第3章

Java 语言基础

本章将对 Java 语言的基本语法成分进行介绍，包括标识符、数据类型、表达式、语句、程序流控制与数组等。

3.1 标识符与数据类型

3.1.1 Java 基本语法

1. 语句与语句块

Java 中是以“；”为语句的分隔符。一个语句可写在连续的若干行内。例如下面的两个语句是等价的：

```
x = a + b + c + d + e;
x = a + b+
      d + e;
```

一对大括号“{”和“}”包含的一系列语句称为语句块。语句块可以嵌套，即语句块中可以嵌套子语句块。

Java 源程序中允许在变量、标识符、表达式、语句等代码元素之间出现任意数量的空白。空格、Tab 键和换行符都是空白。在程序中适当使用空白可以使程序层次清晰，增加程序的可读性。

2. 注释

程序中适当加入注释，会增加程序的可读性。程序中允许加空白的地方就可以写注释，编译器将忽略所有注释。

Java 中有如下 3 种注释。

(1) //：注释一行。表示从//开始到行尾都是注释文字。

(2) /*　　*/：注释一行或多行。表示/*和*/之间的所有内容都是注释。

(3) /**　　*/：文档注释。表示在/**和*/之间的文本，将自动包含在用 javadoc 命令生成的 HTML 格式的文档中。javadoc 是 JDK 中 API 文档生成器。该工具解析一组 Java 源文件中的声明与文档注释，生成一组 HTML 页面描述这些源程序中定义的类、内部类、接口、构造方法、方法与属性。JDK 的 API 文档就是用 javadoc 工具生成的。

3.1.2 标识符

1. 标识符的定义规则

在 Java 语言中,采用标识符对变量、类和方法进行命名。对标识符的定义需要遵守以下规则。

- 标识符是以字母,"_"(下画线),或"$"开始的一个字符序列。
- 数字不能作为标识符的第一个字符。
- 标识符不能是 Java 语言的关键字,但可用关键字作为标识符的一部分。
- 标识符大小写敏感,且长度没有限定。

例如,username、user_name、_sys_var、$change、thisOne 均是合法的标识符。

Java 不采用通常计算机系统采用的 ASCII 代码集,而是采用 Unicode 这样一个国际标准字符集。在这种字符集中,每个字母用 16 位表示,整个字符集中共包含 65536 个字符。其中,ASCII 代码集中的字符如英文字母 A～Z、a～z 和数字 0～9 在 Unicode 字符集中还是用十六进制的 0x0041～0x005a、0x0061～0x007a 和 0x30～0x39 来表示,以表示对 ASCII 码的兼容。另外,Unicode 字符集涵盖了像汉字、日文、朝鲜文、德文、希腊文等多国语言中的符号。这样,Java 中的"字母"和"数字"这两个术语涵盖的范围要广得多。其中,字母被定义成 A～Z、a～z 或国际语言中相当于一个字母的任何 Unicode 字符。

一般情况下,标识符中使用的字母包括下面几种。

(1) A～Z。

(2) a～z。

(3) Unicode 字符集中序号大于等于 0x00c0 的所有国际语言中相当于一个字母的任何 Unicode 字符。

为了准确起见,可以使用 Character 类中的 isJavaIdentifierStart(char ch)方法和 isJavaIdentifierPart(char ch)方法测试参数变量 ch 中的 Unicode 字符是否可以作为标识符的开始字符或后续字符。

2. 标识符风格约定

(1) 对于变量名和方法名,_和 $ 不作为标识符的第一个字符,因为这两个字符对于内部类具有特殊含义。

(2) 类名、接口名、变量名和方法名采用大小写混合的形式,即每个单词的首字母大写,其余小写。但变量名和方法名第一个单词的首字母小写,例如 anyVariableWord。而类名和接口名第一个单词的首字母大写,例如 HelloWorld。

(3) 常量名完全大写,并且用下画线_作为标识符中各个单词的分隔符,例如 MAXIMUM_SIZE。

(4) 方法名应该使用动词,类名与接口名应该使用名词。例如:

```
class Account                    //类名
interface AccountBook            //接口名
balanceAccount()                 //方法名
```

(5) 变量名应该能够表示一定的含义,因此应尽量不使用单个字符作为变量名。但临时性变量如循环控制变量可以采用 i、j、k 等。

3.1.3　关键字

在表 3-1 中列出了 Java 的关键字，这些单词是 Java 语言的保留字。Java 编译器在词法扫描时，需要区分关键字和一般的标识符，因此，用户自定义的标识符不能与这些关键字重名，否则会产生编译错误。另外，true、false 和 null 虽然不是关键字，但也被 Java 保留，同样不能用来定义标识符。

表 3-1　Java 语言的关键字

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

3.1.4　基本数据类型

Java 语言定义了 4 类共 8 种基本类型。

- 逻辑型：boolean。
- 文本型：char。
- 整型：byte、short、int 和 long。
- 浮点型：double 和 float。

1. 逻辑型——boolean

boolean 类型数据有两种取值 true 和 false，在机器中只占 1 位。boolean 型变量的默认初始值为 false。例如：

```
boolean truth = true;                    //定义 truth 为 boolean 类型,且初始值为 true
```

注意：与其他高级语言不同，Java 中的布尔值和数字之间不能来回转换，即 false 和 true 不对应于任何零或非零的整数值。

2. 文字型——char 和 String

char 是文字型的基本数据类型，而 String 是类不是基本类型，但很常用，所以在此一并介绍。

1）char

char 是一个 16 位的 Unicode（国际码）字符，用单引号引上。例如：

```
char mychar = 'Q';                       //mychar 变量的初值被置为 Q 字符对应的 16 位 Unicode 值
```

字符型变量的默认初始值是\u0000。

Unicode 字符集可以支持各类文字的字符,总数达 34168 个字符。通过将国际标准的 Unicode 字符集作为字符变量的取值范围,使 Java 能极为方便地处理各种语言,例如可以将汉字作为字符型变量的值,这为程序的国际化提供了方便。

一些控制字符不能直接显示,Java 与 C/C++一样,利用转义序列来表示这些字符。还有一种直接以八进制或十六进制代表字符值的表示方法,在反斜杠后跟 3 位八进制数字或在反斜杠后跟 u,后面再跟 4 位十六进制数字都可代表一个字符常量,如"\141"和"\u0061"都代表字符常量"a"。表 3-2 是 Java 中的转义字符。

表 3-2 Java 中的转义字符序列

转义字符	描 述	转义字符	描 述
\ddd	1～3 位八进制数所表示的字符	\r	回车
\uxxxx	1～4 位十六进制数所表示的字符	\n	换行
\'	单引号字符	\f	走纸换页
\"	双引号字符	\b	退格
\\	反斜杠字符	\t	水平制表(Tab 键)

一般情况下,char 类型的十六进制 Unicode 编码值可自动转换成等值的 int 类型值,并可与 int 类型数值进行运算。而 int 类型到 char 类型需要通过强制类型转换,如例 3-1 所示。

例 3-1 char 类型的值到 int 类型的转换。

```
public class CharToInt{
    public static void main(String args[]){
        int intResult,intVar = 10;
        char charVar = '语';
        intResult = intVar + charVar;
        System.out.println("The char is :  " + charVar);
        System.out.println("The char's Unicode is :  \\u" + Integer.toHexString(charVar));
        System.out.println ("The int value corresponding to the char is :  "
                            + new Integer(charVar).toString());
        System.out.println("Int " + intVar + " adds the char, the result is :  " + intResult);
    }
}
```

例 3-1 中,字符变量 charVar 的值是"语"字。程序中输出了该字符的 Unicode 值以及 Unicode 对应的十进制数值,并打印输出了 charVar 与一个 int 型变量做加法运算后的值。例 3-1 的运行结果如图 3-1 所示。

