



思政材料

第3章 表达式和流程控制语句

3.1 表达式

表达式由运算符和操作数组成，对操作数进行运算符指定的操作，并得出一个结果。Java 运算符按功能可分为：算术运算符、关系运算符、逻辑运算符、位运算符、赋值运算符和条件运算符；除此之外，还有几个特殊用途运算符，如数组下标运算符等。操作数可以是变量、常量或方法调用等。

如果表达式中仅含有算术运算符，如“*”，则为算术表达式，它的计算结果是一个算术量（“+”用于字符串连接除外）。如果表达式中含有关系运算符，如“>”，则为关系表达式，它的计算结果是一个逻辑值，即 `true` 或 `false`。如果表达式中含有逻辑运算符，如“&&”，则为逻辑表达式，相应的计算结果为逻辑值。

3.1.1 操作数

1. 常量

常量操作数很简单，只有基本数据类型和 `String` 类型才有相应的常量形式。

例 3-1 常量示例。

常量	含义
23.59	<code>double</code> 型常量
-1247.1f	<code>float</code> 型常量
true	<code>boolean</code> 型常量
"This is a String"	<code>String</code> 型常量

2. 变量

变量是存储数据的基本单元，它可以作为表达式中的操作数。变量在使用之前要先说明，变量说明的基本格式如下：

类型 变量名 1[= 初值 1][, 变量名 2 [= 初值 2]]...;

其中，类型是变量所属的类型，既可以是基本数据类型，如 `int` 和 `float` 等；也可以是类类型。有时也把类类型的变量称为引用。

变量说明的地方有两处，一处是在方法内，另一处是在类定义内。方法内定义的变量称为自动变量，有的人也喜欢称之为局部变量、临时变量或栈变量。不管叫什么，说的都是一个意思。这里所说的方法，包括程序员定义的各个方法。类中定义的变量就是它的成员变量。

说明基本数据类型的变量之后，系统会自动在内存中为其分配相应的存储空间。说明引用后，系统只分配引用空间，程序员要调用 `new` 来创建实例对象，然后才分配相应的存储空间。

3. 变量初始化

Java 程序中不允许将未经初始化的变量用于操作数。对于基本数据变量，在说明它们的同时可以进行初始化，如：

```
int x = 3;
```

创建一个对象后，使用 `new` 运算符分配存储空间时，系统按表 3-1 中的值自动初始化成员变量。

表 3-1 变量初始值

类 型	初始值	类 型	初始值
byte	(byte)0	double	0.0
short	(short)0	char	'\u0000'(null)
int	0	boolean	false
long	0L	所有引用类型	null
float	0.0f		

具有 `null` 值的引用不指向任何对象。如果使用它指向的对象，则导致一个异常。异常是运行时发生的一个错误，有关内容在第 6 章介绍。

自动变量在使用之前必须进行初始化。编译器扫描代码，判定每个变量在首次使用前是否已被显式初始化。如果编译器发现某个变量没有初始化，则会发生编译时错误。

例 3-2 变量初始化示例。

```
int x = (int)(Math.random() * 100);
int y;
int z;

if (x > 50) {
    y = 9;
}
z = y + x; //可能在初始化之前使用，导致编译错误
```

例 3-2 中程序的前 3 行说明了 3 个整型变量 `x`、`y`、`z`。`x` 初始化为表达式的值，`y` 和 `z` 都没有进行初始化。`y` 的赋值包含在 `if` 语句块中，而该块是否执行要依 `x` 的值而定。`x` 是随机数，当它小于或等于 50 时，程序流跳过 `if` 块，不会给 `y` 赋值，而是执行 `if` 块后的赋值语句。此时因 `y` 没有进行初始化，这条语句将导致一个编译错误。

4. 变量作用域

变量的作用域是指可访问该变量的代码范围。类中定义的成员变量的作用域是整个

类。方法中定义的局部变量的作用域是从该变量的说明处开始到包含该说明的语句块结束处，块外是不可使用的。

块内说明的变量将屏蔽其所在类定义的同名变量。但同一块中如果定义两个同名变量则引起冲突。Java 允许屏蔽，但冲突会引起编译错误。请看程序 3-1。



源代码

程序 3-1

```
1  class Customer {
2      /* 说明变量屏蔽及作用域实例
3      */
4      public static void main(String [] args) {
5          Customer customer = new Customer();
6          String name = "John Smith";
7          {
8              //下列说明是非法的, 仍在前一个 name 的作用域内
9              String name = "Tom David";
10             customer.name = name;
11             System.out.println(
12                 "The customer's name: " + customer.name);
13         }
14     }
15     private String name;
16 }
```

