
    double x, y, z;

    x = 3;
    y = 4;

    z = Math.sqrt(x*x + y*y);

    System.out.println("Hypotenuse is " +z);
  }
}
```

注意 sqrt( )方法是如何被调用的，我们前置了它所属类的名称

程序的输出如下所示：

```
Hypotenuse is 5.0
```

关于上述示例还有一点说明：如前所述，sqrt( )是标准 Math 类的一个成员。注意调用 sqrt( )的方式，sqrt( )跟在类名 Math 的后面。这与 println( )跟在 System.out 后面的方式相似。尽管不是所有的标准方法都要通过指定它们所属类的类名来调用，但有些方法还是需要的。

2.2.3 字符型

在 Java 中，字符不像在其他计算机语言中那样占用 8 个二进制位，Java 使用的是 Unicode。Unicode 定义了一个字符集合，该集合可以表示所有人类语言中的字符。因此在 Java 中，char 是无符号 16 位类型，取值范围为 0～65 536。标准的 8 位 ASCII 字符集是 Unicode 的子集，取值范围为 0～127。因此，ASCII 字符依然是有效的 Java 字符。

字符变量可由一对单引号中的字符赋值。例如，下面就是将字母 X 赋予变量 ch 的示例：

```
char ch;
ch = 'X';
```

可使用 println( )语句输出字符值。例如，下面这行语句输出了 ch 中的值：

```
System.out.println("This is ch: " + ch);
```

因为 char 是无符号 16 位类型，所以可以对 char 变量进行多种算术运算。例如，考虑下面的程序：

```
// Character variables can be handled like integers.
class CharArithDemo {
  public static void main(String args[]) {
    char ch;
```

```
    ch = 'X';
    System.out.println("ch contains " + ch);

    ch++; // increment ch                ← 递增char
    System.out.println("ch is now " + ch);

     ch = 90; // give ch the value Z     ← 可以给char赋予整数值
    System.out.println("ch is now " + ch);
  }
}
```

程序生成的输出如下所示：

```
ch contains X
ch is now Y
ch is now Z
```

在该程序中，ch 首先被赋值为 X。接着递增 ch。这样它的结果就成了对应 ASCII(和 Unicode)序列中的下一个字符 Y。接着，ch 被赋值为 90，这个 ASCII(和 Unicode)值对应的是字母 Z。因为 ASCII 码占用 Unicode 中的前 127 个值，所以过去使用其他语言时应用于字符的一些技巧依然可在 Java 中使用。

专家解答

问： Java 为什么使用 Unicode?

答： Java 的设计目标是在全世界使用。因此，需要使用可以表示全世界语言的字符集。Unicode 就是为这一目标而设计的标准字符集。当然，在用于诸如英语、德语、西班牙语或法语的“字符可以包括在 8 个二进制位中”的语言时，使用 Unicode 的效率要低一些。但这就是为实现全球可移植性所付出的代价。

2.2.4　布尔类型

布尔(boolean)类型表示真/假(true/false)值。Java 使用保留字 true 和 false 来定义真值和假值。因此，布尔类型的变量或表达式只能取这两个值中的一个。

下面是一个用于说明布尔类型的程序：

```
// Demonstrate boolean values.
class BoolDemo {
  public static void main(String args[]) {
    boolean b;

    b = false;
    System.out.println("b is " + b);
    b = true;
    System.out.println("b is " + b);

    // a boolean value can control the if statement
    if(b) System.out.println("This is executed.");

    b = false;
    if(b) System.out.println("This is not executed.");

    // outcome of a relational operator is a boolean value
    System.out.println("10 > 9 is " + (10 > 9));
```

```
  }
}
```

程序生成的输出如下所示：

```
b is false
b is true
This is executed.
10 > 9 is true
```

这个程序有三个有趣的地方值得注意。第一，可以看到，当布尔值被 println( )输出时显示的是 true 或 false。第二，布尔变量的值本身就可以控制 if 语句，而无须再像下面这样编写 if 语句：

```
if(b == true) ...
```

第三，关系运算符(如“<”)的结果是一个布尔值。这就是表达式 10 > 9 显示的值是 true 的原因。还有，包含 10 > 9 的一组括号是必需的，因为运算符“+”的优先级高于“>”。

练习 2-1(Sound.java) 闪电有多远？

在本练习中，将创建一个程序来计算听到打雷的人距离闪电有多少英尺。声音在空气中的传播速度约为每秒 1100 英尺。因此，当得知看到闪电的时间与听到雷声的时间之差后，就可以计算听者与闪电之间的距离了。就本练习而言，时间差为 7.2 秒。

步骤：

(1) 创建一个名为 Sound.java 的新文件。

(2) 计算距离需要使用浮点值。为什么呢？因为时间差 7.2 秒有小数部分。尽管可以使用 float 类型，但该例使用 double 类型。

(3) 计算距离，需要用 7.2 乘以 1100。然后，把这个值赋给一个变量。

(4) 最后显示结果。

下面是整个 Sound.java 程序的清单：

```
/*
  Try This 2-1
  Compute the distance to a lightning
  strike whose sound takes 7.2 seconds
  to reach you.
*/
class Sound {
  public static void main(String args[]) {
    double dist;

    dist = 7.2 * 1100;

    System.out.println("The lightning is " + dist +
                       " feet away.");

