

程序员书库

初学者的入门宝典，程序员的百科全书



CD-ROM

10小时多媒体视频讲解

#### 本书特色

- ※ 起点低，即使没有任何编程经验，也能通过本书掌握Java
- ※ 避免大段理论讲解，而是通过大量实例进行讲解，有很强的实践性
- ※ 对代码进行了详细注释，阅读起来非常容易，没有任何障碍
- ※ 通过现实中的事物类比Java中的概念，使读者可以很容易理解
- ※ 重点讲解Java语言的基础知识和应用，并对一些设计模式也有所介绍
- ※ 全书提供190个实例和2个综合案例，非常实用

# Java

## 从入门到精通

高宏静 等编著



化学工业出版社

## 第 2 章 Java 基本语言

在上一章中，读者已经了解了如何搭建 Java 开发环境及 Java 程序的开发过程。从本章开始讲解 Java 的基本语言。这些基本语言的语法和其他一些编程语言相比有些是类似的，但还有很多不同之处，读者最好花一定的时间来了解这些最基本的语法。

### 2.1 基础语言要素

一个 Java 程序是由很多部分组成，其中任何一个单词都有它存在的意义。这些单词就是构成一个 Java 程序的基本语言要素。本节就来讲解这些基本语言要素，包括标识符、关键字、分隔符和注释等。

#### 2.1.1 标识符

标识符是程序员为自己定义的类、方法或者变量等起的名称，例如第 1 章程序中的 HelloWorld 和 main 都是标识符，其中 HelloWorld 是类名，main 是方法名，除此之外还可以为变量名、类型名、数组名等。

在 Java 语言中规定标识符由大小写字母、数字、下划线（\_）和美元符号（\$）组成，但是不能以数字开头。例如 HelloWorld、Hello\_World、\$HelloWorld 都是合法的标识符。但是如下几种就不是合法的标识符。

- ❑ 555HelloWorld（以数字开头）。
- ❑ ¥HelloWorld（具有非法字符¥）。

在 Java 中标识符是严格区分大小写的，Hello 和 HELLO 是完全不同的标识符。

注意：标识符不能使用 Java 语言中的关键字，关键字的概念将在下一小节中进行讲解。

正确的标识符不一定是一个好的标识符。在一个大型的程序中，经常要定义上百个标识符，如果没有好的标识符命名习惯，就很可能造成混乱。所以标识符的命名要表达含义，例如定义一个学生类，就使用 Student 来进行命名，而不要为了省事定义为 SD。除此之外，还应有一些根据不同标识符定义的习惯。

- ❑ 包名：使用小写字母。
- ❑ 类名和接口名：通常定义为由具有含义的单词组成，所有单词的首字母大写。
- ❑ 方法名：通常也是由具有含义的单词组成，第一个单词首字母小写，其他单词的首字母都大写。
- ❑ 变量名：成员变量和方法相同，局部变量全部使用小写。
- ❑ 常量名：全部使用大写，最好使用下划线分隔单词。

在本书中，由于前面的程序大部分都是非常简单的，所以命名是很简单的。但是读者一定要从开始就养成好的命名习惯，这样才能在后面的团队开发中适应工作要求。

### 2.1.2 关键字

在过去封建社会中，出现过文字狱，例如不能使用皇帝名字中的字，在朱元璋面前不能提“和尚”两个字等。同样在 Java 语言中也存在这样的“文字狱”，这些字就是 Java 中的关键字，程序员是不能使用这些关键字作为标识符的，这些关键字只能由系统来使用。在程序中，关键字具有特殊的意义，Java 平台根据关键字来执行程序操作。

在很多 Java 书中讲解关键字时都是给出一个表格，然后告诉读者这些是关键字，一定要深刻记忆这些关键字。其实没有几个读者会认真地看完这五十个左右的关键字，更别提记忆了。这里作者简单地给这些关键字分一下类，并进行简单的讲解，让大家有一个简单的了解，在后面的讲解中还要对大部分关键字进行详细的讲解。

#### 1. 访问修饰符关键字

在 HelloWorld 程序中出现的第一个单词就是 `public`，它就是一个访问修饰符关键字。修饰符关键字包括如下几种。

- ❑ `public`：所修饰的类、方法和变量是公共的，其他类可以访问该关键字修饰的类、方法或者变量。
- ❑ `protected`：用于修饰方法和变量。这些方法和变量可以被同一个包中的类或者子类进行访问。
- ❑ `private`：同样修饰方法和变量。方法和变量只能由所在类进行访问。

#### 2. 类、方法和变量修饰符关键字

- ❑ `class`：告诉系统后面的单词是一个类名，从而定义一个类。
- ❑ `interface`：告诉系统后面的单词是一个接口名，从而定义一个接口。
- ❑ `implements`：让类实现接口。
- ❑ `extends`：用于继承。
- ❑ `abstract`：抽象修饰符。
- ❑ `static`：静态修饰符。
- ❑ `new`：实例化对象。

还有几种并不常见的类、方法和变量修饰符，例如 `native`、`strictfp`、`synchronized`、`transient` 和 `volatile` 等。

#### 3. 流程控制关键字

流程控制语句包括 `if-else` 语句、`switch-case-default` 语句、`for` 语句、`do-while` 语句、`break` 语句、`continue` 语句和 `return` 语句，这都是流程控制关键字。还有一个关键字应该也包括在流程控制关键字中，那就是 `instanceof` 关键字，用于判断对象是否是类或者接口的实例。

#### 4. 异常处理关键字

异常处理的基本结构是 `try-catch-finally`，这三个单词都是关键字，异常处理中还包括 `throw` 和 `throws` 这两个关键字。`assert` 关键字用于断言操作中，也算是异常处理关键字。

#### 5. 包控制关键字

包控制关键字只有两个，分别是 `import` 和 `package`。`import` 关键字用于将包或者类导入到程序中；`package` 关键字用于定义包，并将类定义到这个包中。

#### 6. 数据类型关键字

Java 语言中有 8 种基本数据类型，每一种基本数据类型都需要一个关键字来定义，除布尔型（`boolean`）、字符型（`char`）、字节型（`byte`）外，还有数值型。数值型又分为 `short`、`int`、`long`、`float` 和 `double`。