程序 3-1 中定义了 Customer 类，其中有成员变量 name，如第 15 行所示。在 main() 方法中定义了局部变量 name，并赋初值“John Smith”。该局部变量屏蔽了同名的类成员变量。第 9 行又说明变量 name，与第 6 行说明的变量冲突，导致编译错误，如图 3-1 所示，错误的含义是指变量 name 在本方法中已定义。

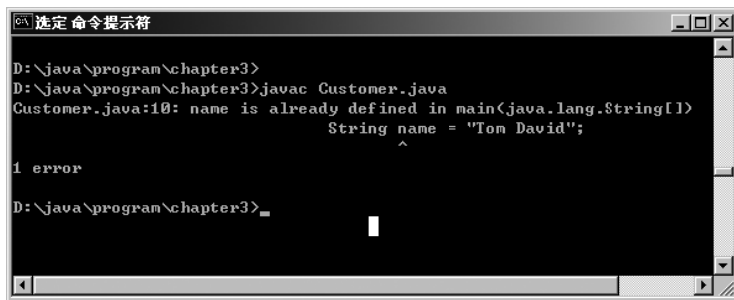


图 3-1 程序 3-1 的运行结果

修改程序 3-1，把第 3 个局部变量的说明改为赋值语句，得到程序 3-2。

程序 3-2

```
1  class Customer {
2      /* 说明变量屏蔽及作用域实例
3      */
4      public static void main(String [] args) {
5          Customer customer = new Customer();
```



源代码

```

6      String name = "John Smith";
7      {
8          name = "Tom David";
9          customer.name = name;
10         System.out.println(
11             "The customer's name: " + customer.name);
12     }
13 }
14 private String name;
15 }

```

该程序编译正确。它的运行结果如图 3-2 所示。

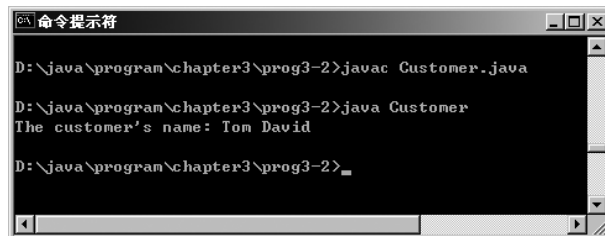


图 3-2 程序 3-2 的运行结果

再看程序 3-3。

程序 3-3

```

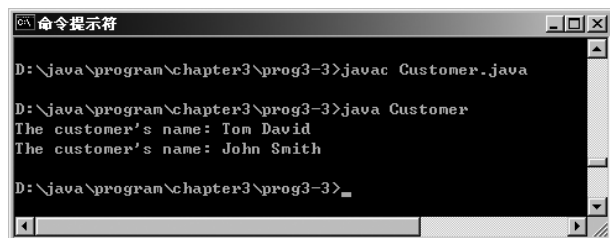
1  class Customer {
2      /* 说明变量屏蔽及作用域实例
3      */
4      public static void main(String [] args) {
5          Customer customer = new Customer();
6          {      String name = "Tom David";
7              customer.name = name;
8              System.out.println("The customer's name: " +
9                  customer.name);
10         }
11         //下面的再说明是正确的,前一个 name 的作用域已结束
12         String name = "John Smith";
13         customer.name = name;
14         System.out.println(
15             "The customer's name: " + customer.name);
16     }
17     private String name;
18 }

```



源代码

程序 3-3 是正确的。虽然 `main()` 方法中两次说明了同名局部变量 `name`，但第 6 行说明的变量只在第 6~10 行的块内有效，在块外该变量消失，第 12 行不在其作用域内。该方法的输出结果如图 3-3 所示。



```
D:\java\program\chapter3\prog3-3>javac Customer.java

D:\java\program\chapter3\prog3-3>java Customer
The customer's name: Tom David
The customer's name: John Smith

D:\java\program\chapter3\prog3-3>_
```

图 3-3 程序 3-3 的运行结果

3.1.2 运算符

Java 的大多数运算符在形式上和功能上都与 C 和 C++ 的运算符非常类似，若熟悉 C 和 C++ 就不会对此感到陌生。

1. 算术运算符

算术运算符包括通常的加 (+)、减 (-)、乘 (*)、除 (/)、取余 (%), 完成整型和浮点型数据的算术运算。许多语言中的取余运算只能用于整型量，Java 对此有所扩展，它允许对浮点类型量进行取余操作。取余运算还可以用于负数，结果的符号与第一个运算数的符号一致。看下面的例子：