  }
}
```

(5) 编译并运行该程序。结果如下所示：

```
The lightning is 7920.0 feet away.
```

附加练习：通过测量回声的时间，可以计算与大型物体(如岩壁)之间的距离。例如，拍手并得到听到回声的时间，得到的是来回的时间。将该值除以 2，得到声音在一个方向所花费的时间。然后，就可以利用该值计算到物体的距离了。假设时间差是回声的时间，修改前面的程序以计算距离。

2.3　字面值

在 Java 中，字面值(literal)是指以人类可读的语言形式出现的固定值。例如，数值 100 就是一个字面值。字面值也常称为常量(constant)。大部分字面值和它们的用法都是非常直观的，前面的示例程序已经以各种形式使用它们了。现在正式地解释它们。

Java 字面值可以是任何基本数据类型。每一个字面值的表示方式都依赖于它的类型。如前所述，字符常量是包含在单引号中的。例如，'a'和'%'都是字符常量。

整数字面值被指定为没有小数部分的数。例如，10和–100是整数字面值。浮点字面值需要在小数部分的前面使用小数点。例如，11.123 就是一个浮点字面值。Java 也允许对浮点数使用科学记数法。

默认情况下，整数字面值是 int 类型。如果想指定 long 类型的字面值，就附加一个 l 或 L。例如，12 是 int 类型，而 12L 就是 long 类型。

默认情况下，浮点字面值是 double 类型。为指定 float 字面值，可在常量后附加一个 f 或 F。例如，10.19F 就是 float 类型。

尽管默认情况下整数字面值创建的是一个 int 值，但是仍可以把它们赋给诸如 char、byte 或 short 类型的变量，只要该值可以由这些类型表示即可。整数字面值总是可以赋给 long 变量。

可以在整数或浮点字面值中嵌入一条或多条下画线，以方便阅读包含多个数位的值。编译字面值时将丢弃下画线。下面给出一个例子：

```
123_45_1234
```

上面这个值为 123 451 234。当编码零件号、客户 ID 和状态码这类通常包含多个数字子组的值时，使用下画线特别有用。

2.3.1　十六进制、八进制和二进制字面值

众所周知，在程序设计中使用基于 8 或 16 的数字系统比使用基于 10 的数字系统要简单。基于 8 的数字系统称为八进制，使用的数字是 0～7。在八进制中，数字 10 等同于十进制中的 8。基于 16 的数字系统称为十六进制，使用的数字是 0～9，再加上字母 A～F，这几个字母分别代表 10、11、12、13、14 和 15。例如，十六进制数 10 在十进制中等同于 16。因为这两个数字系统要经常使用，所以 Java 允许指定十六进制或八进制的整数字面值来取代十进制的整数字面值。十六进制字面值必须以 0x 或 0X 开始(数字 0 后跟一个 x 或 X)，而八进制字面值要以 0 开始。下面是几个示例：

```
hex = 0xFF; // 255 in decimal
oct = 011; // 9 in decimal
```

有趣的是，Java 还允许指定十六进制浮点字面值，只是它们很少使用。

可以使用二进制数字指定整数字面值。这需要在二进制数字的前面加上 0b 或 0B。例如，二进制形式的 0b1100 就是值 12。

2.3.2 字符转义序列

对于多数可打印字符而言，将字符常量包含在单引号中是可以正常工作的，但是对于一些字符，如回车换行符，在使用文本编辑器时就会产生问题。此外，某些其他字符(如单引号、双引号)在 Java 中也有特殊意义。所以，也不能直接使用它们。出于这些原因，Java 提供了特殊的转义序列(escape sequence)，有时称为反斜杠字符常量，如表 2-3 所示。这些转义序列用于替代它们所代表的字符。

表 2-3 字符转义序列

转义序列	描 述	转义序列	描 述
\'	单引号	\n	换行符
\"	双引号	\f	换页符
\\	反斜杠	\t	水平制表符
\r	回车符	\b	退格符
\ddd	八进制常量符号(*ddd* 是八进制常量)	*\uxxxx*	十六进制常量符号(*xxxx* 是十六进制常量)

例如，下面的代码将 tab 字符赋给 ch：

```
ch = '\t';
```

下一个示例将一个单引号赋给 ch：

```
ch = '\'';
```

2.3.3 字符串字面值

Java 支持另一种类型的字面值，即字符串(string)。字符串是包含在双引号内的一组字符。例如

```
"this is a test"
```

就是一个字符串。在前面的示例程序中，你已经在语句 println( )中看到了许多字符串示例。

除了普通字符以外，字符串字面值也可包含一个或多个前面讲到的转义序列。例如，考虑下面的问题，它使用\n 和\t 转义序列。

```
// Demonstrate escape sequences in strings.
class StrDemo {
  public static void main(String args[]) {
    System.out.println("First line\nSecond line");
    System.out.println("A\tB\tC");
    System.out.println("D\tE\tF") ;
  }
}
```

使用\n 换行

使用制表符对齐输出

输出如下所示：

```
First line
Second line
A       B       C
D       E       F
```

专家解答

问： 由单个字符组成的字符串与字符字面值一样吗？例如，"k"与'k'一样吗？

答： 不一样。切勿混淆字符串与字符。字符字面值表示 char 类型的字母。字符串哪怕只包含一个字母也是字符串。尽管字符串是由字符组成的，但它们是不同的类型。

注意\n 转义序列是如何用于生成新行的。不需要使用多条 println( )语句来得到多行输出，只需要将\n 嵌入较长的字符串中想生成新行的位置即可。

2.4　变量详解

变量在第 1 章就提到了。这里对它再做进一步介绍。如前所述，变量使用下面的语句来声明：