#### 7. 特殊类型和方法关键字

`super` 关键字用于引用父类，`this` 关键字用于应用当前类对象。`void` 关键字用于定义一般方法，该方法没有任何返回值。在 `HelloWorld` 程序中的 `main` 方法前就有该关键字。

#### 8. 没有使用的关键字

在关键字家族中有两个另类，那就是 `const` 和 `goto`。在前面已经知道关键字是系统使用的单词，但是对于这两个另类虽然是关键字，但是系统并没有使用他们。这是初学者应特别注意的地方，在一些考试或者公司面试中经常会问到这个问题。

最后说一个显而易见，但是很多人注意不到的问题，那就是所有的关键字都是小写的，如果采用了大写，那就肯定不是关键字。

### 2.1.3 注释

注释在前面介绍使用 `Eclipse` 开发 Java 程序时已经看到了，工具会自动产生一些注释，后面作者又将自动生成的注释去掉，然后加上自定义的注释。从这里可以看到注释对于程序的运行是不起作用的。

注释添加在代码中，是给程序员看的，当系统运行程序，读取注释时会越过不执行。随着技术的发展，现在具有百万行代码的程序已经很常见了，在这样一个大型的代码中，如果没有注释，可想而知对于后面的修改和维护会产生多大的麻烦。在 Java 语言中提供了完善的注释机制，具有三种注释方式，分别是单行注释（`//`）、多行注释（`/* */`）和文档注释（`/** */`）。具有良好的注释习惯是一个优秀程序员不可缺少的职业素质。

在本书中将主要采用单行注释来对开发的程序进行注释。这有可能并不符合某些公司的开发规范，这里主要是为了讲解知识。读者工作中可以根据自己的开发需要，学习相关的开发规范。

## 2.2 基本数据类型

Java 是一门强数据类型语言，Java 程序中定义的所有数据都有一个固定的数据类型。Java 中的数据类型基本可以分为两类：基本数据类型（也称原始数据类型）和复合数据类型。在本节中主要讲解基本数据类型，复合数据类型在后面的章节中将会讲到。学习数据类型的重点是了解每一种数据类型的取值范围。

### 2.2.1 常量和变量

在正式学习 Java 中的基本数据类型前，先来学习一下数据类型的载体常量和变量。从名称上就可以看出常量和变量的不同，常量表示不能改变的数值，而变量表示能够改变的数值。这里先来看一个计算圆面积的程序。

```
public class YuanMianJi {  
    public static void main(String[] args) {  
        final double PI=3.14;    //定义一个表示 PI 的常量  
        int R=5;                //定义一个表示半径的变量  
        double ymj=PI*R*R;      //计算圆的面积  
        System.out.println("圆的面积等于"+ymj);  
    }  
}
```

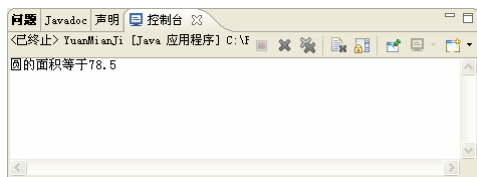


图 2-1 常量和变量实例

运行该程序，运行结果如图 2-1 所示。

在求圆面积时需要两个值，分别是 **PI** 和半径。其中 **PI** 是一个固定的值，可以使用常量来表示，也就是该程序的第 3 行代码，从而知道定义常量需要 **final** 这个关键字。圆的半径是变化的，所以需要使用一个变量来表示。在上面代码中的常量和变量前都有一个关键字 **double** 和 **int**，它们就是这一节将要讲到的数据类型。

### 2.2.2 整数类型

什么是整数这个问题在小学中就学过了，在 Java 中用户存放整数的数据类型称为整数类型。整数类型根据占用的内存空间位数不同可以分为 4 种，分别是 **byte**（字节型）、**short**（短整型）、**int**（整型）和 **long**（长整型），定义数据时默认为 **int** 类型。内存空间位数决定了数据类型的取值范围，表 2-1 中给出了整数类型的位数和取值范围的关系。

表 2-1 整数类型

整数类型	位数	取值范围
字节型	8	$-2^7 \sim 2^7 - 1$
短整型	16	$-2^{15} \sim 2^{15} - 1$
整型	32	$-2^{31} \sim 2^{31} - 1$
长整型	64	$-2^{63} \sim 2^{63} - 1$

注意：在面试或者考试中并不会直接问某一类型的取值范围，而是问具体某一实际例子该使用什么类型，例如表示全球人口该使用什么数据类型。

在 Java 中可以通过 3 种方法来表示整数，分别是十进制、八进制和十六进制。其中十进制都已经非常熟悉了。八进制是使用 0~7 来进行表示的，在 Java 中，使用八进制表示整数必

须在该数的前面放置一个“0”。看下面十进制和八进制数值进行比较的程序。

```
public class JinZhi {
    public static void main(String[] args) {
        int a10=12;           //定义一个十进制数值
        int a8=012;           //定义一个八进制数值
        System.out.println("十进制 12 等于"+a10);
        System.out.println("八进制 12 等于"+a8);
    }
}
```

运行该程序，运行结果如图 2-2 所示。

在程序中定义了两个整型数据类型的变量，值分别是“12”和“012”，如果认为这两个数值相同，那就错了。当一个数值以“0”开头，则表示该数值是一个八进制数值，从运行结果中也可以看到该值为十进制的 10。

除了十进制和八进制外，整数的表示方法还有十六进制。表示十六进制数值除了 0~9 外，还使用 a~f 分别表示从 10 到 15 的数值。表示十六进制时，字母是不区分大小写的，也就是 a 表示 10，A 也表示 10。十六进制同八进制一样，也有一个特殊的表示方式，那就是以“0X”或者“0x”开头。看下面这个使用十六进制表示整数的程序。

```
public class JinZhi16 {
    public static void main(String[] args) {
        int a1=0X12;           //定义一个以数字表示的十六进制整数
        int a2=0xcafe;         //定义一个以字母表示的十六进制整数
        System.out.println("第一个十六进制数值等于"+a1);
        System.out.println("第二个十六进制数值等于"+a2);
    }
}
```