3/2	//结果是 1
15.2 % 5	//结果是 0.2
5%-3	//结果是 2
-5%-3	//结果是 -2

此外，算术运算符还有 “++” “--” 两种，分别表示加 1 和减 1 操作。与 C++ 类似，++i 和 i++ 的执行顺序稍有不同，前者在 i 使用之前加 1，后者先使用再加 1。--i 与 i-- 的情况与此类似。

2. 关系运算符

关系运算符用来比较两个值，包括大于 (>)、大于或等于 (>=)、小于 (<)、小于或等于 (<=)、等于 (==)、不等于 (!=) 6 种。关系运算符都是二元运算符，运算的结果是一个逻辑值。二元运算符也称为双目运算符。

Java 允许 “==” 和 “!=” 两种运算用于任何数据类型。例如，可以判定两个实例是否相等。

3. 逻辑运算符

逻辑运算符包括逻辑与 (&&)、逻辑或 (||) 和逻辑非 (!)。前两个是二元运算符，最后一个是一元运算符。一元运算符也称为单目运算符。

Java 对逻辑与和逻辑或提供短路操作功能。进行运算时，先计算运算符左侧表达式的值，如果使用该值能得到整个表达式的值，则跳过运算符右侧表达式的计算，否则计算运算符右侧表达式，并得到整个表达式的值。

例 3-3 短路操作示例。

```
String unset = null;
if ((unset != null) && (unset.length() > 5)) {
    //对 unset 进行某种操作
}
```

空串 `unset` 不能使用，因此不能访问 `unset.length()`，但该 `if()` 语句中的逻辑表达式是合法的，且完全安全。这是因为第一个子表达式 (`unset != null`) 结果为假，马上可知整个表达式的结果为假，所以 `&&` 运算符跳过不必要的 (`unset.length() > 5`) 计算，因为没有计算它，从而避免了空指针异常。

4. 位运算符

位运算符用来对二进制位进行操作，包括按位取反 (`~`)、按位与 (`&`)、按位或 (`|`)、异或 (`^`)、右移 (`>>`)、左移 (`<<`) 及无符号右移 (`>>>`)。位运算符只能对整型和字符型数据进行操作。

Java 提供两种右移运算符。

运算符 “`>>`” 较为常见，它执行算术右移，使用最高位填充移位后左侧的空位。右移的结果为：每移一位，第一个操作数被 2 除一次，移动的次数由第二个操作数确定。

例 3-4 算术右移示例。

```
128 >> 1 得到 64
256 >> 4 得到 16
-256 >> 4 得到 -16
```

逻辑右移（无符号右移）运算符 `>>>` 只对位进行操作，而没有算术含义，它用 0 填充左侧的空位。

例 3-5 逻辑右移示例。

```
(byte)0x80 >> 2 得到 -32
0xa2 >>> 2 得到 40
(byte) 0xa2 >> 2 得到 -24
(byte) 0xa2 >>> 2 得到 1073741800
```

算术右移不改变原数的符号，而逻辑右移不能保证这一点。

移位运算符约简其右侧的操作数，当左侧操作数是 `int` 类型时，右侧以 32 取模；当左侧是 `long` 类型时，右侧以 64 取模。因此，执行

```
int x;
x = x >>> 32;
```

后，`x` 的结果不改变，而不是通常期望的 0。这样可以保证不会将左侧操作数的各位完全移走。

“`>>>`” 运算符只用于整型，它只对 `int` 或 `long` 值起作用。如果用于 `short` 或 `byte` 值，则在进行 “`>>>`” 操作之前，使用符号扩展将其提升为 `int` 型，然后再移位。

5. 其他运算符

Java 中的运算符还包括扩展赋值运算符（+=、-=、*=、/=、%=、&=、|=、^=、>>=、<<=）及（>>>=）、条件运算符（?:）、点运算符（.）、实例运算符（instanceof）、new 运算符、数组下标运算符（[]）等。

扩展赋值运算符是在赋值号（=）前加上其他运算符，是对表达式的一种简写形式。例如，在如下赋值语句中：

```
var = var op expression;
```

var 是变量，op 是算术运算符或位运算符，expression 为表达式。使用扩展赋值运算符可表示为

```
var op= expression;
```

例 3-6 扩展赋值运算符示例。

```
int x = 3;  
x = x * 3;
```

等价于

```
int x = 3;  
x *= 3;
```

条件运算符（?:）是三元运算符，它的一般形式为

逻辑表达式 ? 表达式 1 : 表达式 2;

逻辑表达式得到一个逻辑值，根据该值的真假决定执行什么操作。如果值为真，则计算表达式 1，否则计算表达式 2。注意，表达式 1 和表达式 2 需要返回相同的类型，且不能是 void。