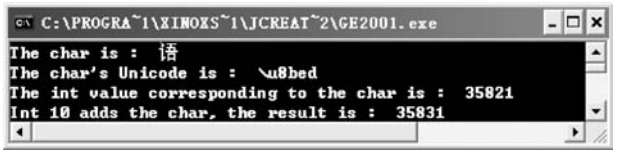

图 3-1 例 3-1 的运行结果

2）String

String 不是基本类型而是一个类。字符串在 Java 中是对象，在 Java 中有两个类可以表达字符串：String 和 StringBuffer。一个 String 的对象表示一个字符串，字符串要放在双引号（""）中。字符串中的字符也是 Unicode。与 C 和 C++不同，Java 中的字符串不以'\0'为结束符。例如：

```
//声明了两个字符串变量并初始化
String greeting = "good morning! \n";
String anotherGreeting = "How are you?";
```

注意：String 对象表示的字符串是不能修改的。如果需要对字符串修改，应该使用 StringBuffer 类。

3. 整数类型：byte，short，int 和 long

Java 提供了 4 种整数类型：byte、short、int 和 long。由于 char 类型的值可以转换为 int 型，所以表 3-3 将这 4 种类型和 char 类型的长度与取值范围一起列出。

表 3-3　Java 整数类型和 char 类型长度与取值范围

类　型	长　度	取 值 范 围
byte	8 位	$-2^7 \sim 2^7-1$，即－128～127
short	16 位	$-2^{15} \sim 2^{15}-1$，即－32 768～32 767
int	32 位	$-2^{31} \sim 2^{31}-1$，即－2 147 483 648～2 147 483 647
long	64 位	$-2^{63} \sim 2^{63}-1$，即－9 223 372 036 854 775 808～9 223 372 036 854 775 807
char	16 位	'\u0000' ～ '\uffff'，即 0～65 535

注意：Java 中所有的整数类型都是有符号的整数类型，Java 没有无符号整数类型。

所有整型变量的默认初始值为 0。

int 类型是最常使用的一种整数类型。它所表示的数据范围足够大，而且适合于 32 位和 64 位处理器。但对于大型计算，常会遇到很大的整数，超出 int 类型所表示的范围，这时要使用 long 类型。

由于不同的机器对于多字节数据的存储方式不同，可能是从低字节向高字节存储，也可能是从高字节向低字节存储，这样，在分析网络协议或文件格式时，为了解决不同机器上的字节存储顺序问题，用 byte 类型来表示数据是比较合适的。而通常情况下，由于其表示的数据范围很小，容易造成溢出，因此要尽量少使用。

如果一个数超出了计算机的表达范围，称为溢出；如果超出最大值，称为上溢；如果超过最小值，称为下溢。将一个整型类型数的最大值加 1 后，产生上溢而变成了同类型的最小值；最小值减 1 后，产生下溢而变成了同类型的最大值。

整型常量可以有 3 种形式：十进制、八进制和十六进制。八进制整数以 0 为前导，十六进制整数以 0X 或 0x 为前导。整型常量的默认类型是 int。对于 long 型常量，则要在数值后加 L 或 l，建议使用 L，因为小写的 L 看起来与数字 1 很相像。表 3-4 是整型常量示例。

表 3-4 整型常量示例

类 型	十进制	八进制	十六进制
int	24	030	0x18
long	24L	030L	0x18L

4. 浮点型：float 和 double

Java 提供了两种浮点类型：float 和 double，如表 3-5 所示。

表 3-5 Java 浮点类型长度与取值范围

类 型	长 度	取 值 范 围
float	32 位	1.4e－45～3.402 823 5e＋38
double	64 位	4.9e－324～1.797 693 134 862 315 7e＋308

双精度类型 double 比单精度类型 float 具有更高的精度和更大的表示范围，但 float 类型具有速度快、占用内存小的优点。

浮点型变量的默认初值是 0.0。浮点数在运算过程中不会因溢出而导致异常处理。如果出现下溢，则结果为 0.0；如果上溢，则结果为正或负无穷大。此外，如果出现数学上没有定义的值，如 0.0/0.0，则结果将被视为非法数，表示为 NaN(Not-a-Number)。

浮点类型的常量默认是 double 类型。如 3.14 是 double 型，在机器内存中占 64 位。浮点型常量还可以用科学记数法表达，用 E 或 e，如 6.02×10^{23} 可表达为 6.02e23。另外可以用 F 或 f 表示 float 类型的常量，如 6.02e23F；用 D 或 d 表示 double 型的常量，如 2.718D。

下面的例 3-2 说明 Java 基本数据类型的使用，例 3-3 显示了 Java 定义基本数据类型相关的常量值，包括各种类型的最大值、最小值以及浮点型中的无穷大与非法数的定义等。

例 3-2 基本数据类型的声明与赋值。

```
public class Assign {
    public static void main (String args[]) {
        int  x, y;                          //声明整型变量
        float  z = 3.414f ;                 //声明并赋值 float 型变量
        double  w = 3.1415;                 //声明并赋值 double 型变量
        boolean  truth = true;              //声明并赋值 boolean 型变量
        char  c;                            //声明字符变量
        String  str;                        //声明 String 类变量
        String  str1 = "bye";               //声明并赋值 String 类变量
        c = 'A';                            //给字符变量赋值
        str = "Hi out there";               //Java 中所有字符串都作为 String 类的对象实现
                                            //所以可采用这种方式给 String 变量赋值
        x = 6;
        y = 1000;                           //给 int 型变量赋值
        System.out.println("x = " + x);
        System.out.println("y = " + y);
        System.out.println("z = " + z);
        System.out.println("w = " + w);
        System.out.println("truth = " + truth);
```

```
        System.out.println("c = " + c);
        System.out.println("str = " + str);
        System.out.println("str1 = " + str1);
    }
}
```

例 3-2 的运行结果如下：

```
x = 6
y = 1000
z = 3.414
w = 3.1415
truth = true
c = A
str = Hi out there
str1 = bye
```

例 3-3　输出 Java 基本数据类型相关的一些常量。

```
public class SomeConstTest{
    public static void main(String args[]){

        //输出 byte 型的最大值与最小值
        System.out.println("Byte.MAX_VALUE = " + Byte.MAX_VALUE);
        System.out.println("Byte.MIN_VALUE = " + Byte.MIN_VALUE);
        System.out.println();

        //输出 short 型的最大值与最小值
        System.out.println("Short.MAX_VALUE = " + Short.MAX_VALUE);
        System.out.println("Short.MIN_VALUE = " + Short.MIN_VALUE);
        System.out.println();

        //输出 int 型的最大值与最小值
        System.out.println("Integer.MAX_VALUE = " + Integer.MAX_VALUE);
        System.out.println("Integer.MIN_VALUE = " + Integer.MIN_VALUE);
        System.out.println();

        //输出 long 型的最大值与最小值
        System.out.println("Long.MAX_VALUE = " + Long.MAX_VALUE);
        System.out.println("Long.MIN_VALUE = " + Long.MIN_VALUE);
        System.out.println();

        //输出 float 型的最大值与最小值
        System.out.println("Float.MAX_VALUE = " + Float.MAX_VALUE);
        System.out.println("Float.MIN_VALUE = " + Float.MIN_VALUE);
        System.out.println();

        //输出 double 型的最大值与最小值
        System.out.println("Double.MAX_VALUE = " + Double.MAX_VALUE);
        System.out.println("Double.MIN_VALUE = " + Double.MIN_VALUE);
        System.out.println();

        //输出 float 型的正无穷大与负无穷大
```

```
        System.out.println("Float.POSITIVE_INFINITY = " + Float.POSITIVE_INFINITY);
        System.out.println("Float.NEGATIVE_INFINITY = " + Float.NEGATIVE_INFINITY);
        System.out.println();

        //输出 double 型的正无穷大与负无穷大
        System.out.println("Double.POSITIVE_INFINITY = " + Double.POSITIVE_INFINITY);
        System.out.println("Double.NEGATIVE_INFINITY = " + Double.NEGATIVE_INFINITY);
        System.out.println();

        //输出 float 型 0/0
        System.out.println("Float.NaN = " + Float.NaN);
        System.out.println();

        //输出 double 型 0/0
        System.out.println("Double.NaN = " + Double.NaN);
        System.out.println();
    }
}
```