运行该程序，运行结果如图 2-3 所示。

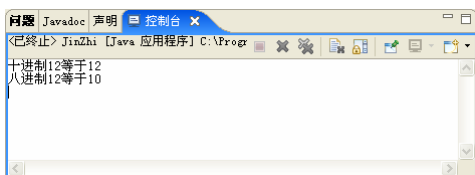


图 2-2 十进制和八进制对比

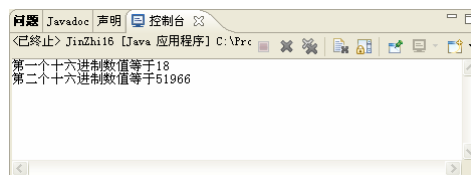


图 2-3 十六进制

该程序中的 0xcafe 是很容易让人迷惑的，在一些面试中经常使用这样的程序来考程序员的细心程度。读者一定要了解它就是一个使用十六进制表示的整数。

在使用 3 种方法表示整数时，都被定义为 int 类型。这里完全可以定义为其其他几种整数类型。但这里需要注意的是如果定义为 long 长整型，则需要在数值后面加上 L 或者 l，例如定义长整型的 12 数值，则应该为 12L。

### 2.2.3 浮点类型

浮点类型和整数类型一样，也是用来表示数值。整数类型是表示整数，而浮点类型表示

的是小数，在 Java 中不称作小数，而称之为浮点数。浮点类型就是表述 Java 中的浮点数。Java 中的浮点类型分为两种，分别是单精度浮点类型和双精度浮点类型。表 2-2 给出了两种浮点类型的取值范围。

表 2-2 浮点类型

类型	位数	取值范围
单精度浮点类型	32	1.4e-45~3.4e+38
双精度浮点类型	64	4.9e-324~1.7e+308

在前面学习计算圆面积时，已经使用到了双精度浮点类型，Java 中默认的浮点类型也是双精度浮点类型。当使用单精度浮点类型时，必须在数值后面跟上 F 或者 f，这和 long 类型是一样的。在双精度浮点类型中，也可以使用 D 或者 d 为后缀，但是它不是必须的，因为双精度浮点类型是默认形式。看下面定义浮点类型的程序。

```
public class FuDian {
    public static void main(String[] args) {
        float f=1.23f;           //定义一个单精度浮点类型
        double d1=1.23;           //定义一个不带后缀的双精度浮点类型
        double d2=1.23D;          //定义一个带后缀的双精度浮点类型
        System.out.println("单精度浮点类型数值等于"+f);
        System.out.println("双精度浮点类型数值等于"+d1);
        System.out.println("双精度浮点类型数值等于"+d2);
    }
}
```

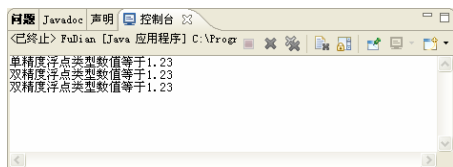


图 2-4 浮点类型

运行该程序，运行结果如图 2-4 所示。

在该程序中，如果将定义单精度浮点类型数值后的 f 去掉，该程序就会发生错误。从定义的是否带 D 后缀的两个双精度浮点类型数值结果可以看出，定义双精度浮点类型时，是否有后缀对结果是没有变化的。

## 2.2.4 字符类型

在开发中，经常要定义一些字符，例如“A”，这时候就要用到字符类型。在 Java 中，字符类型就是用于存储字符的数据类型。在 Java 中，有时会使用 Unicode 码来表示字符。在 Unicode 码中定义了至今人类语言的所有字符集，Unicode 码是通过“\uxxxx”来表示的，x 表示的十六进制数值。Unicode 编码字符是用 16 位无符号整数表示的，即有  $2^{16}$  个可能值，也就是 0~65535。看下面定义字符类型的程序。

```
public class ZiFu {
    public static void main(String[] args) {
        char a='A';
        char b='\u003a';
        System.out.println("第一个字符类型的值等于"+a);
        System.out.println("第二个字符类型的值等于"+b);
    }
}
```

```
}
}
```

运行该程序，运行结果如图 2-5 所示。

从程序可以看到，定义字符类型数值时，可以直接定义一个字符，也可以使用 Unicode 码来进行定义。由于 Unicode 码表示的是全人类字符，所以大部分是看不懂的。还有一些是受操作系统的影响不能显示的，通常会显示为一个问号，所以当显示问号时，可能该 Unicode 表示问号，更有可能是因为该 Unicode 所表示的字符不能正确显示造成的。

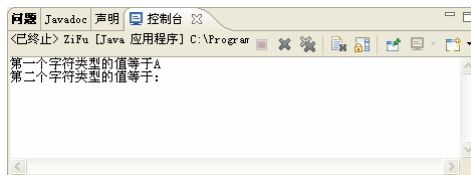


图 2-5 字符类型

在运行结果的显示中，会有一些内容不能显示，例如回车、换行等效果。在 Java 中为了解决这个问题，定义了转义字符。转义字符通常使用“\”开头，在表 2-3 中列出了 Java 中的部分转义字符。

表 2-3 转义字符

转义	说明	转义	说明
'	单引号	\n	换行
"	双引号	\f	换页
\\	斜杠	\t	跳格
\r	回车	\b	退格

在 Java 中，单引号和双引号都表示特定的作用，所以如果想在结果中输入这两个符号，就需要使用转义字符。由于转义字符使用的符号是斜杠，所以如果想输出斜杠时，就需要使用双斜杠。看下面使用转义字符的程序。

```
public class ZhuYiZiFu {
    public static void main(String[ ] args) {
        System.out.println("Hello \n World");
        System.out.println("Hello \\n World");
    }
}
```

运行该程序，运行结果如图 2-6 所示。

从运行结果中可以看到，当把“\n”放到一个字符中输出时，并不是作为字符串输出，而是起到换行的作用。但是如果想直接输出“\n”时，同样需要使用转义字符，先输出一个“\\”，然后后面跟上“n”，这样就输出“\n”这个字符。