6. 运算符的优先次序

在对一个表达式进行计算时，如果表达式中含有多种运算符，则应按运算符的优先级依次从高向低进行，同级运算符则按结合律进行。括号可以改变运算次序。运算符的优先次序见表 3-2。

表 3-2 运算符的优先次序

优先级	运算符	运 算	结合律
1	[]	数组下标	自左至右
	.	对象成员引用	
	(参数)	参数计算和方法调用	
	++	后缀加	
	--	后缀减	

(续表)

优先级	运算符	运 算	结合律
2	++	前缀加	自右至左
	--	前缀减	
	+	一元加	
	-	一元减	
	~	位运算非	
	!	逻辑非	
3	new	对象实例	自右至左
	(类型)	转换	
4	*	乘法	自左至右
	/	除法	
	%	取余	
5	+	加法	自左至右
	+	字符串连接	
	-	减法	
6	<<	左移	自左至右
	>>	用符号位填充的右移	
	>>>	用 0 填充的右移	
7	<	小于	自左至右
	<=	小于或等于	
	>	大于	
	>=	大于或等于	
	instanceof	类型比较	
8	==	等于	自左至右
	!=	不等于	
9	&	位运算与	自左至右
	&	布尔与	
10	^	位运算异或	自左至右
	^	布尔异或	
11		位或	自左至右
		布尔或	
12	&&	逻辑与	自左至右
13		逻辑或	自左至右
14	?:	条件运算	自右至左
15	=	赋值	自右至左
	+=	加法赋值	
	+=	字符串连接赋值	
	-=	减法赋值	

(续表)

优先级	运算符	运 算	结合律
15	*=	乘法赋值	自右至左
	/=	除法赋值	
	%=	取余赋值	
	<<=	左移赋值	
	>>=	右移（符号位）赋值	
	>>>=	右移（0）赋值	
	&=	位与赋值	
	&=	布尔与赋值	
	^=	位异或赋值	
	^=	布尔异或赋值	
	=	位或赋值	
	=	布尔或赋值	

3.1.3 表达式的提升和转换

Java 语言不支持变量类型间的自动任意转换，有时必须显式地进行变量类型的转换。一般的原则是，变量和表达式可转换为更一般的形式，而不能转换为更受限制的形式。例如，int 型表达式可看作是 long 型的；而 long 型表达式当不使用显式转换时是不能看作 int 型的。通常，如果变量类型至少与表达式类型一样（即位数一样多），就可以认为表达式是赋值相容的。

例 3-7 类型转换示例。

```
long bigval = 6;           //6 是整型量，该语句正确
int smallval = 99L;        //99L 是长整型量，该语句错误
float z = 12.414F;         //12.414F 是浮点量，该语句正确
float z1 = 12.414;         //12.414 是双精度量，该语句错误
```

99L 是长整型量，smallval 是 int 型量，赋值不相容。同样，12.414 是双精度型的，不能赋给单精度变量 z1。

当表达式不是赋值相容时，有时需进行转换以便让编译器认可该赋值。例如，为一个 long 型值“挤”入 int 型变量中，显式转换如下：

```
long bigValue = 99L;
int squashed = (int) (bigValue);
```

转换时，目标类型用括号括起来，放到要修改的表达式的前面。为避免歧义，被转换的整个表达式最好也用括号括起来。

3.1.4 数学函数

表达式中还经常出现另一类元素，这就是数学函数。Java 语言提供了数学函数类

Math，其中包含了常用的数学函数，读者可以按需调用。下面列出几个常用的函数调用情况：

```
Math.sin(0)           //返回 0.0，这是 double 类型的值
Math.cos(0)           //返回 1.0
Math.tan(0.5)         //返回 0.5463024898437905
Math.round(6.6)       //返回 7
Math.round(6.3)       //返回 6
Math.ceil(9.2)        //返回 10.0
Math.ceil(-9.8)       //返回 -9.0
Math.floor(9.2)       //返回 9.0
Math.floor(-9.8)      //返回 -10.0
Math.sqrt(144)        //返回 12.0
Math.pow(5,2)         //返回 25.0
Math.exp(2)           //返回 7.38905609893065
Math.log(7.38905609893065) //返回 2.0
Math.max(560,289)     //返回 560
Math.min(560,289)     //返回 289
Math.random()         //返回 0.0 到 1.0 之间双精度的一个随机数值
```



源代码

3.2 流 控 制

Java 程序中的语句指示计算机完成某些操作，一个语句的操作完成后会把控制转给另一个语句。

语句是 Java 的最小执行单位，语句间以分号 (;) 作为分隔符。语句分为简单语句及复合语句，简单语句就是通常意义下的一条语句，即单语句；而复合语句是一对大括号 “{}” 括起来的语句组，也称为“块”，块中往往含有多个语句，块后没有分号。