例 3-3 的运行结果如图 3-2 所示。

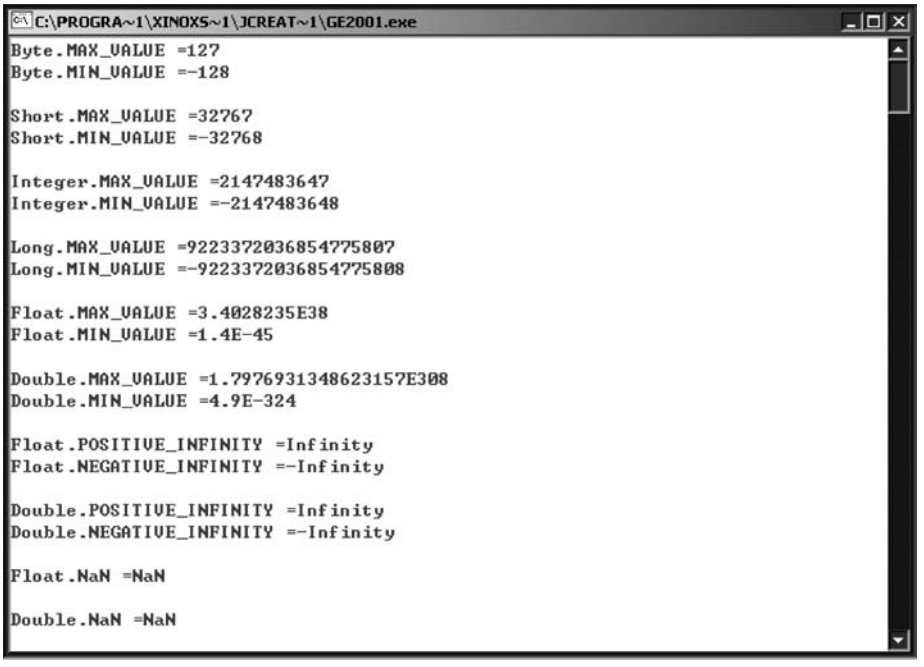

图 3-2　例 3-3 的运行结果

3.1.5　复合数据类型

3.1.4 节介绍的 4 类 8 种基本数据类型,是 Java 的内置类型。在很多应用程序的开发中,仅使用这几种类型是远远不够的。例如,如果要处理日期,则要独立声明 3 个整数,分别代表日、月、年:

```
int  day, month, year ;
```

该语句有个含义：表明 day、month、year 的类型是整数类型，另外还为这些整数分配了存储空间。用这种方式表示日期，虽然容易理解，但存在明显不足。首先如果程序要处理多个日期，则需声明很多变量。例如要保存两个日期，则需如下定义：

```
int  day1, month1, year1;
int  day2, month2, year2;
```

这种方法因使用了多个变量而使程序显得混乱，并且容易出错。另外，这种定义方法忽略了日期的年、月、日之间的联系，把这些变量孤立起来，它们之间将毫无联系，各自的取值范围将只受整数类型取值范围的限制。而在概念上，日、月、年之间是有联系的，它们是同一事物“日期”的各个组成部分，三部分的取值是有约束的。例如，日的取值范围为 0～31，月的取值范围是 1～12，并且对于一般的月份，日的取值上限又有 30 和 31 的差别，而 2 月份的天数又与年有关系。如果要在程序中维护日期的 3 个元素之间的约束，则需要编写程序，并注意在使用日期的这些变量时启动约束检查。这样，不仅增加了程序编写的复杂度，还可能由于程序员的疏忽而造成错误。

如果程序设计语言能够允许用户定义新的数据类型，则上述问题就可以得到很好的解决。而实际上目前很多种语言都具有这种扩展能力，有些语言提供结构或记录来定义新的类型。一般地将用户定义的新类型称为复合数据类型。Java 是一种面向对象的语言，基于面向对象概念，以类和接口的形式定义新类型。因此在 Java 中类和接口是两种用户定义的复合数据类型。

在上述日期的例子中，将日期相关的 3 个变量进行封装，用 class 关键字创建了一个日期类。这个用 class 定义的日期类在 Java 语言中是一种新创建的数据类型。日期类的定义如下：

```
class MyDate{
    int day ;
    int month;
    int year;
}
```

使用语言内置类型定义变量时，因为每种类型都是预定义的，所以无须程序员指定变量的存储结构。例如通过整型变量 day 的定义，Java 运行系统就可以知道要分配多大的空间，并能够解释所存储的内容。对于新的数据类型，需要指定该类型所需的存储空间以及如何解释这些空间中的内容。新类型不是通过字节数指定空间大小，也不是通过位的顺序和含义定义该存储空间的含义，而是通过包含在类定义中的已有数据类型来提供这些信息的。例如，上述类 MyDate 的定义，表明为了表示一个日期，需要保存 3 个整数所需的空间，并且这些整数分别表示了一个日期中的日、月、年。

复合数据类型由程序员在源程序中定义。一旦有了定义，该类型就可像其他类型一样使用。可以声明 MyDate 类的变量，并且日期的年、月、日三部分也都由该变量表示。例如：

```
MyDate  a,b;
```

对于一个日期的年、月、日各组成部分的使用，都是通过 MyDate 类型的 a、b 变量进行，例如：

```
a.day = 30;
```

```
a.month = 12;
a.year = 1999;
```

在定义了 MyDate 类后,对于一个日期的定义只需要声明一个变量,并且日期中的 3 个组成部分被封装为一个有机的整体,它们之间的取值约束关系可以通过在 MyDate 类内部定义方法实现,操纵 MyDate 变量的程序员不需关心这些问题。因此在 Java 中使用类或接口这样的复合数据类型,不仅反映了现实世界事物的本质形态,还使程序简练并且更可靠。

3.1.6 基本类型变量与引用类型变量

Java 中数据类型分为两大类:基本数据类型与复合类型。相应地,变量也有两种类型:基本类型与引用类型。Java 的 8 种基本类型的变量称为基本类型变量,而类、接口和数组变量是引用类型变量。这两种类型变量的结构和含义不同,系统对它们的处理也不相同。

1. 基本类型与引用类型变量

1) 基本类型(primitive type)

基本数据类型的变量包含了单个值,这个值的长度和格式符合变量所属数据类型的要求,可以是一个数字、一个字符或一个布尔值,如图 3-3(a)所示。例如一个整型值是 32 位的二进制补码格式的数据,而一个字符型的值是 16 位的 Unicode 字符格式的数据等。

2) 引用类型(reference type)

引用类型变量的值与基本类型变量不同,变量值是指向内存空间的引用(地址)。所指向的内存中保存着变量所表示的一个值或一组值,如图 3-3(b)所示。

图 3-3 基本类型与引用类型的内存结构

引用在其他语言中称为指针或内存地址。Java 语言与其他程序设计语言不同,不支持显式使用内存地址,而必须通过变量名对某个内存地址进行访问。

2. 两种类型变量的不同处理

在 Java 中基本类型变量声明时,系统直接给该变量分配空间,因此程序中可以直接操作。例如:

```
int a;                                    //声明变量 a 的同时,系统给 a 分配了空间
a = 12;
```

引用类型(或称为引用型)变量声明时,只是给该变量分配引用空间,数据空间未分配。因此引用型变量声明后不能直接引用,下列第二条语句是错误的。

```
MyDate  today;
today.day = 14;                           //错误!因为 today 对象的数据空间未分配
```

引用型变量在声明后必须通过实例化开辟数据空间,才能对变量所指向的对象进行访问。通过对引用型变量声明与实例化语句的执行过程分析,可以理解系统对引用型变量的

上述处理。例如有如下语句：

```
1       MyDate  today;
2       today = new Date();
```

第 1 条语句的执行，将给 today 变量分配一个保存引用的空间，如图 3-4(a)所示。第 2 条语句分两个步骤执行，首先执行 new Date()，给 today 变量开辟数据空间，如图 3-4(b)所示，然后再执行第 2 条语句中的赋值操作，结果如图 3-4(c)所示。