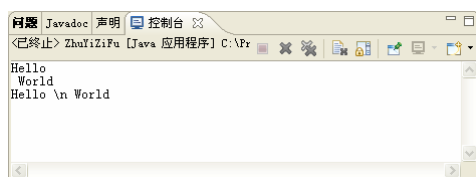


图 2-6 转义字符

## 2.2.5 布尔类型

在 C 语言或者其他一些编程语言中，使用数字来表示 true 和 false。在 Java 中，true 和



false 的待遇明显提高了，为这两个值单独定义了一种数据类型，那就是布尔类型。布尔类型是用于判断逻辑值真假的数据类型。

所有的关系运算的返回类型都是布尔类型。布尔类型也大量应用在控制语句中。运算符和控制语句将在后面的介绍中进行讲解。

## 2.3 数据类型转换

讲完基本数据类型后，在本节中讲解一个重要的知识点，那就是数据类型转换。在前面的学习中，已经知道 Java 是一门强数据类型语言，所以当遇到不同数据类型同时操作时，就需要进行数据类型转换。数据类型转换要满足一个最基础的要求，那就是数据类型要兼容。例如将一个布尔类型转换成整数类型是肯定不能成功的。在 Java 中，有两种数据类型转换方式，分别是自动类型转换和强制类型转换。

### 2.3.1 自动类型转换

在前面学习计算圆面积时已经看到，在程序中定义了半径为 int 类型，而计算的面积为 double 类型，这里就用到了自动类型转换。自动类型转换除了前面讲过的数据类型要兼容外，还需要转换前的数据类型的位数要低于转换后的数据类型。先来看下面的程序。

```
public class ZiDongZhuanHuan {
    public static void main(String[] args) {
        short s=3;           //定义一个 short 类型变量
        int i=s;              //short 自动类型转换为 int
        float f=1.0f;         //定义一个 float 类型变量
        double d1=f;          //float 自动类型转换为 double
        long l=234L;          //定义一个 long 类型变量
        double d2=l;          //long 自动类型转换为 double
        System.out.println("short 自动类型转换为 int 后的值等于"+i);
        System.out.println("float 自动类型转换为 double 后的值等于"+d1);
        System.out.println("long 自动类型转换为 double 后的值等于"+d2);
    }
}
```

运行该程序，运行结果如图 2-7 所示。

从该程序中可以看出位数低的类型数据可以自动转换成位数高的类型数据。例如 short 数据类型的位数为 16，就可以自动转换为位数为 32 的 int 类型。同样 float 数据类型的位数为 32，则就可以自动转换为 64 位的 double 类型。

由于整数类型和浮点类型的数据都是数值，则它们之间也是可以互相转换的，从而有了 long 类型自动转换为 double 类型，但是需要注意的是，转换后的值相同，但是表示上一定要在后面加上小数位，这样才能表示为 double 类型。

需要注意的是，整数类型转换成浮点类型值可能会发生变化，这是由浮点类型的本身定义决定的。计算机内部是没有浮点数的，浮点数是靠整数模拟计算出来的，比如说 0.5 其实

就是 1/2，所以这样的换算过程难免存在误差。看一个整数类型自动转换为浮点类型的程序。

```
public class ZiDongZhuanHuan2 {
    public static void main(String[] args) {
        int l=234234234;           //定义一个 long 类型变量
        float d=l;                  //int 自动类型转换为 double
        System.out.println("int 自动类型转换为 float 后的值等于"+d);
    }
}
```

运行程序，运行结果如图 2-8 所示。

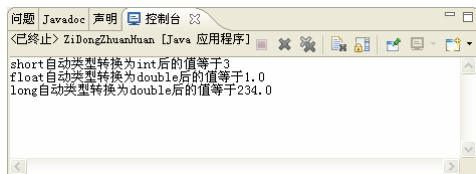


图 2-7 自动类型转换

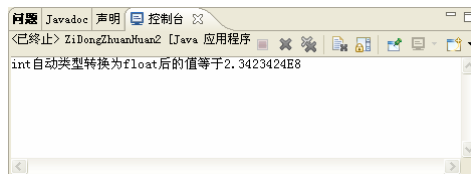


图 2-8 整数转浮点数

从程序和运行结果中可以看到，程序定义的 int 类型为 234234234，而自动转换后的 float 类型为 2.3423424E8。

在前面学习字符类型时，已经知道字符类型占 16 数据位，而且也可以使用 Unicode 码来表示，因此字符类型也可以自动转换为 int 类型的，从而还可以自动转换为更高位的 long 类型，以及浮点类型。看下面字符类型自动类型转换为 int 类型的程序。

```
public class ZiDongLeiZhuan3 {
    public static void main(String[] args) {
        char c1='a';               //定义一个 char 类型
        int i1=c1;                  //char 自动类型转换为 int
        System.out.println("char 自动类型转换为 int 后的值等于"+i1);
        char c2='A';               //定义一个 char 类型
        int i2=c2+1;               //char 类型和 int 类型计算
        System.out.println("char 类型和 int 类型计算后的值等于"+i2);
    }
}
```

运行该程序，运行结果如图 2-9 所示。

从程序可以看到，定义的字符类型数据显示出来为 97 这个数值，这里就是进行了自动转换。而且字符类型还可以作为数值进行计算，学习下一小节的强制类型转换内容，对计算后的数值进行强制类型转换后，会发现输出的结果是 B 这个字符。

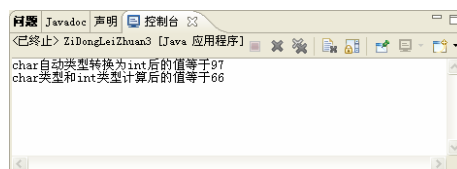


图 2-9 字符转换为 int 类型

### 2.3.2 强制类型转换

在上一小节中已经学习了自动类型转换，自动类型转换是从低位数转换为高位数。有些读者就会有疑问，高位数的数据是否能转换为低位数的数据，这是可以的，这里就要用到强制类型转换。强制数据类型转换的前提条件也是转换的数据类型必须兼容。强制类型转换是

有固定语法格式的，格式如下。

(type) value

其中 type 就是要强制类型转换后的数据类型。看下面进行强制类型转换的程序。

```
public class QiangZhiZhuanHuan1 {
    public static void main(String[] args) {
        int i1=123;           //定义一个 int 类型
        byte b=(byte)i1;      //强制类型转换为 byte
        System.out.println("int 强制类型转换 byte 后值等于"+b);
    }
}
```