下面介绍几种类型的语句，包括表达式语句、块、分支语句和循环语句等。

3.2.1 表达式语句

在 Java 程序中，表达式都可当做一个值，而有的表达式也可当作语句。

语句与表达式有相同的地方也有不同的地方。首先，有的表达式可以当作语句，但并不是所有的语句都是表达式；另外，每个表达式都会得到一个值，即表达式的计算结果。虽然语句也会有一个值，但这个值并不是语句的计算结果，而是运行结果。

下面是一些表达式语句：

```
customer1 = new Customer();
point2 = new Point();
x = 12;
x++;
```

方法调用通常返回一个值，一般用在表达式中。有的方法调用可直接当做语句。例如：

```
System.out.println("Hello World!");
```

3.2.2 块

块是一对大括号“{}”括起来的语句组。例如，下面是两个块：

```
{ }  
{ Point point1 = new Point();  
  int x = point1.x;  
}
```

第一个块是空块，其中不含任何语句。第二个块含两条语句。

方法体是一个块。块还用在流控制的语句（如 `if` 语句、`switch` 语句及循环语句）中。

3.2.3 分支语句

分支语句根据一定的条件，动态决定程序的流程方向，从程序的多个分支中选择一个或几个来执行。分支语句共有两种：`if` 语句和 `switch` 语句。

1. `if` 语句

`if` 语句是单重选择，最多只有两个分支。`if` 语句的基本格式是

```
if (逻辑表达式)  
    语句 1;  
[else  
    语句 2;  
]
```

`if` 语句中的语句 1 或是语句 2 可以是任意语句。其中 `else` 子句是可选的，当然还可以是 `if` 语句，这样的 `if` 语句称为嵌套的 `if` 语句。使用嵌套的 `if` 语句可以实现多重选择，即可以有多个分支。

`if` 关键字之后的逻辑表达式必须得到一个逻辑值，不能像其他语言那样以数值代替。因为 `Java` 不提供数值与逻辑值之间的转换。例如，`C` 语言中的语句形式：

```
int x = 3;  
if (x)  
{...}
```

在 `Java` 程序中应该写作：

```
int x = 3;  
if (x!=0)  
{...}
```

`if` 语句的含义是：当逻辑表达式结果为 `true` 时，执行语句 1，然后继续执行 `if` 后面的语句。当逻辑表达式为 `false` 时，如果有 `else` 子句，则执行语句 2，否则跳过该 `if` 语句，继续执行后面的语句。语句 1 和语句 2 既可以是单语句，也可以是语句块。

下面的示例是 if 语句的常见形式，其中形式 3 就是常见的 if 语句的嵌套。

形式 1:

```
if (逻辑表达式) {  
    //逻辑表达式为 true 时要执行的语句;  
}
```

形式 2:

```
if (逻辑表达式) {  
    //逻辑表达式为 true 时要执行的语句;  
}  
else {  
    //逻辑表达式为 false 时要执行的语句;  
}
```

形式 3:

```
if (逻辑表达式 1) {  
    //逻辑表达式 1 为 true 时要执行的语句;  
}  
else if (逻辑表达式 2) {  
    //逻辑表达式 1 为 false, 且逻辑表达式 2 为 true 时要执行的语句;  
}  
...  
else {  
    //前面的逻辑表达式全为 false 时要执行的语句;  
}
```

例 3-8 if 语句示例。

```
1  int count;  
2  count = getCount(); //程序中定义的一个方法, 返回一个整型值  
3  if (count < 0) {  
4      System.out.println("Error: count value is negative!");  
5  }  
6  else {  
7      System.out.println("There will be " + count +  
8      " people for lunch today.");  
9  }
```

if 语句可以嵌套，嵌套时，由于 else 子句是可选的，故 if 与 else 的个数可能不一致，这就存在配对匹配的问题。else 对应的是哪个 if 呢？在什么条件下执行其后的语句呢？如果不明确 else 与哪个 if 对应，就不能做出正确的判断，程序的运行结果也会有差异。与大多数语言的规定相同，Java 规定 else 子句属于逻辑上离它最近的 if 语句。

例 3-9 嵌套 if 语句示例。

```
1  if (firstVal == 0)  
2      if (secondVal == 1)
```

```

3         firstVal++;
4     else
5         firstVal--;

```

第 4 行的 `else` 子句与第 2 行的 `if` 配对，当 `firstVal` 为 0 且 `secondVal` 不为 1 时，执行 `firstVal`—语句。如果想改变 `else` 的匹配关系，可以使用 “{ }” 改变语句结构。