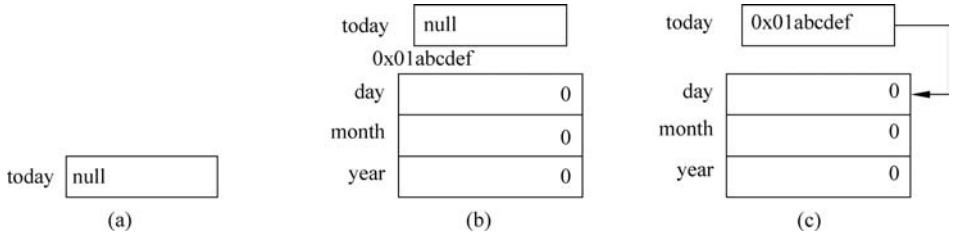

图 3-4　引用型变量的声明与实例化过程

3. 引用型变量的赋值

Java 中引用型变量之间的赋值是引用赋值。例如，下列语句执行后，内存的布局如图 3-5 所示。

```
MyDate  a, b;                  //在内存开辟两个引用空间
a =  new MyDate();             //开辟 MyDate 对象的数据空间,并把该空间的首地址赋给 a
b = a;                         //将 a 存储空间中的地址写到 b 的存储空间中
```

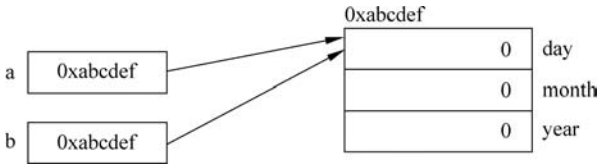

图 3-5　引用型变量之间赋值示例

3.2　表达式与语句

3.2.1　变量

1. 变量及作用域

变量有基本类型与引用类型。同时变量按照作用域来分，有局部变量、类成员变量、方法参数和异常处理参数。

1）局部变量

在一个方法或由一对{}表示的代码块内定义的变量称为局部变量，有时也称为自动变量、临时变量或堆栈变量。局部变量的作用域是所在的方法或代码块，当程序执行流进入所在方法(或代码块)时创建，在方法(或代码块)退出时消亡，因此也称为自动变量或临时变量。

2) 类成员变量

在方法外进行声明且属于一个类的定义体的变量称为类成员变量。类成员变量的作用域是整个类,具体可以有两种类型:一种是用 static 关键字声明的类变量,该变量在类加载时创建并且只要所属的类存在,该变量就将一直存在;另一种是声明中没有 static 关键字的变量,称为实例变量。实例变量在调用类的构造方法(new *XXX*())创建实例对象时创建,并且只要有引用指向变量所属的对象,该变量就将存在。

3) 方法参数

方法参数定义了方法调用时传递的参数,其作用域就是所在的方法。当方法被调用时创建方法参数变量,而在方法运行结束时,这些变量就消亡了。

4) 异常处理器(catch 语句块)参数

异常处理器参数是 catch 语句块的入口参数。这种参数的作用域是 catch 语句后由{}表示的语句块。

各种变量及其作用域如图 3-6 所示。

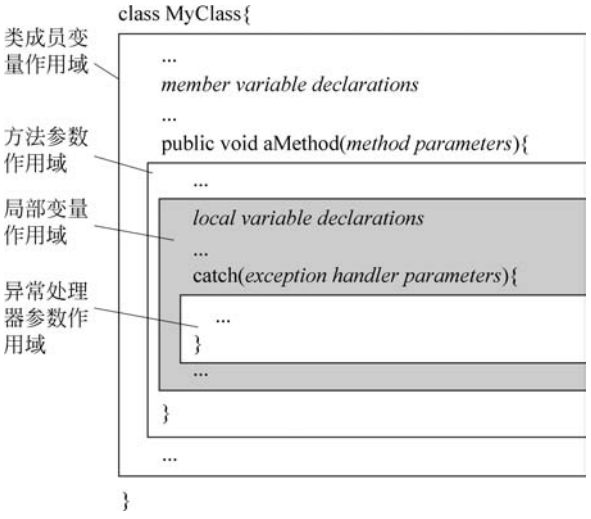

图 3-6 变量及其作用域

2. 变量的初始化

在 Java 程序中,变量在使用前必须经过初始化。当创建一个对象时,对象所包含的实例变量在存储空间分配后就由系统按照表 3-6 列出的值进行初始化。

表 3-6 各种类型变量的初始值

变量类型	初始值	变量类型	初始值
byte	0	double	0.0D
short	0	char	'\u0000'
int	0	boolean	false
long	0L	所有引用型(类、接口、数组)	null
float	0.0F		

因此，类成员变量是系统自动进行初始化，而局部变量必须在使用前手工赋初值进行初始化。如果编译器确定局部变量没有经过初始化，就将产生编译错误，如例 3-4 所示。

例 3-4　局部变量的初始化。

```
1  public class TestInit{
2        int x;
3
4    public static void main(String args[]){
5        TestInit init = new TestInit();
6
7        int x = (int)(Math.random() * 100);
8        int z;
9        int y;
10
11       if (x > 50){
12          y = 9;
13       }
14
15       z = x + y + init.x;
16
17       System.out.println("x = " + x + " y = " + y + " z = " + z);
18    }
19 }
```

例 3-4TestInit.java 的编译结果如图 3-7 所示。

图 3-7　例 3-4 的编译结果

将例 3-4 中的第 9 行语句改为 int y=0；或在第 15 行语句前能够给变量 y 赋值，则程序可以通过编译并运行。例如，将第 9 行语句改为 int y=0 后，例 3-4 的运行结果为：

```
x = 81 y = 9 z = 90 init.x = 0
```

3.2.2　运算符与表达式

Java 的运算符与标准 C 基本相同，C 语言中提供的运算符几乎完全适合于 Java，但有两方面不同。首先，Java 是强类型语言，其类型限制比 C 语言严格，表现在表达式上就是运算符的操作对象类型会受到更多的限制。其次，C 语言提供的指针运算符等，在 Java 中不再提供，而 Java 增加了对象操作符 instanceof、字符串运算符“+”和零填充的右移“>>>”等。

1. 概述

Java 中，操作符可以分成如下的类别：算术运算操作符、关系和条件操作符、位操作符、逻辑操作符和赋值操作符。

(1) 算术运算符：+、-、*、/、%、++、--。

(2) 关系运算符：>、>=、<、<=、==、!=。

(3) 逻辑运算符：&、|、!、^、&&、||。

(4) 位操作符：>>、<<、>>>、&、|、^、~。

(5) 赋值操作符：=、+=、-=、*=、/=、%=、&=、|=、^=、<<=、>>=、>>>=。其中<变量><赋值运算符><表达式>等价于<变量>=<变量><运算符><表达式>，例如 a+=6 等价于 a=a+6。另外，赋值运算符遵循从右向左的结合性。例如 a=b=c=5 等价于 a=(b=(c=5))；a=5+(c=6)-(d=2)的执行结果是：d=2，c=6，a=9。

2. 算术运算符和算术表达式

算术表达式由操作数和算术运算符组成。在算术表达式中，操作数只能是整型或浮点型。Java 的算术运算符有两种：二元算术运算符和一元算术运算符。

二元算术运算符涉及两个操作数，共有 5 种：+，-，*，/，%，如表 3-7 所示。这些算术运算符适用于所有数值型数据类型。

表 3-7 二元算术运算符

运算符	表达式	名称及功能
+	op1+op2	加
-	op1-op2	减
*	op1 * op2	乘
/	op1/op2	除
%	op1%op2	模数除(求余)

整型、浮点型经常进行混合运算。运算中，不同类型的数据先转换为同一类型，然后进行运算。这种转换是按照如下优先关系自动进行的。

低 ——————————————————————→ 高

byte ⟶ short ⟶ char ⟶ int ⟶ long ⟶ float ⟶ double

按照这种优先级关系，在混合运算中低级数据转换为高级数据时，自动进行类型转换。转换规则如表 3-8 所示。

表 3-8 低级数据向高级数据的自动转换规则

操作数 1 类型	操作数 2 类型	转换后类型
byte 或 short	int	int
byte，short 或 int	long	long
byte，short，int 或 long	float	float
byte，short，int，long 或 float	double	double
char	int	int

注意：

(1) 即使两个操作数全是 byte 型或 short 型，表达式的结果也是 int 型。

(2) /运算和%运算中除数为 0 时，会产生异常。

(3) 与 C 和 C++不同,取模运算符%的操作数可以为浮点数,如 9.5%3=0.5。

(4) +运算符可以用来连接字符串。例如,

```
String  salutation = "Dr.  ";
String  name = "Peter" + "Symour";
String  title = salutation + name;
```