```
type var-name;
```

type 是变量的数据类型，var-name 是变量的名称。可以声明任何有效类型的变量，包括前面描述的简单类型在内，并且每个变量都有类型。因此，变量的功能是由其类型决定的。例如，boolean 类型的变量无法存储浮点值。而且变量的类型在其生命期内不能更改。例如，int 类型的变量不能变为 char 类型的变量。

Java 中的所有变量都必须在使用前声明。这是必需的，因为编译器必须在正确编译使用变量的任何语句前知道变量的数据类型是什么。这也使 Java 执行严格类型检查成为可能。

2.4.1　初始化变量

在使用变量之前必须赋给它一个值。使用赋值语句是为变量赋值的方法之一。另一种方法就是在声明变量时，赋给变量一个初值。为此，需要在变量名的后面添加一个等号，后跟一个值。初始化的基本形式如下所示：

```
type var = value;
```

value 是创建 *var* 时赋给它的值。该值必须与指定的类型兼容。下面是几个示例：

```
int count = 10; // give count an initial value of 10
char ch = 'X'; // initialize ch with the letter X
float f = 1.2F; // f is initialized with 1.2
```

当使用逗号分隔列表来声明同一类型的两个或多个变量时，可以赋给这些变量一个或多个初值。例如：

```
int a, b = 8, c = 19, d; // b and c have initializations
```

本例中，只有 b 和 c 被初始化。

2.4.2　动态初始化

虽然前面的示例只使用常量进行初始化，但是 Java 允许在声明变量时，使用任何有效的表达式来动态初始化变量。例如，下面是一个计算圆柱体体积的小程序，其中已经给出底圆的半径和圆柱体的高。

```
// Demonstrate dynamic initialization.
class DynInit {
   public static void main(String args[]) {
     double radius = 4, height = 5;
```

```
    // dynamically initialize volume
    double volume = 3.1416 * radius * radius * height;

    System.out.println("Volume is " + volume);
  }
}
```

volume 在运行时被动态初始化

这里声明了三个局部变量：radius、height 和 volume。其中 radius 和 height 被初始化为常量，而 volume 则被动态初始化为圆柱体的体积。这里的关键之处在于，初始化表达式在初始化时可以使用任何有效元素，包括调用方法、其他变量和字面值。

2.5 变量的作用域和生命期

目前，我们使用的变量都是在 main( )方法的开始处声明的。然而 Java 允许变量在任何代码块中声明。如第 1 章所述，代码块以左花括号开始，以右花括号结束。代码块定义了作用域(scope)。因此，每次创建新的代码块时，就是在创建新的作用域。作用域决定了对于程序的其他部分哪些对象是可见的，还决定了这些对象的生命期。

通常，Java 中的每个声明都有一个作用域。因此，Java 定义了一个强大的、细粒度的“作用域”概念。在 Java 中最常用的作用域是由类和方法定义的作用域。关于类作用域(以及在其中声明的变量)的讨论将放在本书后面介绍类时进行。现在我们只讨论方法中的作用域或由方法定义的作用域。

方法定义的作用域从方法的左花括号开始。然而，如果方法有形参，那么这些形参也包含在方法的作用域之内。方法定义的作用域以右花括号结束。将这个代码块称为方法体。

作为一项基本原则，在作用域内声明的变量对于作用域之外定义的代码是不可见(即不可访问)的。因此，当在作用域内声明一个变量时，就把这个变量局部化了，并且防止它受到未授权的访问或修改。实际上，正是作用域规则提供了封装的基础。在代码块中声明的变量称为局部变量。

作用域是可嵌套的。例如每次创建代码块都会创建新的、嵌套的作用域。当这种情况发生时，外层作用域包含内层作用域。这意味着在外层作用域中定义的对象对于内层作用域中的代码是可见的；反之则不然，在内层作用域中定义的对象对于外层作用域中的代码是不可见的。

考虑下面的程序来理解嵌套作用域的作用：

```
// Demonstrate block scope.
class ScopeDemo {
  public static void main(String args[]) {
    int x; // known to all code within main

    x = 10;
    if(x == 10) { // start new scope

      int y = 20; // known only to this block

      // x and y both known here.

      System.out.println("x and y: " + x + " " + y);
      x = y * 2;
    }
    // y = 100; // Error! y not known here

    // x is still known here.
    System.out.println("x is " + x);
  }
}
```

y 在作用域之外

如代码注释所示，变量 x 是在 main( )作用域的开始处声明的，对于 main( )中其后的所有代码都是可访问的。我们在 if 代码块中声明了 y。因为代码块可定义作用域，所以 y 只对代码块中的其他代码可见。这就是代码块以外的 y=100;这一行被注释掉的原因。如果删除前面的注释符号，就会发生编译时错误，因为 y 对于所在代码块外面的代码是不可见的。在 if 代码块中可以使用 x，因为代码块(即嵌套作用域)中的代码有权访问在被嵌套作用域中声明的变量。

在代码块中的任何位置都可以声明变量，而变量只有在声明后才有效。因此，如果在某个方法的开始处定义了一个变量，那么它对于方法中的所有代码都是有效的。相反，如果在代码块的结尾处声明了一个变量，那么它是没有任何作用的，因为没有代码可以访问它。

此外，还有一个要点必须记住：变量在进入作用域时创建，在离开作用域时销毁。这就意味着变量一旦离开作用域就不会存储原来的值。因此，在方法调用期间，在方法中声明的变量是无法保存其值的，而且在代码块中声明的变量在离开此代码块时也将失去它的值。因此，变量的生命期被限制在作用域内。

如果变量声明包含初始化值，那么在每次进入声明它的代码块时都会重新初始化，如下面的程序所示：