运行该程序，运行结果如图 2-10 所示。

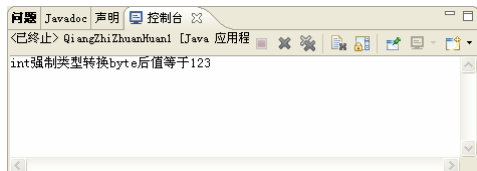


图 2-10 强制类型转换

这是一个简单的强制类型转换的程序，在其中将一个 int 类型的数据强制转换为一个比它位数低的 byte 类型。由于是高位数转换为低位数，也就是大范围转换为小范围，当数值很大的时候，转换就可能造成数据的丢失。例如已经知道 byte 范围的最大值为 127，而其中定义的 int 类型为 128，这时候强制类型就会发生问题，看下面的程序。

```
public class QiangZhiZhuanHuan2 {
    public static void main(String[] args) {
        int i1=128;           //定义一个 int 类型
        byte b=(byte)i1;      //强制类型转换为 byte
        System.out.println("int 强制类型转换 byte 后值等于"+b);
        double d=123.456;     //定义一个 double 类型
        int i2=(int)d;         //强制类型转换为 int
        System.out.println("double 强制类型转换 int 后值等于"+i2);
    }
}
```

运行该程序，运行结果如图 2-11 所示。

在程序中发生了两种数据丢失的现象。首先是 int 类型强制类型转换为 byte 类型，由于是整数类型，所以采用截取的方式进行转换，这是由计算机的二进制表示方法决定的，有兴趣的读者可以自己研究一下，对于 Java 初学者只要知道这样会丢失精度就可以了。第二种情况是浮点类型强制转换为整数类型，这种情况下会丢失小数部分。

在学习自动类型转换时，已经知道字符类型是可以自动转换为数值型的，相反数值型也是可以强制类型转换为字符类型的。继续开发自动类型转换一节最后的程序。

```
public class QiangZhiZhuanHuan3 {
    public static void main(String[] args) {
        char c1='A';          //定义一个 char 类型
        int i=c1+1;            //char 类型和 int 类型计算
        char c2=(char)i;       //进行强制类型转换
        System.out.println("int 强制类型转后为 char 后的值等于"+c2);
    }
}
```

运行该程序，运行结果如图 2-12 所示。

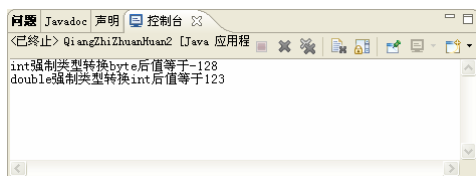


图 2-11 丢失精度

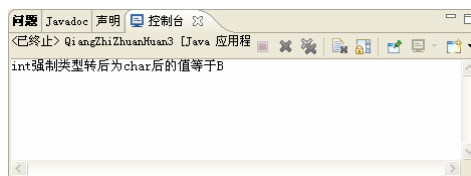


图 2-12 字符和数值转换

在该程序中，将计算后所得到的 `int` 类型强制类型转换为 `char` 类型，从而得到结果 `B` 字符。从这里也可以看出，在 `Unicode` 码中所有的字母都是依次排列的。大写字母和小写字母是不同的，都有自己对应的 `Unicode` 码。

### 2.3.3 隐含强制类型转换

在 `Java` 中有一个特殊的机制，那就是隐含自动类型转换机制。在前面学习整数类型时，已经知道整数的默认类型是 `int`，而程序中经常会出现如下的代码。

```
byte b=123;
```

在这个代码中，`123` 这个数据的类型是 `int` 类型，而定义的 `b` 这个变量是 `byte` 类型。按照前面的理论，这种是需要进行强制类型转换的。这样 `Java` 就提供这种隐含强制类型转换机制，这个工作不再由程序来完成，而是 `Java` 系统自动完成。

有些读者可能会想整数类型如此，浮点类型应该也这样。浮点类型是不存在这种情况的，因为定义 `float` 类型时必须在数值后面跟上 `F` 或者 `f`。

## 2.4 运算符和表达式

学习完前面的基本数据相关内容后，在本节中讲学习运算符。运算符和数学中的加减乘法类似，但是 `Java` 中的运算要比数学中的运算多。表达式可以简单地认为是数据和运算符的结合。

### 2.4.1 算术运算符

算术运算符就是用于计算的运算符，包括加（+）、减（-）、乘（\*）、除（/）等数学中最基本的运算，还包括数学中没有的求余运算（%）。算术运算符的使用是非常简单的，看下面使用算术运算符的程序。

```
public class SuanShu {
    public static void main(String[] args) {
        int i1=7;           //定义两个变量
        int i2=3;
        int jia=i1+i2;       //进行加运算
    }
}
```

```

int jian=i1-i2;      //进行减运算
int cheng=i1*i2;     //进行乘运算
int chu=i1/i2;       //进行除运算
int yu=i1%i2;        //进行求余运算
System.out.println("进行加运算的结果是"+jia);
System.out.println("进行减运算的结果是"+jian);
System.out.println("进行乘运算的结果是"+cheng);
System.out.println("进行除运算的结果是"+chu);
System.out.println("进行求余运算的结果是"+yu);
}
}

```

运行该程序，运行结果如图 2-13 所示。

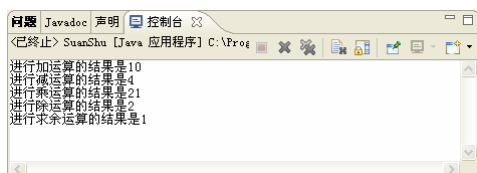


图 2-13 算术运算符

该程序是一个非常简单的使用算术运算符的程序。除法运算中除数是不能为零的。同样在 Java 中也是这样的，而且进行求余运算也是要首先进行除法运算，所以求余运算符的第二个操作数也不能为零。如果为零，则程序就发生错误，看下面定义除数为零的程序。

```

public class SuanShu2 {
    public static void main(String[] args) {
        int i1=7;      //定义两个变量
        int i2=0;       //定义一个值为零的变量
        int chu=i1/i2;   //进行除运算
        int yu=i1%i2;    //进行求余运算
        System.out.println("进行除运算的结果是"+chu);
        System.out.println("进行求余运算的结果是"+yu);
    }
}

```

运行该程序，运行结果如图 2-14 所示。

从该程序的运行结果可以看出，当除数为零时，就会发生 `java.lang.ArithmeticException` 异常，异常的概念在后面会用一章单独讲解。在讲解异常时，也会经常使用到除数为零这个异常程序。