则 title 值为: Dr. Peter Seymour。

下面给出一些混合算术运算的例子,如表 3-9 所示。

表 3-9　混合算术运算示例

操作数 1	操作数 2	算术运算表达式	表达式结果及其类型
8	3	8/3	2,int
5	2.0	5/2.0	2.5,double
byte i=4	byte j=7	i*j	28,int
long r=40L	int a=2	r/a	20L,double
float x=6.5f	float y=3.1f	x+y	9.6f,float
double b=2.5	int a=2	b%a	0.5,double
float x=6.5f	int c=28	x-c	-21.5f,float
'a'	60	'a'+60	157,int

一元算术运算符所涉及的操作数只有一个,共有 4 种: +、-、++、--。各种一元算术运算符用法以及功能如表 3-10 所示。

表 3-10　Java 一元算术运算符

运算符	用法	功 能 描 述
+	+op	如果 op 是 byte,short 或 char 类型,则将 op 类型提升为 int 型
-	-op	取 op 的负值
++	op++	op 加 1,先求 op 的值再把 op 加 1
++	++op	op 加 1,先把 op 加 1 再求 op 的值
--	op--	op 减 1,先求 op 的值再把 op 减 1
--	--op	op 减 1,先把 op 减 1 再求 op 的值

下面举一个一元算术运算符使用的例子。

例 3-5　一元算术运算符的使用。

```
public class TestUnary{
    public static void main(String args[]){
        int a = 9;
        int b = - a;
        byte bb = 9;
        int ib = + bb;
        int x = 4, y = 8;
        int z;
        int i = 0;
        int j = i++ ;
        int k = ++ j;
```

```
        z = (x++) * (--y);

        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("bb = " + bb);
        System.out.println("ib = " + ib);
        System.out.println("i = " + i);
        System.out.println("j = " + j);
        System.out.println("k = " + k);
        System.out.println("x = " + x);
        System.out.println("y = " + y);
        System.out.println("z = " + z);
    }
}
```

例 3-5 的运算结果如下。

```
a = 9
b = -9
bb = 9
ib = 9
i = 1
j = 1
k = 1
x = 5
y = 7
z = 28
```

另外,Java 中没有幂运算符,必须采用 Java.lang.Math 类的 pow()方法。该方法的定义如下:

```
public static double pow(double a,double b);     //返回值是 a^b
```

Java.lang.Math 类提供了大量数学和工程函数。例如,π 和 e 均由常数表示,而且为双精度类型,所以非常精确。Math 类包含了平方根、自然对数、乘幂、三角函数等数学函数。另外,它也包含了一些基本函数,如可对浮点数进行四舍五入运算,计算同样类型两个数字的最大值和最小值,计算绝对值等。

3. 关系运算符与关系表达式

关系运算符用来比较两个操作数,由两个操作数和关系运算符构成一个关系表达式。关系运算符的操作结果是布尔类型的,即如果运算符对应的关系成立,则关系表达式结果为 true,否则为 false。关系运算符都是二元运算符,共有 6 种,如表 3-11 所示。

例如:

表达式 3 > 5 的值为 false;

表达式 3 <=5 的值为 true;

表达式 3==5 的值为 false;

表达式 3!=5 的值为 true。

表 3-11　关系运算符

运算符	表达式	返回 true 值时的情况
>	op1 > op2	op1 大于 op2
<	op1 < op2	op1 小于 op2
>=	op1 >= op2	op1 大于或等于 op2
<=	op1 <= op2	op1 小于或等于 op2
==	op1 == op2	op1 等于 op2
!=	op1 != op2	op1 不等于 op2

Java 中，任何类型的数据（包括基本数据类型和复合类型）都可以通过==或!=来比较是否相等（这与 C 和 C++不同）。

4. 逻辑运算符与逻辑表达式

逻辑表达式由逻辑型操作数和逻辑运算符组成。一个或多个关系表达式可以进行逻辑运算。Java 中逻辑运算符共有 6 种，即 5 个二元运算符和 1 个一元运算符。这些运算符及其使用方法、功能与含义，如表 3-12 所示。

表 3-12　逻辑运算符

运算符	用　法	返回 true 时的情况
&&	op1&&op2	op1 和 op2 都为 true，并且在 op1 为 true 时才求 op2 的值
||	op1||op2	op1 或 op2 为 true，并且在 op1 为 false 时才求 op2 的值
!	!op	op 为 false
&	op1&op2	op1 和 op2 都为 true，并且总是计算 op1 和 op2 的值
|	op1|op2	op1 或 op2 为 true，并且总是计算 op1 和 op2 的值
^	op1^op2	op1 和 op2 的值不同，即一个取 true，另一个取 false

注意：表 3-12 中，有两种“与”和“或”的运算符：&&、||以及 &、|。这两种运算符的运算过程有所差别，如下所述。

&&、||：称为短路与、或运算。表达式求值过程中先求出运算符左边的表达式的值，对于或运算如果为 true，则整个布尔逻辑表达式的结果确定为 true，从而不再对运算符右边的表达式进行运算；同样对于与运算，如果左边表达式的值为 false，则不再对运算符右边的表达式求值，整个布尔逻辑表达式的结果已确定为 false。

&、|：称为不短路与、或运算，即不管第一个操作数的值是 true 还是 false，仍然要把第二个操作数的值求出，然后再做逻辑运算求出逻辑表达式的值。

5. 位运算符

对于任何一种整数类型的数值，可以直接使用位运算符对这些组成整型的二进制位进行操作。这意味着可以利用屏蔽和置位技术来设置或获得一个数字中的单个位或几位，或者将一个位模式向右或向左移动。由位运算符和整型操作数组成位运算表达式。

位运算符分为位逻辑运算符和位移位运算符。

1）位逻辑运算符

位逻辑运算符有 3 个二元运算符和 1 个一元运算符。二元运算中，是在两个操作数的

每个对应位上进行相应的逻辑运算。一元运算符是对操作数按位进行相应的运算。位逻辑运算符以及操作描述如表 3-13 所示。表 3-14 给出了位逻辑运算 &、|、^、~ 的表达式取值。

表 3-13 位逻辑运算符的操作规则

运 算 符	位运算表达式	操 作 描 述
&	op1&op2	按位与
|	op1|op2	按位或
^	op1^op2	按位异或
~	~op2	按位取反

表 3-14 位逻辑运算表达式取值规则

op1	op2	op1&op2
0	0	0
0	1	0
1	0	0
1	1	1

op1	op2	op1|op2
0	0	0
0	1	1
1	0	1
1	1	1

op1	op2	op1^op2
0	0	0
0	1	1
1	0	1
1	1	0

按位取反运算符 ~对数据的每个二进制位取反,即把 1 变为 0,把 0 变为 1。

下面是几个位逻辑运算的例子:

00101101&01001111=00001101;

00101101|01001111=01101111;

00101101^01001111=01100010;

~01001111=10110000。

2) 移位运算符

Java 使用补码来表示二进制数。因此移位运算都是针对整型数的二进制补码进行。在补码表示中,最高位为符号位,正数的符号位为 0,负数为 1。补码的规定如下。

(1) 对正数来说,最高位为 0,其余各位代表数值本身(以二进制表示),如+42 的补码为 00101010。

(2) 对负数来说,把该数绝对值的补码按位取反,然后对整个数加 1,即得该数的补码。如-1 的补码为 11111111(-1 绝对值的补码为 00000001,按位取反再加 1 为 11111110+1=11111111)。用补码来表示数,0 的补码是唯一的,为 00000000。

移位运算符把它的第一个操作数向左或向右移动一定的位数。Java 中的位运算符及其操作的描述,如表 3-15 所示。

表 3-15 移位运算符

运 算 符	用 法	操 作 描 述
>>	op1 >> op2	将 op1 向右移动 op2 个位
<<	op1 << op2	将 op1 向左移动 op2 个位
>>>	op1 >>> op2	将 op1 向右移动 op2 个位(无符号)