```
// Demonstrate lifetime of a variable.
class VarInitDemo {
  public static void main(String args[]) {
    int x;
    for(x = 0; x < 3; x++) {
      int y = -1; // y is initialized each time block is entered
      System.out.println("y is: " + y); // this always prints -1
      y = 100;
      System.out.println("y is now: " + y);
    }
  }
}
```

该程序生成的输出如下所示：

```
y is: -1
y is now: 100
y is: -1
y is now: 100
y is: -1
y is now: 100
```

可以看出，在每次进入内部 for 循环时，y 都会重新初始化为-1。尽管 y 在后面被赋值为 100，但该值无法保存。

Java 作用域有一条可能会令人吃惊的奇怪规则：尽管可以嵌套代码块，但是在内部作用域中声明的变量不能与包含它的作用域中已经声明的变量同名。例如，下面的程序试图用同一名称声明两个变量，该程序无法编译：

```
/*
   This program attempts to declare a variable
   in an inner scope with the same name as one
   defined in an outer scope.

   *** This program will not compile. ***
*/
class NestVar {
  public static void main(String args[]) {
    int count;

    for(count = 0; count < 10; count = count+1) {
      System.out.println("This is count: " + count);

      int count; // illegal!!!
```

不能再次声明 count

因为它已被声明

```
      for(count = 0; count < 2; count++)
        System.out.println("This program is in error!");
    }
  }
}
```

2.6 运算符

Java 提供了丰富的运算符环境。运算符(operator)是告知编译器执行特定数学或逻辑操作的符号。Java 有 4 种基本运算符类型：算术运算符、位运算符、关系运算符和逻辑运算符。此外，Java 还定义了一些处理某种特殊情况的运算符。本章将介绍算术运算符、关系运算符和逻辑运算符，另外还将介绍赋值运算符。位运算符和其他特殊运算符将在后面介绍。

2.7 算术运算符

Java 定义的算术运算符如表 2-4 所示。

表 2-4 Java 定义的算术运算符

运 算 符	含 义	运 算 符	含 义
+	加(也称为一元加)	%	求余
–	减(也称为一元减)	++	自增
*	乘	––	自减
/	除		

Java 中的+、–、*和/运算符与其他计算机语言(或代数)中的一样。这些运算符可应用于任何内置数值数据类型，也可应用于 char 类型的对象。

尽管所有读者都非常熟悉算术运算符的操作，但由于存在一些特殊情况，对此还是值得解释一下。首先，当/用于整数时，任何余数将被删除。例如，在整除中 10/3 等于 3。使用求余运算符%可获得整除的余数。得出的是整除的余数。例如，10 % 3 等于 1。在 Java 中，%可应用于整数类型和浮点类型。因此 10.0 % 3.0 也是 1。下面的程序说明了求余运算符的用法：

```
// Demonstrate the % operator.
class ModDemo {
  public static void main(String args[]) {
    int iresult, irem;
    double dresult, drem;

    iresult = 10 / 3;
    irem = 10 % 3;

    dresult = 10.0 / 3.0;
    drem = 10.0 % 3.0;

    System.out.println("Result and remainder of 10 / 3: " +
                       iresult + " " + irem);
    System.out.println("Result and remainder of 10.0 / 3.0: " +
                       dresult + " " + drem);

  }
}
```

该程序的输出如下所示：

```
Result and remainder of 10 / 3: 3 1
Result and remainder of 10.0 / 3.0: 3.3333333333333335 1.0
```

可以看到，对于整数类型和浮点类型操作，%都得到了余数 1。

自增和自减

第 1 章介绍过，++和--分别是 Java 的自增和自减运算符。它们的一些特殊属性使其变得十分有趣。我们首先详细回顾一下自增和自减运算符的作用。

自增运算符对操作数加 1，而自减运算符对操作数减 1。因此，下面两条语句的效果是一样的：

```
x = x + 1;
x++;
```

而下面这两条语句的效果也是一样的：

```
x = x - 1;
x--;
```

自增和自减运算符既可放在操作数的前面(前缀)，也可放在操作数的后面(后缀)。例如：

```
x = x + 1;
```

可以写成：

```
++x; // 前缀形式
```

也可以写成：

```
x++; // 后缀形式
```

在前述示例中，自增运算符作为前缀或后缀是没有区别的。然而，当自增或自减运算符作为较大表达式的一部分时，情况就大相径庭了。当自增或自减运算符位于操作数之前时，Java 会在表达式的其余部分使用操作数之前先对相应的操作数进行运算。如果自增或自减运算符在操作数之后，Java 会先将操作数的值用于表达式，然后进行自增或自减运算。考虑下面的示例：