例 3-10 改变匹配关系。

```

1  if (firstVal == 0){
2      if (secondVal == 1)
3          firstVal++;
4  }
5  else
6      firstVal--;

```

这次，`else` 子句与第 1 行的 `if` 配对，当 `firstVal` 不为 0 时执行 `firstVal`—操作。

2. switch 语句

前面已经看到，使用 `if` 语句可以实现简单的分支判断，并进而执行不同的语句。当需要进行多种条件的判断时，可以使用嵌套的 `if` 语句来实现。当然，这样的语句写起来较烦琐，最主要的是它的条件判定不太直观。实际上，为了方便地实现多重分支，Java 语言还提供了 `switch` 语句。它的含义与嵌套的 `if` 语句类似，只是格式上更加简洁。`switch` 语句的语法格式是

```

switch (表达式) {
    case c1:
        语句组 1;
        break;
    case c2:
        语句组 2;
        break;
    ...
    case ck:
        语句组 k;
        break;
    [default:
        语句组;
        break;]
}

```

这里，表达式的计算结果必须是 `int` 型或字符型，即是 `int` 型赋值相容的。当用 `byte`、`short` 或 `char` 类型时，要进行提升。Java 规定 `switch` 语句不允许使用浮点型或 `long` 型表达式。`c1`，`c2`，`...`，`ck` 是 `int` 型或字符型常量。`default` 子句是可选的，并且，最后一个 `break` 语句完全可以不写。

`switch` 语句的语义是：计算表达式的值，用该值依次和 `c1`，`c2`，`...`，`ck` 相比较。如果该值等于其中之一，例如 `ci`，那么执行 `case ci` 之后的语句组 `i`，直到遇到 `break` 语句跳

到 `switch` 之后的语句。如果没有相匹配的 `ci`，则执行 `default` 之后的语句。也可以将 `default` 看作一个分支，即前面的条件均不满足时要执行的语句。`switch` 语句中各 `ci` 之后的语句既可以是单语句，也可以是语句组。不论执行哪个分支，程序流都会顺序执行下去，直到遇到 `break` 语句为止。

例 3-11 `switch` 语句示例。

```
//colorNum 是整型变量
switch (colorNum) {
    case 0:
        setBackground(Color.red);
        break;
    case 1:
        setBackground(Color.green);
        break;
    default:
        setBackground(Color.black);
        break;
}
```

例 3-11 根据 `colorNum` 的值来设置背景色，其值为 0 时设置为红色，其值为 1 时设置为绿色，否则设置为黑色。这里使用了 Java 预设的一个类 `Color`。

实际上，`switch` 语句和 `if` 语句可以互相代替。例如例 3-11 也可以用 `if` 语句实现。

例 3-12 替换为 `if` 语句。

```
if (colorNum == 0)
    setBackground(Color.red);
else if (colorNum == 1)
    setBackground(Color.green);
else
    setBackground(Color.black);
```

例 3-13 的两段程序实现的逻辑相同，都是根据 `month` 的值返回该月的天数，当然这里只处理平年的情况，没有考虑闰年的特殊处理。

例 3-13 `switch` 语句与 `if` 语句的等价性示例。

使用 `if` 语句：

```
static int daysInMonth(int month) {
    if (month == 2)
        return(28);
    if ((month == 4) || (month == 6) || (month == 9) || (month == 11))
        return(30);
    return(31);
}
```

使用 `switch` 语句：

```
static int daysInMonth(int month) {
    int days;
```

```

        switch(month) {
            case 2: days = 28; break;
            case 4:
            case 6:
            case 9:
            case 11: days = 30; break;
            default: days = 31;
        }
        return(days);
    }
}