在算术运算符中需要特别说一下加（+）和减（-），它们不单可以用于基本运算，而且也可以作为正数和负数的前缀，这和数学中一样。并且加（+）不但可以用于数字相加，而且被重载为可以用于字符串之间的相加，看下面的程序。

```

public class SuanShu3 {
    public static void main(String[] args) {
        String s1="Hello "; //定义两个字符串
        String s2=" World";
        String s3=s1+s2;     //使用加运算
        System.out.println(s3);
    }
}

```

运行该程序，运行结果如图 2-15 所示。

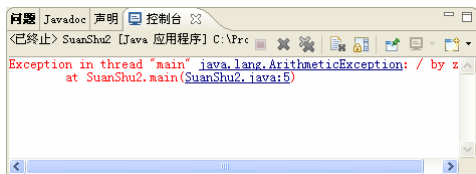


图 2-14 除数为零

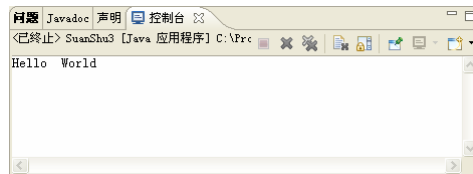


图 2-15 加运算

从运行结果可以看出，使用加运算可以将两个字符串连到一起。这里只要了解这些就可以了，字符串的定义以及其他操作会在后面篇幅中进行讲解。

## 2.4.2 自增自减运算符

自增自减运算符可以算是一种特殊的算术运算符。在算术运算符中需要两个操作数来进行运算，而自增自减运算符是一个操作数，自增运算符表示该操作数递增加 1，自减运算符表示该操作数递增减 1。看下面这个简单的使用自增自减运算符的程序。

```
public class ZiZENGJian1 {
    public static void main(String[] args) {
        int a=3;           //定义一个变量
        int b=++a;          //进行自增运算
        int c=3;           //定义一个变量
        int d=--c;          //进行自减运算
        System.out.println("进行自增运算后的值等于"+b);
        System.out.println("进行自减运算后的值等于"+d);
    }
}
```

运行该程序，运行结果如图 2-16 所示。

从程序和运行结果中可以看出，使用自增运算符后，结果的数值增加 1；使用自减运算符后，结果的数值减小 1。

在前面学习类型转换时，已经知道当两个不同类型的数据进行运算时，低位的数据会自动提升为高位的数据。例如一个 byte 类型的数据和一个 int 类型的数据相加，最后的结果肯定是一个 int 类型。

但是这一点在自增自减运算中是有所不同的，先来看下面的程序。

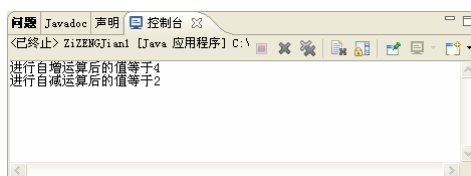


图 2-16 自增自减运算符

```
public class ZiZENGJian2 {
    public static void main(String[] args) {
        byte b1=5;           //定义一个 byte 类型的变量
        byte b2=(byte)(b1+1); //进行强制类型转换
        System.out.println("使用加运算符的结果是"+b2);
        byte b3=5;           //定义一个 byte 类型的变量
        byte b4=++b3;         //进行自增运算，不需要类型转换
        System.out.println("使用自增运算符的结果是"+b4);
    }
}
```

}

运行该程序，运行结果如图 2-17 所示。

在该程序中，当对 `byte` 类型执行加 1 运算时，由于 Java 默认整数类型为 `int`，所以 1 为 `int` 类型，加运算后的结果也为 `int` 类型，从而需要进行强制类型转换。而在使用自增运算时，并不需要强制类型转换。使用自增自减运算符时，并不进行类型的提升，操作前数值是什么类型，操作后的数值仍然是什么类型。

在上面的讲解中，所有的自增自减运算符都放在操作数的前面，自增自减运算符也是可以放在操作数的后面的。这两种方法都可以对操作数进行自增自减操作，只是执行的顺序不同。

□ 前缀方式：先进行自增或者自减运算，再进行表达式运算。

□ 后缀方式：先进行表达式运算，后进行自增或者自减运算。

通过下面的程序来演示这两种方式的不同。

```
public class ZiZENGJian3 {
    public static void main(String[] args) {
        int a=5;           //定义两个值相同的变量
        int b=5;
        int x=2*++a;        //自增运算符前缀
        int y=2*b++;        //自增运算符后缀
        System.out.println("自增运算符前缀运算后 a="+a+"表达式 x="+x);
        System.out.println("自增运算符后缀运算后 b="+b+"表达式 y="+y);
    }
}
```

运行该程序，运行结果如图 2-18 所示。

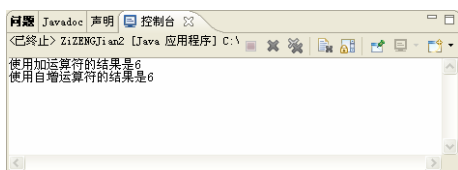


图 2-17 自增运算符的类型转换

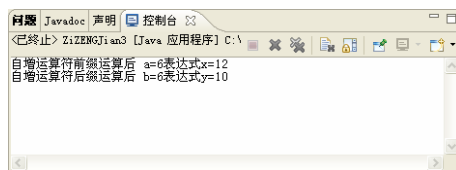


图 2-18 前缀和后缀的不同

从运行结果中首先可以看到，自增运算符不管是前缀还是后缀，最后的结果都增加 1，但是表达式结果完全不同。在计算 `x` 值时，首先执行前缀操作，`a` 的值变为 6，再执行乘操作，从而得到结果为 12；而计算 `y` 值时，先进行乘运算操作，得到结果 10，然后复制给 `y`，从而得到结果为 10。然后才进行自增操作，从而得到 `b` 的值为 6。

注意：从操作中可以看到，自增自减运算符是比较复杂的，而且有很多需要注意的问题。所以在开发中，不是非常必要的时候，不要使用自增自减运算符。

### 2.4.3 关系运算符

关系运算符用于计算两个操作数之间的关系，其结果是布尔类型。关系运算符包括等于（`==`）、不等于（`!=`）、大于（`>`）、大于等于（`>=`）、小于（`<`）和小于等于（`<=`）。首先来讲解等于运算符和不等于运算符，它们可以用于所有基本数据类型和引用类型。由于目前只学过基本数据类型，所以以基本数据类型为例，看下面的程序。



```

public class GuanXi1 {
    public static void main(String[] args) {
        int i=5;                //定义一个 int 类型变量
        double d=5.0;           //定义一个 double 类型变量
        boolean b1=(i==d);      //运用关系运算符的结果
        System.out.println("b1 的结果为: "+b1);
        char c='a';             //定义一个 char 类型变量
        long l=97L;             //定义一个 long 类型变量
        boolean b2=(c==l);      //运用关系运算符的结果
        System.out.println("b2 的结果为: "+b2);
        boolean b1=true;        //定义一个 boolean 类型变量
        boolean b12=false;      //定义一个 boolean 类型变量
        boolean b3=(b1==b12);   //运用关系运算符的结果
        System.out.println("b3 的结果为: "+b3);
    }
}