注意：

(1) 右移运算中右移一位相当于除 2 取商；在不产生溢出的情况下，左移一位相当于乘 2，并且用移位运算实现乘除法比执行乘除法的速度要快。例如：

$-256 >> 4$ 结果是 $-256/2^4 = -16$

$128 >> 1$ 结果是 $128/2^1 = 64$

$-16 << 2$ 结果是 $-16 * 2^2 = -64$

$128 << 1$ 结果是 $128 * 2^1 = 256$

(2) 右移运算符>>和>>>之间的区别。

- >>称为带符号的右移。进行向右移位运算时，最高位移入原来高位的值，即移位操作是对符号位的复制。例如：

1010…>> 2 结果是 111010…

- >>>称为无符号右移。进行向右移位运算时，最高位以 0 填充。例如：

1010…>>> 2 结果是 001010…

下面是移位运算的例子：

1357＝00000000 00000000 00000101 01001101

－1357＝11111111 11111111 11111010 10110011

则：

1357 >> 5＝00000000 00000000 00000000 00101010

－1357 >> 5＝11111111 11111111 11111111 11010101

1357 >>> 5＝00000000 00000000 00000000 00101010

－1357 >>> 5＝00000111 11111111 11111111 11010101

1357 << 5＝00000000 00000000 10101001 101000000

－1357 << 5＝11111111 11111111 01010110 01100000

(3) 逻辑运算的运算符 &、|、^与位逻辑运算的运算符 &、|、^相同，实际运算时根据操作数的类型判定进行何种运算。如果操作数的类型是 boolean，则进行逻辑运算；如果操作数的类型是整数类型，则进行位逻辑运算。

6. 赋值运算符和赋值表达式

赋值表达式由变量、赋值运算符和表达式组成。赋值运算符把一个表达式的值赋给一个变量。赋值运算符分为赋值运算符“＝”和扩展赋值运算符两种。在赋值运算符两侧的类型不一致的情况下，如果左侧变量类型的级别高，则右侧的数据被转化为与左侧相同的高级数据类型后赋给左侧变量；否则，需要使用强制类型转换运算符。例如：

```
byte  b = 121;
int   i = b;                    //自动类型转换
byte  c = 13;
byte  d = (byte)(b + c);        //强制类型转换
```

在赋值运算符“＝”前加上其他运算符，则构成扩展赋值运算符。表 3-16 列出了 Java 中的扩展赋值运算符及等价的表达式。扩展赋值运算符的特点是可以使程序表达简练，并且还能提高程序的编译速度。例如：

- b%＝6 等价于 b＝b%6；
- a＋＝3 等价于 a＝a＋3。

表 3-16 Java 中的扩展赋值运算符

运算符	表 达 式	等效表达式
+=	op1+=op2	op1=op1+op2
-=	op1-=op2	op1=op1-op2
*=	op1*=op2	op1=op1*op2
/=	op1/=op2	op1=op1/op2
%=	op1%=op2	op1=op1%op2
&=	op1&=op2	op1=op1&op2
|=	op1|=op2	op1=op1|op2
^=	op1^=op2	op1=op1^op2
>>=	op1>>=op2	op1=op1>>op2
<<=	op1<<=op2	op1=op1<<op2
>>>=	op1>>>=op2	op1=op1>>>op2

7. 其他运算符

Java 支持的其他运算符的使用方法与功能描述如表 3-17 所示。

表 3-17 其他运算符

运算符	格 式	操 作 描 述
?:	op1? op2: op3	条件运算符。如果 op1 是 true,返回 op2,否则返回 op3
[ ]	type[]	声明类型为 type 的数组
[ ]	type[op1]	创建 op1 个元素的数组
[]	op1[op2]	访问 op1 数组的索引为 op2 的元素
.	op1.op2	引用 op1 对象的成员 op2
( )	op1(params)	方法调用
(type)	(type)op1	将 op1 强制类型转换为 type 类型
new	new op1	创建对象或数组。op1 可以是构造方法的调用,也可以是数组的类型和长度
instanceof	op1 instanceof op2	如果 op1 是 op2 的实例,则返回 true

8. 运算符的优先级

具有两个或两个以上运算符的复合表达式在进行运算时,要按表 3-18 所示的运算符的优先级顺序从高到低进行,同级的运算符则按从左到右的方向进行。

表 3-18 Java 运算符的优先级

优先级顺序	运算符类别	运 算 符
1	后缀运算符	[],.,(*params*)*expr*++,*expr*--
2	一元运算符	++*expr*,--*expr*,+*expr*,-*expr*,~,!
3	创建或强制类型转换	new(*type*)*expr*
4	乘、除、求余	*,/,%
5	加、减	+,-
6	移位	<<,>>,>>>
7	关系运算	<,>,<=,>=,instanceof
8	相等性判定	==,!=

续表

优先级顺序	运算符类别	运　算　符
9	按位与	&
10	按位异或	^
11	按位或	|
12	逻辑与	&&
13	逻辑或	||
14	条件运算	?:
15	赋值	=,+=,-=,*=,/=,%=,&=,^=,|=,<<=,>>=,>>>=

9. Java 强制类型转换

造型(casting)的含义是把一种类型的值赋给另外一种类型的变量。如果这两种类型是兼容的,则是低优先级类型的值赋给高优先级类型的变量,例如一个 int 型值可以赋值给一个 long 型的变量,则 Java 自动执行转换。如果将高优先级类型的值赋给低优先级的变量,例如将一个 long 型值赋给一个 int 型变量,则可能造成信息的丢失。这时,Java 不能执行自动转换,编译器需要程序员通过强制类型转换方式确定这种转换。例如:

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

Java 通过强制类型转换将一表达式类型强制为某一指定类型,其一般形式为:

```
(type)expression
```

引用型变量也可以进行造型,将在本书后续章节中论述。但要注意基本类型和数组或对象等引用型变量之间不能相互转换。

3.2.3　语句

Java 语言中语句以“;”为终结符,一条语句构成了一个执行单元。Java 中有如下 3 类语句。

1. 表达式语句

下列表达式以语句分隔符“;”终结,则构成了语句,称为表达式语句。

- 赋值表达式;
- 增量表达式(使用++或--);
- 方法调用表达式;
- 对象创建表达式。

表达式语句举例:

```
aValue = 8933.234;                          //赋值语句
aValue++;                                   //增量语句
System.out.println(aValue);                 //方法调用语句
Integer integerObject = new Integer(4);     //对象创建语句
```

2. 声明语句

声明变量或方法。例如：

```
double aValue = 8933.234;                    //声明语句
```

3. 程序流控制语句

程序流控制语句控制程序中语句的执行顺序。例如，for 循环和 if 语句都是程序流控制语句。

语句块由“{ }”括起来的 0 个或多个语句组成，可以出现在任何单个语句可以出现的位置。例如：

```
if (Character.isUpperCase(aChar)) {
    System.out.println("The character " + aChar + " is upper case.");
} else {
    System.out.println("The character " + aChar + " is lower case.");
}
```

在程序流控制语句中，即使只有一条语句也最好使用语句块，能够增强程序可读性并且在以后对代码进行增删时不易发生语法错误。

3.3 程序流控制

程序流控制语句可以使程序运行时，有条件地执行或重复执行某些语句，改变程序正规的顺序执行流向。Java 语言提供了如下 4 类程序流控制语句。

- 循环语句：包括 while 语句、do while 语句和 for 语句。
- 分支语句：包括 if 语句和 switch 语句。
- 跳转语句：包括 break 语句、continue 语句、label 语句、return 语句。
- 异常处理语句：包括 try catch finally 语句和 throw 语句。

本节介绍前 3 种程序流控制语句，异常处理语句将在后续章节中结合 Java 的异常处理机制进行专门介绍。

3.3.1 while 和 do while 语句

1. while 语句

while 语句使一个语句块在某种条件为真时循环执行。该语句的语法如下：

```
while (逻辑表达式) {
    语句或语句块
}
```

while 语句中的表达式必须是逻辑型的。在该语句执行时，首先求表达式的值，如果得到的值为 true，则执行 while 语句块中的语句。while 语句将一直测试表达式的值并执行语句块，直到表达式的值变为 false 时为止。