```

程序 3-4 是 switch 语句的应用实例。该程序输出第一个命令行参数首字符的分类信息，并进一步输出该字符。

程序 3-4

```

1  class SwitchTest{
2      public static void main(String [] args) {
3          char ch = args[0].charAt(0);
4          switch (ch) {
5              case '0' : case '1' : case '2' : case '3':
6              case '4' : case '5' : case '6' : case '7':
7              case '8' : case '9' :
8                  System.out.println(
9                      "The character is digit " + ch);
10                     break;
11
12                     case 'a' : case 'b' : case 'c' : case 'd':
13                     case 'e' : case 'f' : case 'g' : case 'h':
14                     case 'i' : case 'j' : case 'k' : case 'l':
15                     case 'm' : case 'n' : case 'o' : case 'p':
16                     case 'q' : case 'r' : case 's' : case 't':
17                     case 'u' : case 'v' : case 'w' : case 'x':
18                     case 'y' : case 'z' :
19                         System.out.println(
20                             "The char is lowercase letter " + ch);
21                         break;
22
23                         case 'A' : case 'B' : case 'C' : case 'D':
24                         case 'E' : case 'F' : case 'G' : case 'H':
25                         case 'I' : case 'J' : case 'K' : case 'L':
26                         case 'M' : case 'N' : case 'O' : case 'P':
27                         case 'Q' : case 'R' : case 'S' : case 'T':
28                         case 'U' : case 'V' : case 'W' : case 'X':
29                         case 'Y' : case 'Z' :
30                             System.out.println(
31                                 "The char is uppercase letter " + ch);
32                             break;
33                     default: System.out.println("The character" + ch
34                         + " is neither a digit nor a letter.");

```



源代码

```

35         }
36     }
37 }

```

当主程序执行时，如果第一个命令行参数的首字符分别是数字、小写字母及大写字母，系统就会显示这个首字符。如果输入的是非数字或字母，则提示该字符不是数字或字母。输出如图 3-4 所示。

```

D:\java\program\chapter3>javac SwitchTest.java

D:\java\program\chapter3>java SwitchTest 1
The character is digit 1

D:\java\program\chapter3>java SwitchTest w
The char is lowercase letter w

D:\java\program\chapter3>java SwitchTest A
The char is uppercase letter A

D:\java\program\chapter3>java SwitchTest .
The character. is neither a digit nor a letter.

D:\java\program\chapter3>

```

图 3-4 程序 3-4 的运行结果

如果不写上述方法中的最后一个 break 语句（第 32 行），则程序执行完第 30、31 行后将不停止，一直执行下去。程序的输出如图 3-5 所示。

```

D:\java\program\chapter3>javac SwitchTest1.java

D:\java\program\chapter3>java SwitchTest1 A
The char is uppercase letter A
The characterA is neither a digit nor a letter.

D:\java\program\chapter3>

```

图 3-5 去掉 break 语句的运行结果

3.2.4 循环语句

循环语句控制程序流多次执行一段程序。Java 语言提供 3 种循环语句：for 语句、while 语句和 do 语句。

1. for 循环

for 语句的语法格式是

```

for (初始语句; 逻辑表达式; 迭代语句)
    语句;           //循环体

```

初始语句和迭代语句中可以含有多个语句，各语句间以逗号分隔。for 语句括号内的 3 个部分都是可选的，逻辑表达式为空时，默认规定为恒真。

for 语句的语义是：先执行初始语句，判断逻辑表达式的值。当逻辑表达式为真时，

执行循环体语句，执行迭代语句，然后再去判断逻辑表达式的值。这个过程一直进行下去，直到逻辑表达式的值为假时，循环结束，转到 for 之后的语句。for 语句中定义的循环控制变量只在该块内有效。

例 3-14 for 语句示例。

```
1  for (int i = 0; i < 3; i++) {
2      System.out.println("Are you finished yet?");
3  }
4  System.out.println("Finally!");
```

该段程序共执行 3 次第 2 行的输出语句（i 为 0、1、2 时）。当 i 等于 3 时，逻辑表达式的值为假，退出循环，执行第 4 行语句。程序输出结果为

```
Are you finished yet?
Are you finished yet?
Are you finished yet?
Finally!
```

如果逻辑表达式的值永远为真，则循环会无限制地执行下去，直到系统资源耗尽为止。

例 3-15 无限循环示例。

```
for ( ; ; )
    System.out.println("Always print!");
```

上述语句等价于：

```
for ( ; true ; )
    System.out.println("Always print!");
```

这段循环不会停止。

例 3-16 是初始语句及迭代语句包含多个语句时的情况。

例 3-16 初始语句和迭代语句示例。

```
int sumi = 0, sumj = 0;
for (int i = 0, j = 0; j < 10; i++, j++) {
    sumi += i;
    sumj += j;
}
```

2. while 循环

for 语句中常常用循环控制变量显式控制循环的执行次数。当程序中不能明确地指明循环的执行次数时，可以仅用逻辑表达式决定循环的执行与否。这样的循环可用 while 语句实现。

while 语句的语法格式是

```
while (逻辑表达式)
    语句;                //循环体
```

与 if 语句一样，while 语句中的逻辑表达式亦不能用数值代替。

while 语句的语义是：计算逻辑表达式，当逻辑表达式为真时，重复执行循环体语句，直到逻辑表达式为假时结束。如果第一次检查时逻辑表达式为假，则循环体语句一次也不执行。如果逻辑表达式始终为真，则循环不会终止。