```

运行该程序，运行结果如图 2-19 所示。

从程序和运行结果中可以看出，int 类型和 double 类型之间、char 类型和 long 类型之间、两个 boolean 类型之间都可以使用关系运算符进行比较。进行关系运算符操作时，自动进行了类型转换，当两个类型兼容时，就可以进行比较。因此 boolean 类型和其他类型是不能使用关系运算符操作的，只能进行两个 boolean 类型间的比较。

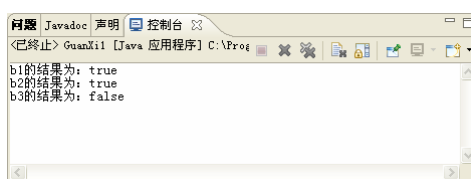


图 2-19 等于关系运算符

除了等于和不等于关系运算符外，其他 4 种关系运算符都是同理的。惟一不同就是，boolean 类型之间是不能进行大小比较的，只能进行是否相等比较。

#### 2.4.4 位运算符

在计算机中，所有的整数都是通过二进制进行保存的，即由一串 0 或者 1 数字组成，每一个数字占一个比特位。位运算符就是对数据的比特位进行操作，只能用于整数类型。位运算符有如下 4 种。

- ❑ 与 (&)：如果对位都是 1，则结果为 1，否则为 0。
- ❑ 或 (|)：如果对位都是 0，则结果为 0，否则为 1。
- ❑ 异或 (^)：如果对位值相同，则结果为 0，否则为 1。
- ❑ 非 (~)：将操作数的每一位按位取反。

下面通过一个程序来演示位运算符的使用。

```

public class Wei {
    public static void main(String[] args) {
        int a=6;                //二进制后四位为 0110
        int b=3;                //二进制后四位为 0011
        int i=a&b;              //执行与位运算操作
        System.out.println("执行与位运算符后的结果等于"+i);
    }
}

```





图 2-20 与位运算符

运行该程序，运行结果如图 2-20 所示。

该程序的运行顺序是，首先将 a 和 b 这两个变量转换为二进制表示，则它们的后四位分别是 0110 和 0011。然后进行与位运算符操作，则计算的结果为 0010。最后将二进制转换为十进制表示，则结果为 2，从而得到图 2-20 中的结果。这里只以与位运算符进行演示，其他的位运算符由读者自己来进行程序开发。

### 2.4.5 移位运算符

移位运算符和位运算符一样都是对二进制数的比特位进行操作的运算符，因此移位运算符也是只对整数进行操作。移位运算符是通过移动比特位的数值来改变数值大小的，最后得到一个新数值。移位运算符包括左移运算符（<<）、右移运算符（>>）和无符号右移（>>>）。

#### 1. 左移运算符

左移运算符用于将第一个操作数的比特位向左移动第二个操作数指定的位数，右边空缺的位用 0 来补充。看下面使用左移运算符的程序。

```
public class YiWei1 {
    public static void main(String[] args) {
        int i=6<<1;           //将数值 6 左移 1 位
        System.out.println("6 左移 1 位的值等于"+i);
    }
}
```

运行该程序，运行结果如图 2-21 所示。

这是一个简单的使用左移运算符的程序，下面通过步骤来进行讲解。首先将数值 6 转换为二进制表示：

```
0000 0000 0000 0000 0000 0000 0000 0110
```

然后执行移位操作，向左移 1 位，则二进制表示为：

```
0000 0000 0000 0000 0000 0000 0000 1100
```

最后将该二进制转换为十进制，则数值为 12，也就是运行结果。从运行结果中也可以看出左移运算相当于执行乘 2 运算。

#### 2. 右移运算符

右移运算符用于将第一个操作数的比特位向右移动第二个操作数指定的位数。在二进制中，首位是用来表示正负的，0 表示正，1 表示负。如果右移运算符的第一个操作数是正数，则填充 0；如果为负数，则填充 1，从而保存正负不变。看下面使用右移运算符的程序。

```
public class YiWei2 {
```

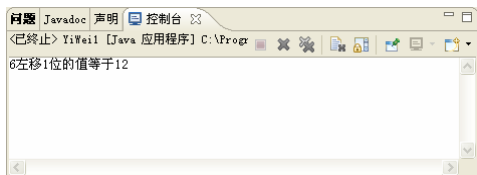


图 2-21 左移运算符

```
public static void main(String[] args) {
    int i=7>>1;           //将数值 7 右移 1 位
    System.out.println("7 右移 1 位的值等于"+i);
}
```

运行该程序，运行结果如图 2-22 所示。

同样一步步来分析该程序的运行经过。首先将数值 7 转换为二进制表示：

```
0000 0000 0000 0000 0000 0000 0000 0111
```

然后执行移位操作，向右移 1 位，因为这是一个正数，所以前面使用 0 填充，二进制表示为：

```
0000 0000 0000 0000 0000 0000 0000 0011
```

将该二进制转换为十进制，则数值为 3，也就是运行结果。从运行方式上可以看出，当第 1 操作数 X 为奇数时，相当于  $(X-1)/2$  操作；当第一操作数 X 为偶数时，相当于  $X/2$  操作。

### 3. 无符号右移运算符

无符号右移运算符和右移运算符的规则是一样的，只是填充时，不管原数是正或是负，都使用 0 来填充。对于正数而言，使用无符号右移运算符是没有意义的，因为都使用 0 来填充。看下面一个对负数使用无符号右移运算符的程序。

```
public class YiWei3 {
    public static void main(String[] args) {
        int i=-8>>>1;           //将数值-8 无符号右移 1 位
        System.out.println("-8 无符号右移 1 位的值等于"+i);
    }
}
```