```
x = 10;
y = ++x;
```

本例中，y 的值为 11。然而，如果代码如下所示：

```
x = 10;
y = x++;
```

y 的值将会是 10。这两个示例中，x 的值都为 11。不同之处在于何时进行自增运算。能够控制自增或自减操作发生的时间具有显著优势。

2.8　关系运算符和逻辑运算符

在术语关系运算符(relational operator)和逻辑运算符(logical operator)中，关系指值与值之间的相互关系，逻辑指将真值和假值连接在一起的方式。关系运算符产生的结果是真或假，所以它们经常与逻辑运算符一起使用。出于这一原因，这里对这两种运算符一起讨论。

关系运算符如表 2-5 所示。

表 2-5 关系运算符

运 算 符	含 义	运 算 符	含 义
==	等于	<	小于
!=	不等于	>=	大于或等于
>	大于	<=	小于或等于

逻辑运算符如表 2-6 所示。

表 2-6 逻辑运算符

运 算 符	含 义	运 算 符	含 义
&	与(AND)	||	短路或(short-circuit OR)
|	或(OR)	&&	短路与(short-circuit AND)
^	异或(XOR)	!	非(NOT)

关系运算符与逻辑运算符的结果是 boolean 类型的值。

在 Java 中，所有对象都可以使用==和!=进行等于或不等于比较。然而，比较运算符<、>、<=或>=则只能用于支持顺序关系的类型。因此，所有关系运算符都可用于数值类型和 char 类型。然而，boolean 类型的值只可以用于进行等于或不等于比较，因为 true 和 false 值是无序的。例如，在 Java 中 true > false 是无意义的。

对于逻辑运算符，操作数必须是 boolean 类型，逻辑运算的结果也必须是 boolean 类型。逻辑运算符&、|、^和!按照表 2-7 所示的真值表执行基本的逻辑运算 AND、OR、XOR 和 NOT。

表 2-7 逻辑运算符的真值表

p	q	p & q	p | q	p ^ q	!p
false	false	false	false	false	true
true	false	false	true	true	false
false	true	false	true	true	true
true	true	true	true	false	false

如表 2-7 所示，只有一个操作数为真时，异或(exclusive OR)运算的结果为真。

下面的程序对这几个关系运算符和逻辑运算符进行了演示：

```
// Demonstrate the relational and logical operators.
class RelLogOps {
  public static void main(String args[]) {
    int i, j;
    boolean b1, b2;

    i = 10;
    j = 11;
    if(i < j) System.out.println("i < j");
    if(i <= j) System.out.println("i <= j");
    if(i != j) System.out.println("i != j");
    if(i == j) System.out.println("this won't execute");
    if(i >= j) System.out.println("this won't execute");
    if(i > j) System.out.println("this won't execute");

    b1 = true;
    b2 = false;
    if(b1 & b2) System.out.println("this won't execute");
    if(!(b1 & b2)) System.out.println("!(b1 & b2) is true");
    if(b1 | b2) System.out.println("b1 | b2 is true");
```

```
    if(b1 ^ b2) System.out.println("b1 ^ b2 is true");
  }
}
```

该程序的输出如下所示：

```
i < j
i <= j
i != j
!(b1 & b2) is true
b1 | b2 is true
b1 ^ b2 is true
```

2.9 短路逻辑运算符

Java 支持用于生成高效代码的短路(short-circuit)AND 和 OR 逻辑运算符。为什么这样设计呢？在 AND 运算中，如果第一个操作数为假，那么无论第二个操作数是什么，运算结果都为假。在 OR 运算中，如果第一个操作数为真，那么无论第二个操作数是什么，运算结果都为真。因此，这些情况下就无须对第二个操作数进行计算。不计算第二个操作数，就节省了时间，从而生成效率更高的代码。

短路 AND 运算符是&&，短路 OR 运算符是||。而普通的 AND 和 OR 运算符分别是&和|。两者的唯一区别在于普通运算符将计算每一个操作数，而短路运算符只在必要时计算第二个操作数。

下面是一个演示短路 AND 运算符的程序。程序要确定 d 中的值是不是 n 的因数。程序利用求余运算来完成这一功能。如果 n/d 的余数为 0，那么 d 是因数。然而，因为求余运算涉及除法，短路 AND 运算可防止发生除 0 错误。

```
// Demonstrate the short-circuit operators.
class SCops {
  public static void main(String args[]) {
    int n, d, q;

    n = 10;
    d = 2;
    if(d != 0 && (n % d) == 0)
      System.out.println(d + " is a factor of " + n);

    d = 0; // now, set d to zero

    // Since d is zero, the second operand is not evaluated.
    if(d != 0 && (n % d) == 0) <-------- 短路运算符能防止除 0 操作
      System.out.println(d + " is a factor of " + n);

    /* Now, try same thing without short-circuit operator.
       This will cause a divide-by-zero error.
    */
     if(d != 0 & (n % d) == 0) <-------- 这里计算两个表达式，允许除 0 操作
      System.out.println(d + " is a factor of " + n);
  }
}
```

为防止出现除 0 错误，if 语句首先检查 d 是否等于 0。如果等于 0，短路 AND 运算结束，不执行求余运算。因此，在第一个条件测试中 d 是 2，执行求余运算。第二个条件测试失败，因为 d 为 0，求余运算被跳过，以免发生除 0 错误。最后一个条件测试使用了普通 AND 运算符。这样会计算两个操作数，在除以 0 时会导致运行时错误的发生。

最后要注意的一点是，Java 的正式规范将短路运算符称为“条件或”和“条件与”运算符，但通常使用“短路”这个词。

2.10 赋值运算符

我们从第 1 章就开始使用赋值运算符了。现在对其进行正式介绍。赋值运算符(assignment operator)是一个等号。这个运算符在 Java 中与在其他语言中的使用方式相同。基本形式如下：