例 3-14 的 for 语句可以改写为例 3-17 中的 while 语句。

例 3-17 while 语句示例。

```
int i = 0;
while (i < 3) {
    System.out.println("Are you finished yet?");
    i++;
}
System.out.println("Finally!");
```

3. do 循环

do 语句与 while 语句很相似。它把 while 语句中的逻辑表达式移到循环体之后。do 语句的语法格式是

```
do
    语句;                //循环体
while (逻辑表达式);
```

do 语句的语义是：首先执行循环体语句，然后判断逻辑表达式的值。当表达式为真时，重复执行循环体语句，直到表达式为假时结束。不论逻辑表达式的值是真是假，do 循环中的循环体都至少执行一次。

例 3-18 do 语句示例。

```
//do 语句
int i = 0;
do {
    System.out.println("Are you finished yet?");
    i++;
} while (i < 3);
System.out.println("Finally!");
```

实际上，for、while 及 do 语句可互相替代。例如：

```
do
    语句 1;
while (逻辑表达式);
```

等价于：

```
语句 1;
while(逻辑表达式)
    语句 1;
```

3.2.5 break 语句与 continue 语句

Java 语言抛弃了有争议的 goto 语句，代之以两条特殊的流控制语句：break 语句和 continue 语句，它们用在分支语句或循环语句中，使得程序员更方便控制程序执行的方向。

1. 标号

标号可以放在 for、while 或 do 语句之前，其语法格式为

标号:语句;

2. break 语句

break 语句可用于 3 类语句中，一类是 switch 语句，一类是 for、while 及 do 等循环语句，还有一类是块语句。在 switch 语句及循环语句中，break 的语义是跳过本块中余下的所有语句，转到块尾，执行其后的语句。

例 3-19 break 语句示例。

```
for (int i = 0; i < 100; i++) {  
    if (i == 5)  
        break;  
    System.out.println("i= " + i);  
}
```

循环控制条件控制循环应该执行 100 次（i 为 0~99）。当 i 等于 5 时，执行 break 语句，它跳过余下的语句，结束循环。实际上，循环体 System.out.println 只执行了 5 次（i 从 0 到 4）。

break 语句的第 3 种使用方法是在块中和标号配合使用，其语法格式为

break 标号;

其语义是跳出标号所标记的语句块，继续执行其后的语句。这种形式的 break 语句多用于嵌套块中，控制从内层块跳到外层块之后。

再看例 3-20 和程序 3-5。

例 3-20 break 语句与标号示例。

```
int x;  
out:   for (i=0; i<10; i++) {  
        x=10;  
        while (x<50) {  
            x++;  
            if (i*x == 400)  
                break out;  
        }  
    }
```

程序 3-5

```
class Break {
    public static void main (String args[]){
        int i, j = 0, k = 0, h;
        label1: for(i = 0; i < 100; i++, j += 2)
        label2: {
            label3:      switch(i%2) {
                           case 1: h=1;
                               break;
                           default:h=0;
                               break;
                           }
                        if(i==50)
                            break label1;
            }
            System.out.println("i=" + i);
        }
    }
}
```



源代码

程序执行后，根据 *i* 的值进行判断。如果 *i* 是奇数，则 *h* 为 1；如果 *i* 是偶数，则 *h* 为 0。`switch` 语句中的 `break` 语句都只是跳过 `switch` 本身，并没有跳出 `for` 循环。当 *i* 增到 50 时，进入 `if` 语句块，执行的结果是跳出 `label1` 标记的语句块，即 `for` 语句块。接下来的语句是打印语句，输出 *i* 的值。运行结果如图 3-6 所示。

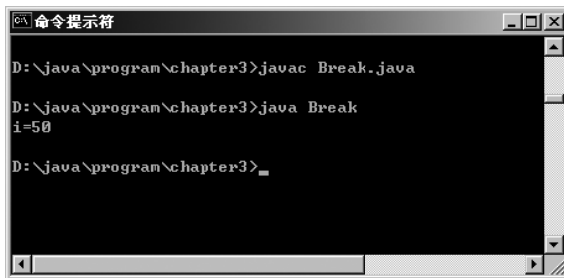


图 3-6 程序 3-5 的运行结果

3. continue 语句

在循环语句中，`continue` 可以立即结束当次循环而执行下一次循环，当然执行前要先判断循环条件是否满足。

`continue` 语句也可以和标号一起使用，其语法格式为

continue 标号;

它立即结束标号标记的那重循环的当次执行，开始下一次循环。这种形式的语句多用于多重循环中。

例 3-21 `continue` 语句示例。

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



源代码