运行该程序，运行结果如图 2-23 所示。



图 2-22 右移运算符



图 2-23 无符号右移运算符

从运行结果中可以看出，对一个负数无符号右移得到一个很大的正数，下面同样进行分步讲解。首先将 -8 转换为二进制表示：

```
1111 1111 1111 1111 1111 1111 1111 1000
```

然后执行移位操作，向右移 1 位，然后左侧使用 0 填充，二进制表示为：

```
0111 1111 1111 1111 1111 1111 1111 1100
```

将该二进制转换为十进制就是结果中的 2147483644。

## 2.4.6 逻辑运算符

逻辑运算符用于对产生布尔类型数值的表达式进行计算，结果为一个布尔类型。逻辑运算符和位运算符很相似，它也是包括与、或和非，只是各自操作数的类型不同。逻辑运算符可以分为两大类，分别是短路和非短路。

### 1. 非短路逻辑运算符

非短路逻辑运算符包括与（&）、或（|）和非（!）。与逻辑运算符表示当运算符两边的操作数都为 true 时，结果为 true，否则都为 false。或逻辑运算符表示当运算符两边的操作数都为 false 时，结果为 false，否则都为 true。非逻辑运算符表示对操作数的结果取反，当操作数为 true 时，则结果为 false；当操作数为 false 时，则结果为 true。看下面使用非短路逻辑运算符的程序。

```
public class LuoJi {
    public static void main(String[] args) {
        int a=5;                //定义两个变量
        int b=3;
        boolean b1=(a>4)&(b<4);    //使用与逻辑运算符
        boolean b2=(a<4)|(b>4);    //使用或逻辑运算符
        boolean b3=! (a>4);        //使用非逻辑运算符
        System.out.println("使用与逻辑运算符的结果为"+b1);
        System.out.println("使用或逻辑运算符的结果为"+b2);
        System.out.println("使用非逻辑运算符的结果为"+b3);
    }
}
```

运行该程序，运行结果如图 2-24 所示。

该程序非常简单，读者自己分析一下就很容易理解。这里把重点放在短路逻辑运算符的讲解上。

### 2. 短路逻辑运算符

当使用与逻辑运算符时，当两个操作数都为 true 时，结果才为 true。要判断两个操作数，但是当得到第一个操作为 false 时，其结果就必定是 false，这时候再判断第二个操作时就没有任何意义。看下面使用短路逻辑运算符的程序。

```
public class LuoJi2 {
    public static void main(String[] args) {
        int a=5;                //定义一个 int 变量
        boolean b=(a<4)&&(a++<10);    //使用逻辑运算符
        System.out.println("使用短路逻辑运算符的结果为"+b);
        System.out.println("a 的结果为"+a);
    }
}
```

运行该程序，运行结果如图 2-25 所示。

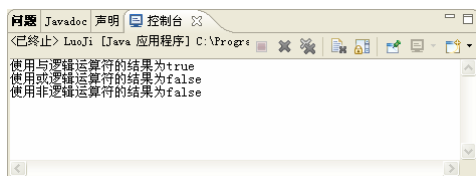


图 2-24 非短路逻辑运算符

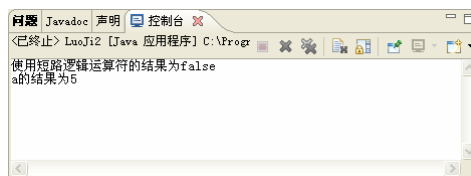


图 2-25 短路逻辑运算符

在该程序中，使用到了短路逻辑运算符（&&）。首先判断  $a < 4$  的结果，则该结果为 `false`，则 `b` 的值肯定为 `false`。这时候就不再执行短路逻辑运算符后面的表达式，也就是不再执行 `a` 的自增操作，从而 `a` 的结果没有变，仍然是 5。

### 2.4.7 三元运算符

Java 中有一个特殊的三元运算符，它支持条件表达式，当需要进行条件判断时可用它来替代 `if-else` 语句。它的语法稍微有点麻烦，但是它能非常高效地完成功能，让代码看起来简洁，优雅。其一般格式如下。

```
expression? statement1 : statement2
```

其中 `expression` 是一个可以计算出 `boolean` 值的表达式。如果 `expression` 的值为真，则执行 `statement1` 的语句，否则执行 `statement2` 的语句。下面是一个使用的示例。

```
ration= denom==0?0:num/denom;
```

上面的语句的意思是，如果 `denom` 的值为 0，则对 `ration` 赋值为 0，否则令 `ration` 的值为 `num/denom`。如果使用条件表达式的话，表达起来要麻烦一点。这样比较简单，下面是一个完整的示例，程序对一个数取绝对值。

```
public class Sanyuan
{
    public static void main(String args[] )
    {
        //声明一系列的 int 类型变量
        int i,k;
        i=5;
        //使用三元运算符对 k 进行赋值操作
        k=(i>=0?i:-i);
        System.out.println("the absolute of "+i+" is "+k);
        i=-5;
        k=(i>=0?i:-i);
        System.out.println("the absolute of "+i+" is "+k);
    }
}
```

程序的运行结果如下。

```
the absolute of 5 is 5
the absolute of -5 is 5
```

该程序的作用是求数的绝对值。当 `i` 变量的值为大于等于 0 时，得到的是 `i` 变量本身。如果 `i` 变量的值小于 0，则得到的是对 `i` 取负的值。

2.4.8 运算符优先级

在一个表达式中可能含有多个运算符，它们之间是有优先级关系的，这样才能有效地把它们组织到一起进行复杂的运算，表 2-3 是 Java 中运算符的优先级。

表 2-3 运算符优先级

最高优先级			
()	[]	.	
++	--	~	!
*	/	%	
+	-		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
&&			
?:			
=	+=	-=	*=
最低优先级			

在所有的运算符中，圆括号的优先级最高，所以适当地使用圆括号可以改变表达式的含义。语句如下。

```
i=a+b*c;  
i=(a+b)*c;
```

上面两个语句的表达意思是不同的。还有就是可以适当地使用括号，使得表达式读起来清晰易懂。

2.5 小结

在这一章中，主要学习了基本数据类型和各种表达式的使用。在实际运用中这些表达式是十分有用的。通过表达式将各种数据合理有效地结合在一起，是使程序高效、简洁的秘诀所在。希望读者认真阅读，为以后的学习打下良好的基础。