```
var = expression;
```

这里，var 的类型必须与表达式(expression)的类型兼容。

赋值运算符有一个读者可能不太熟悉的有趣属性：它允许创建一个赋值链。例如下面的代码段：

```
int x, y, z;
x = y = z = 100; // set x, y, and z to 100
```

这个代码段使用一条语句将变量 x、y 和 z 的值设置为 100。因为=是一个得到右侧表达式值的运算符，所以上面的代码段是可以运行的。因此，z = 100 的值是 100，然后该值被赋给 y，y 再将值赋给 x。使用“赋值链”是为一组变量设置相同值的捷径。

2.11 速记赋值

Java 提供特殊的速记赋值来简化某些赋值语句的编码。下面首先列举一个示例。使用 Java 的速记赋值可将赋值语句

```
x = x + 10;
```

写为：

```
x += 10;
```

专家解答

问：既然在某些情况下，短路运算符比对应的普通运算符效率更高，Java 为何还要提供普通的 AND 和 OR 运算符？

答： 某些情况下，因为会出现副作用，所以需要同时计算 AND 或 OR 运算的两个操作数，如下所示：

```
// Side effects can be important.
class SideEffects {
  public static void main(String args[]) {
    int i;

    i = 0;

    /* Here, i is still incremented even though
       the if statement fails. */
    if(false & (++i < 100))
      System.out.println("this won't be displayed");
    System.out.println("if statement executed: " + i); // displays 1

    /* In this case, i is not incremented because
       the short-circuit operator skips the increment. */
    if(false && (++i < 100))
      System.out.println("this won't be displayed");
    System.out.println("if statement executed: " + i); // still 1 !!
  }
}
```

如注释所示，在第一个 if 语句中，无论 if 语句真假与否，i 都会自增。而在使用短路运算符时，当第一个操作数为 false 时，变量 i 就不会自增。这里的教训就是，如果代码希望计算 AND 或 OR 运算右侧的操作数，就必须使用 Java 的非短路逻辑运算符。

运算符+=让编译器将 x 加 10 的值赋给 x。这里还有一个示例。下面的两条语句是等效的，这两条语句都将 x 减 100 后的值赋给 x：

```
x = x - 100;
```

等同于：

```
x -= 100;
```

速记赋值可以用于 Java 的所有二元运算符(即需要两个操作数的运算符)。速记赋值的基本形式为：

```
var op = expression;
```

因此，算术和逻辑赋值运算符的速记形式如下所示：

+=	-=	*=	/=
%=	&=	|=	^=

由于这些运算符把运算和赋值组合在一起，因此称为“组合赋值运算符”。

使用组合赋值运算符有两个优点。第一，它们比普通表示法更简洁。第二，在一些情况下它们的效率更高。由于这些原因，在专业的 Java 程序中我们经常会看到这些组合赋值运算。

2.12　赋值中的类型转换

在程序设计中，经常要将一种类型的变量赋值给另一种类型的变量。例如，可能要将 int 类型的值赋给 float 类型的变量，如下所示：

```
int i;
float f;

i = 10;
f = i; // assign an int to a float
```

当一条赋值语句中混合兼容类型时，等号右侧值的类型会自动转换为左侧值的类型。因此，在上面的代码段中，i 的值被转换为一个 float 类型的值，然后赋给变量 f。然而因为 Java 有严格的类型检查，所以不是所有类型都是相互兼容的，也就是说，并非所有的隐式类型转换都是允许的。例如，boolean 和 int 就是不兼容的。

将一种类型的数据赋给另一种类型的变量时，会发生自动类型转换的条件是：

(1) 两种类型兼容。

(2) 目标类型比源类型大。

当两个条件都满足时，发生扩展转换(widening conversion)。例如，int 类型足以包含所有有效的 byte 值，而 int 和 byte 又都是整数类型，所以可执行从 byte 到 int 的自动类型转换。

对于扩展转换，数值类型(包括整数类型和浮点类型在内)相互都是兼容的。例如，因为从 long 到 double 会自动执行扩展转换，所以下面的程序是完全有效的：

```
// Demonstrate automatic conversion from long to double.
class LtoD {
  public static void main(String args[]) {
    long L;
    double D;
```

```
    L = 100123285L;
    D = L;  <-- 从 long 到 double 的自动转换

    System.out.println("L and D: " + L + " " + D);

  }
}
```

尽管有从 long 到 double 的自动转换，但从 double 到 long 没有自动转换，因为这不是扩展转换。因此，下面的程序是无效的：

```
// *** This program will not compile. ***
class LtoD {
  public static void main(String args[]) {
    long L;
    double D;

    D = 100123285.0;
    L = D; // Illegal!!!  <-- 无法自动执行从 double 到 long 的转换

    System.out.println("L and D: " + L + " " + D);

  }
}
```

从数值类型到 char 或 boolean 没有自动类型转换，而且 char 和 boolean 之间也是不兼容的。但是，整数字面值可以赋给 char 类型的变量。

2.13 不兼容类型的强制转换

尽管自动类型转换是很有帮助的，但它们不能满足所有的程序设计需要，因为它们只能应用于兼容类型之间的扩展转换。对于其他情况，必须使用强制转换。强制转换(cast)是一条让编译器将一种类型转换为另一种类型的指令。因此，需要进行显式的类型转换。强制转换的基本形式是：