在例 3-6 中，使用一个 while 语句把一个字符串中字符'g'之前的字符逐一添加到一个字符串缓冲区。

例 3-6　用 while 语句复制字符串。

```
public class WhileDemo {
    public static void main(String[ ] args) {
        String copyFromMe = "Copy this string until you encounter the letter 'g'.";
        StringBuffer copyToMe = new StringBuffer();
        int i = 0;
        char c = copyFromMe.charAt(i);
        while (c != 'g') {
            copyToMe.append(c);
            c = copyFromMe.charAt( ++ i);
        }
        System.out.println(copyToMe);
    }
}
```

例 3-6 的运行结果如下：

```
Copy this strin
```

2. do while 语句

do while 语句的语法如下：

```
do {
    语句或语句块
} while(逻辑表达式);
```

do while 语句在执行时，首先执行循环体中的语句，然后再求表达式的值。do while 语句将一直执行语句块并测试表达式的值，直到表达式的值变为 false。do while 语句是在循环体的底部求得表达式的值。因此 do while 语句至少要把循环体中的语句执行一遍。

例 3-7 是用 do while 语句对例 3-6 进行了改写。

例 3-7　用 do while 语句复制字符串。

```
public class DoWhileDemo {
    public static void main(String[ ] args) {
        String copyFromMe = "Copy this string until you encounter the letter 'g'.";
        StringBuffer copyToMe = new StringBuffer();
        int i = 0;
        char c = copyFromMe.charAt(i);
        do {
            copyToMe.append(c);
            c = copyFromMe.charAt( ++ i);
        } while (c != 'g');
        System.out.println(copyToMe);
    }
}
```

例 3-7 的运行结果如下：

```
Copy this strin
```

3.3.2 for 语句

for 循环执行的次数是可以在执行前确定的。for 语句的语法格式是:

```
for (初始语句; 逻辑表达式; 迭代语句) {
    语句或语句块
}
```

for 语句执行时先执行初始语句,判断逻辑表达式的值,当逻辑表达式为 true 时,执行循环体语句,接着执行迭代语句,然后再去判别逻辑表达式的值。这个过程一直进行下去,直到逻辑表达式的值为 false,循环结束并转到 for 之后的语句。关于 for 语句有以下几点说明。

(1) 可以在 for 循环的初始化部分声明一个变量,它的作用域为整个 for 循环。

(2) for 循环通常用于循环次数确定的情况,但也可以根据循环结束条件完成循环次数不确定的情况。

(3) 在初始化部分和迭代部分可以使用逗号语句来进行多个操作。逗号语句是用逗号分隔的语句序列。例如:

```
for(i = 0,j = 10; i < j; i++ ,j-- ){
  …
}
```

(4) 初始化、终止以及迭代部分都可以为空语句(但分号不能省略),逻辑表达式为空时,默认表达式为恒真,可以用下列 for 语句表示无限循环。

```
for( ; ; ){
  …
}
```

for 循环经常用于数组各个元素或一个字符串中各个字符的处理。在例 3-8 中,for 语句用来打印一个数组的所有元素。

例 3-8 用 for 语句输出一个数组的元素。

```
public class ForDemo {
    public static void main(String[] args) {
        int[] arrayOfInts = {32, 87, 3, 589, 12, 1076, 2000, 8, 622, 127};

        for (int i = 0; i < arrayOfInts.length; i++ ) {
            System.out.print(arrayOfInts[i] + " ");
        }
        System.out.println();
    }
}
```

例 3-8 的运行结果如下:

```
32 87 3 589 12 1076 2000 8 622 127
```

3.3.3　if else 语句

if 语句使程序能够基于某种条件有选择地执行某些语句。

if 语句的语法格式是：

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

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。

注意：

(1) 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来代替。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 中应该写为：

```
int x = 3;
if (x!= 0){…}
```

(2) else 语句的另外一种形式是 else if 语句。可以利用 else if 语句构造嵌套的 if 语句。一个 if 语句可以有多个 else if 语句，但只能有一个 else 语句。

例 3-9 根据一个百分制的成绩值，输出相应的等级，如 90 分以上是 A，80 分以上是 B 等。

例 3-9　利用 if else 语句输出成绩等级。

```
public class IfElseDemo {
    public static void main(String[] args) {
        int testscore = 76;
        char grade;

        if (testscore >= 90) {
            grade = 'A';
        } else if (testscore >= 80) {
            grade = 'B';
        } else if (testscore >= 70) {
            grade = 'C';
        } else if (testscore >= 60) {
            grade = 'D';
        } else {
            grade = 'F';
        }
        System.out.println("Grade = " + grade);
```

```
    }
}
```

例 3-9 的运行结果如下：

```
Grade = C
```

在 3.1 节介绍的条件运算符“?：”是 if else 语句的一种紧缩格式的表达。某些情况下使用条件表达式会使程序更简洁易读。例如下列 if else 语句可以用等价的条件表达式表达。

```
if (Character.isUpperCase(aChar)) {
    System.out.println("The character " + aChar + " is upper case.");
} else {
    System.out.println("The character " + aChar + " is lower case.");
}
```

用条件表达式将上述 if else 语句表达为：

```
System.out.println("The character " + aChar + " is " +
                   (Character.isUpperCase(aChar) ? "upper" : "lower") + "case.");
```

3.3.4 switch 语句

switch 语句是基于一个表达式的值决定要执行的一组语句。Switch 语句的语法格式如下：

```
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; ]
}
```

switch 语句的语义是：计算表达式的值，用该值依次和 c_1，c_2，…，c_k 相比较。如果该值等于其中之一，例如 c_i，那么执行 case c_i 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 c_i，则执行 default 之后的语句。

说明：

(1) switch 中的表达式的值，可以是整型、枚举类型或 String 类的对象。整型表达式的值须是 int 兼容的类型，即可以是 byte，short，char 和 int，不允许使用浮点型(float，double)或 long 型，并且各 case 子句中的 c_1，c_2，…c_k 是 int 型或字符型常量。在 Java SE 7 以后的

版本中，switch 的表达式可以使用 String 类的对象，例如下面的代码：

```
public static int getMonthNumber(String month) {
    int monthNumber = 0;
    if (month == null) {
        return monthNumber;
    }
    switch (month.toLowerCase()) {
        case "january": monthNumber = 1;
                        break;
        case "february": monthNumber = 2;
                         break;
        …
    }
    return monthNumber;
}
```

(2) switch 语句中各 case 分支既可以是单条语句，也可以是由多条语句组成的语句组，该语句组可以不用{}括起来。

(3) default 子句是可选的，并且最后一个 break 语句可以省略。

(4) 不论执行哪个 case 分支，程序流都会顺序执行下去，直到遇到 break 语句为止。可以利用这个特点简化程序的编写，如例 3-10(b)。

(5) switch 结构的功能可以用 if else if 结构来实现，但在某些情况下，使用 switch 结构更简单，可读性强，而且程序的执行效率也得到提高。但 switch 结构在数据类型上受到了限制，如果要比较的数据类型是 double 型，则不能使用 switch 结构。

下面给出两个使用 switch 语句的例子。

例 3-10(a)　输出一个月份的英文名称。

```
public class SwitchDemo {
    public static void main(String[] args) {
        int month = 8;
        switch (month) {
            case 1:  System.out.println("January"); break;
            case 2:  System.out.println("February"); break;
            case 3:  System.out.println("March"); break;
            case 4:  System.out.println("April"); break;
            case 5:  System.out.println("May"); break;
            case 6:  System.out.println("June"); break;
            case 7:  System.out.println("July"); break;
            case 8:  System.out.println("August"); break;
            case 9:  System.out.println("September"); break;
            case 10: System.out.println("October"); break;
            case 11: System.out.println("November"); break;
            case 12: System.out.println("December"); break;
            default: System.out.println("Hey, that's not a valid month!"); break;
        }
    }
```

```
}
```

例 3-10(a)的运行结果如下：

```
August
```

例 3-10(b) 输出一个日期所包含月份的天数。

```
public class SwitchDemo2 {
    public static void main(String[ ] args) {
        int month = 2;
        int year = 2000;
        int numDays = 0;

        switch (month) {
            case 1:
            case 3:
            case 5:
            case 7:
            case 8:
            case 10:
            case 12:
                numDays = 31;
                break;
            case 4:
            case 6:
            case 9:
            case 11:
                numDays = 30;
                break;
            case 2:
                if (((year % 4 = = 0) && !(year % 100 = = 0))
                    || (year % 400 = = 0))
                    numDays = 29;
                else
                    numDays = 28;
                break;
        }
        System.out.println("The date is 2000.2. The number of Days = " + numDays);
    }
}
```