```
(target-type) expression
```

这里是将特定表达式转换为 target-type 指定的类型。例如，如果要将表达式 x/y 的类型转换为 int，可以编写如下代码：

```
double x, y;
// ...
(int) (x / y)
```

在此，即使 x 和 y 是 double 类型，强制转换也会将表达式的结果类型转换为 int。表达式 x / y 必须加圆括号。否则，只会对 x 执行强制转换，而不会对除法结果进行强制转换。因为没有从 double 到 int 的自动转换，所以强制转换是必需的。

当强制转换涉及缩减转换(narrowing conversion)时，一些信息可能会丢失。例如，当把 long 强制转换为 short 时，如果 long 类型的值超出 short 类型的范围，那么因为高序二进制位会被删除，所以会造成信息丢失。当把一个浮点值强制转换为整数时，小数部分也会由于被删除而丢失。例如，如果将 1.23 赋给一个整数，那么结果会是 1，丢失了 0.23。

下面的程序演示了几个需要强制执行的类型转换：

```
// Demonstrate casting.
class CastDemo {
  public static void main(String args[]) {
    double x, y;
    byte b;
    int i;
    char ch;

    x = 10.0;
    y = 3.0;

    i = (int) (x / y); // cast double to int    ← 该转换将丢失信息
    System.out.println("Integer outcome of x / y: " + i);

    i = 100;
    b = (byte) i;    ← 不会丢失信息，byte 可以存储值 100
    System.out.println("Value of b: " + b);

    i = 257;
    b = (byte) i;    ← 将会丢失信息，byte 不能存储值 257
    System.out.println("Value of b: " + b);

    b = 88; // ASCII code for X
    ch = (char) b;    ← 不兼容类型间的强制转换
    System.out.println("ch: " + ch);
  }
}
```

该程序的输出如下所示：

```
Integer outcome of x / y: 3
Value of b: 100
Value of b: 1
ch: X
```

在该程序中，从(x / y)到 int 的强制转换导致小数部分被截去，从而丢失信息。接下来，因为 byte 类型可以存储值 100，所以将值 100 赋给 b 时不会丢失信息。然而，当试图将值 257 赋给 b 时，就会丢失信息，因为 257 超出了 byte 类型的最大值。最后一条赋值语句没有信息会丢失，但当把 byte 类型的值赋给 char 类型的变量时，需要执行强制转换。

2.14　运算符的优先级

表 2-8 从高到低显示了所有 Java 运算符的优先次序。该表包括本书后面才会讨论的几个运算符。从技术上讲，[]、()和.也可以作为运算符，此时，它们的优先级最高。

表 2-8　从高到低显示的所有 Java 运算符的优先次序(由高到低，第一行运算符的优先级最高)

++(后缀)	--(后缀)					
++(前缀)	--(前缀)	~	!	+(一元)	-(一元)	(类型强制转换)
*	/	%				
+	-					
>>	>>>	<<				
>	>=	<	<=	instanceof		

(续表)

==	!=					
&						
^						
|						
&&						
||						
?:						
=	op=					

练习 2-2(LogicalOpTable.java) 显示逻辑运算符的真值表

在本练习中，将创建一个显示 Java 逻辑运算符的真值表的程序。必须使真值表按列排列。程序会利用本章涉及的几个特性，包括 Java 转义序列和逻辑运算符。它还演示了算术运算符+和逻辑运算符之间优先顺序的不同。

步骤：

(1) 创建一个名为 LogicalOpTable.java 的新文件。

(2) 为使真值表按列排列，可以使用转义序列\t 在每一个输出字符串中嵌入制表符。例如，下面的 println( ) 语句显示了表头：

```
System.out.println("P\tQ\tAND\tOR\tXOR\tNOT");
```

(3) 真值表中后面的每一行将使用制表符将每个运算的结果定位于相应的标题下。

(4) 下面是完整的 LogicalOpTable.java 程序清单：

```
// Try This 2-2: a truth table for the logical operators.
class LogicalOpTable {
  public static void main(String args[]) {

    boolean p, q;

    System.out.println("P\tQ\tAND\tOR\tXOR\tNOT");

    p = true; q = true;
    System.out.print(p + "\t" + q +"\t");
    System.out.print((p&q) + "\t" + (p|q) + "\t");
    System.out.println((p^q) + "\t" + (!p));

    p = true; q = false;
    System.out.print(p + "\t" + q +"\t");
    System.out.print((p&q) + "\t" + (p|q) + "\t");
    System.out.println((p^q) + "\t" + (!p));

    p = false; q = true;
    System.out.print(p + "\t" + q +"\t");
    System.out.print((p&q) + "\t" + (p|q) + "\t");
    System.out.println((p^q) + "\t" + (!p));

    p = false; q = false;
    System.out.print(p + "\t" + q +"\t");
```

```
    System.out.print((p&q) + "\t" + (p|q) + "\t");
    System.out.println((p^q) + "\t" + (!p));
  }
}
```

注意 println( )语句中包围逻辑运算的圆括号。因为 Java 的运算符有先后顺序，所以这些圆括号是必需的。运算符+的优先级高于逻辑运算符。

(5) 编译并运行该程序，会显示如下所示的真值表：

```
P       Q       AND     OR      XOR     NOT
true    true    true    true    false   false
true    false   false   true    true    false
false   true    false   true    true    true
false   false   false   false   false   true
```

(6) 独立尝试修改程序，使其显示 1 和 0，而不是 true 和 false。这可能会比一开始想的费些时间。

2.15　表达式

表达式由运算符、变量和字面值组成。从其他编程经验或代数中，你可能已经知道了表达式的基本形式。然而，这里还需要对表达式的几个方面进行介绍。

2.15.1　表达式中的类型转换

只要类型兼容，在表达式中就可以混合使用一种或多种不同类型的数据。例如，可在表达式中混合使用 short 和 long，因为它们都是数值类型。当在表达式中混合使用不同类型的数据时，它们都被转换为同一类型。这是使用 Java 的类型升级规则(type promotion rule)完成的。

首先，所有 char、byte 和 short 值都升级为 int 类型。其次，如果有一个操作数是 long 类型，整个表达式就全部升级为 long 类型。如果有一个操作数是 float 类型，整个表达式就全部升级为 float 类型。如果有一个操作数是 double 类型，整个表达式就全部升级为 double 类型。

当计算表达式时，类型升级只应用于被操作的值，理解这一点是非常重要的。例如，如果在表达式中，将 byte 类型变量的值升级为 int 类型；那么在表达式以外，该变量仍然是 byte 类型。类型升级只影响表达式的计算。

然而类型升级可能导致某些不可预测的错误。例如，当一个算术运算包括两个 byte 值时，就会顺序发生以下事情：首先，byte 类型的操作数升级为 int 类型；然后进行运算，生成一个 int 类型的结果。因此，包括两个 byte 类型的运算结果将生成一个 int 类型的值，而这可能不是一下就能预见到的。考虑下面的程序：

```
// A promotion surprise!
class PromDemo {
  public static void main(String args[]) {
    byte b;
    int i;

    b = 10;
    i = b * b; // OK, no cast needed

    b = 10;
     b = (byte) (b * b); // cast needed!!

    System.out.println("i and b: " + i + " " + b);
  }
}
```

不必执行强制类型转换，因为结果已经运算为 int 类型

需要执行强制类型转换，把 int 值赋给 byte 变量

有些出人意料的是，因为当计算表达式时 b 被升级为 int 类型，所以把 b * b 赋值给 i 时不需要执行强制转换。然而当试图将 b * b 赋值给 b 时，却需要强制转换回 byte 类型！如果看上去表达式完全没有问题，却出现意外的类型不兼容错误，那就应该想到这一点。

在执行有关 char 的运算时也会有同样的情况发生。例如，在下面的代码段中，因为在表达式中 ch1 和 ch2 升级为 int 类型，所以需要将其强制转换为 char 类型：

```
char ch1 = 'a', ch2 = 'b';

ch1 = (char) (ch1 + ch2);
```

如果没有执行强制转换，ch1 加 ch2 的结果为 int 类型，这是无法赋值给 char 类型的变量的。

强制转换不只在赋值的类型转换时有用。例如，考虑下面的程序，把整除的结果强制转换为 double 类型来获得小数部分：

```
// Using a cast.
class UseCast {
  public static void main(String args[]) {
    int i;

    for(i = 0; i < 5; i++) {
      System.out.println(i + " / 3: " + i / 3);
      System.out.println(i + " / 3 with fractions: "
                         + (double) i / 3);
      System.out.println();
    }
  }
```

程序的输出结果如下所示：

```
0 / 3: 0
0 / 3 with fractions: 0.0

1 / 3: 0
1 / 3 with fractions: 0.3333333333333333

2 / 3: 0
2 / 3 with fractions: 0.6666666666666666

3 / 3: 1
3 / 3 with fractions: 1.0

4 / 3: 1
4 / 3 with fractions: 1.3333333333333333
```

2.15.2 间距和圆括号

Java 中的表达式可以使用制表符和空格来增强可读性。例如，下面两个表达式的作用相同，但是第二个表达式更易读：

```
x=10/y*(127/x);

x = 10 / y * (127/x);
```

与代数中圆括号的作用一样，Java 中的圆括号增加了所包括运算的优先级。使用多余或附加的圆括号不会引发错误，也不会减慢表达式的执行。Java 提倡使用圆括号以使计算的顺序清晰，帮助自己以及其他阅读程序的人

更好地理解程序。例如，下面的两个表达式中哪一个更易读一些呢？

```
x = y/3-34*temp+127;

x = (y/3) - (34*temp) + 127;
```

2.16　自测题

1. Java 为什么要严格指定基本类型的取值范围和行为？
2. Java 的字符类型是什么，与其他大多数程序设计语言的字符类型的不同之处是什么？
3. 因为任何非 0 值都为 true，所以 boolean 值可以取任何想要的值，对吗？
4. 给定输出如下：

```
One
Two
Three
```

使用一个字符串，显示生成以上输出的 println( )语句。

5. 下面的代码段有什么错误？

```
for(i = 0; i < 10; i++) {
  int sum;

  sum = sum + i;
}
System.out.println("Sum is: " + sum);
```

6. 解释自增运算符的前缀形式与后缀形式有什么不同。
7. 说明短路 AND 是如何用于防止除 0 错误的。
8. 在表达式中，byte 和 short 升级为什么类型？
9. 通常什么时候需要执行强制转换？
10. 编写程序，找出 2 到 100 之间的所有素数。
11. 多余圆括号的使用会影响程序的性能吗？
12. 代码块能定义作用域吗？