例 3-10(b)的运行结果如下：

```
The date is 2000.2. The number of Days = 29
```

3.3.5 switch 语句扩展与 switch 表达式

3.3.4 节中介绍的传统 switch 存在一些不方便之处，包括：switch 标签之间的控制流贯穿问题，即一个 case 分支执行后如果没有遇到 break 语句，会继续执行之后 case 分支的

语句，直到遇到 break 为止；switch 中各语句块的缺省作用域是整个 switch 语句；switch 是语句而不是表达式，而很多时候需要一种表达多分支条件的表达式。在 Java SE 12 和 Java SE 13 中，对 switch 进行扩展，使上述问题得到解决。

switch 的扩展，主要体现在 case 分支的定义，有两种形式：传统的“case …：”分支（被称为“冒号 case”），以及新的“case …->”分支（被称为“箭头 case”）。箭头 case 可以带有一个为 switch 表达式产生值的语句。“case …->”分支定义的具体形式如下：

```
case label_1, label_2, …, label_n -> expression;|throw-statement;|block
```

上述定义中，允许在一个 case 分支里出现以逗号分隔的多个常量值，如 label_1，label_2，…，label_n。程序运行时，这些值中的任何一个匹配成功，箭头右侧的代码都将被运行，并且在这些代码结束后，将不会运行 switch 表达式或语句中任何其他分支中的代码。因此，箭头 case 解决了传统冒号 case 的控制流贯穿问题。箭头右侧的代码可以是一个表达式、一个语句块（包含多个语句）或一个 throw 语句。在语句块中声明的局部变量，其作用域只是所在的语句块而不是整个 switch 语句。如果箭头右侧是一个表达式，则该表达式的值将是 switch 表达式的值。

例如，下列代码使用了箭头 case 的 switch 语句：

```
int numLetters = 0;
Day day = Day.WEDNESDAY;
switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> numLetters = 6;
    case TUESDAY -> numLetters = 7;
    case THURSDAY, SATURDAY -> numLetters = 8;
    case WEDNESDAY -> numLetters = 9;
    default -> throw new IllegalStateException("Invalid day: " + day);
};
System.out.println(numLetters);
```

下列代码使用了箭头 case 的 switch 表达式：

```
Day day = Day.WEDNESDAY;
System.out.println(
    switch (day) {
        case MONDAY, FRIDAY, SUNDAY -> 6;
        case TUESDAY -> 7;
        case THURSDAY, SATURDAY -> 8;
        case WEDNESDAY -> 9;
        default -> throw new IllegalStateException("Invalid day: " + day);}
    }
);
```

Java SE 13 中，引入了 yield 语句，其定义形式为：yield someValue ，其中 someValue 是 yield 所在 case 分支产生的 switch 表达式的值。例如下面的代码：

```
int j = switch (day) {
    case MONDAY -> 0;
    case TUESDAY -> 1;
```

```
        default -> {
            int k = day.toString().length();
            int result = f(k);
            yield result;
        }
    };
```

Break 和 yield 语句使得 switch 语句和 switch 表达式易于区分：yield 语句用于 switch 表达式，而 break 语句用于 switch 语句。传统的冒号 case 可以使用 yield 语句构造 switch 表达式。例如下面的代码：

```
int result = switch (s) {
    case "Foo":
            yield 1;
    case "Bar":
            yield 2;
    default:
            System.out.println("Neither Foo nor Bar, hmmm...");
            yield 0;
};
```

3.3.6 循环跳转语句

Java 中可以用 break 和 continue 两个循环跳转语句进一步控制循环。这两个语句的一般格式如下：

```
break [label] ;          //用来从 switch 语句或循环语句中跳出
continue [label] ;       //跳过循环体的剩余语句,开始执行下一次循环
```

这两个语句都可以带标签(label)使用，也可以不带标签使用。标签是出现在一个语句之前的标识符，标签后面要跟上一个冒号(：)，标签的定义如下：

```
label: statement;
```

下面分别对这两种语句进行介绍。

1. break 语句

break 语句有两种形式：不带标签和带标签。在 switch 语句的介绍中，已经了解了不带标签的 break 在 switch 语句中的作用，它结束了 switch 语句的执行，并把控制流转移到紧跟在 switch 之后的语句。还可以使用不带标签的 break 语句终止循环，包括 for、while 和 do while。不带标签的 break 语句用来结束最内层的 switch、for、while 和 do while 语句的执行。例 3-11 中，包含了一个在数组中搜索指定值的 for 循环。指定的值在数组中找到后，则结束 for 循环，进行结果的输出。

例 3-11 在数组中寻找指定的值。

```
public class BreakDemo {
    public static void main(String[] args) {
        int[] arrayOfInts = { 32, 87, 3, 589, 12, 1076, 2000, 8, 622, 127 };
        int searchfor = 12;
```

```
        int i = 0;
        boolean foundIt = false;
        for ( ; i < arrayOfInts.length; i++ ) {
            if (arrayOfInts[i] = = searchfor) {
                foundIt = true;
                break;
            }
        }

        if (foundIt) {
            System.out.println("Found " + searchfor + " at index " + i);
        } else {
            System.out.println(searchfor + "not in the array");
        }
    }
}
```

例 3-11 的运算结果如下：

```
Found 12 at index 4
```

break 语句后可带有标签。带标签的 break 语句将结束标签所指示的循环的执行。在例 3-12 中，程序为寻找一个值，通过两个嵌套的 for 循环遍历一个二维数组。当指定的值找到时，带着标签 search 的 break 语句结束了 search 所指示的外层 for 循环。

例 3-12　在二维数组中搜索指定的值。

```
public class BreakWithLabelDemo {
    public static void main(String[] args) {
        int[][] arrayOfInts = { { 32, 87, 3, 589 },
                                { 12, 1076, 2000, 8 },
                                { 622, 127, 77, 955 }
                              };
        int searchfor = 12;
        int i = 0;
        int j = 0;
        boolean foundIt = false;

  search:
        for ( ; i < arrayOfInts.length; i++ ) {
            for (j = 0; j < arrayOfInts[i].length; j++ ) {
                if (arrayOfInts[i][j] == searchfor) {
                    foundIt = true;
                    break search;
                }
            }
        }
        if (foundIt) {
            System.out.println("Found " + searchfor + " at " + i + ", " + j);
        } else {
            System.out.println(searchfor + "not in the array");
        }
```

```
    }
}
```

例 3-12 的运行结果如下:

```
Found 12 at 1, 0
```

2. continue 语句

在 for、while 和 do while 循环中,continue 语句跳过当前循环的其余语句,执行下一次循环,当然执行下次循环前要判定循环条件是否满足。不带标签的 continue 语句跳过最内层的循环,并开始执行最内层循环的下一次循环。例 3-13 的程序对一个字符串缓冲区中的字符逐一检查。如果当前字符不是字母 p,则 continue 语句跳过循环的其余语句并开始处理下一个字符;如果是字母 p,则程序将计数器加 1 并把 p 转换为大写。

例 3-13 在字符串缓冲区中寻找指定字符并进行处理。

```
public class ContinueDemo {
    public static void main(String[] args) {
        StringBuffer searchMe = new StringBuffer(
                  "peter piper picked a peck of pickled peppers");
        int max = searchMe.length();
        int numPs = 0;
        for (int i = 0; i < max; i++) {
            if (searchMe.charAt(i) != 'p')
                continue;
            //对字母 p 进行处理
            numPs++;
            searchMe.setCharAt(i, 'P');
        }
        System.out.println("Found " + numPs + " p's in the string.");
        System.out.println(searchMe);
    }
}
```

例 3-13 的运行结果如下:

```
Found 9 p's in the string.
Peter PiPer Picked a Peck of Pickled PePPers
```

带标签的 continue 语句结束由标签所指外层循环的当前循环,开始执行该循环的下次循环。在例 3-14 中,使用了两层嵌套的循环实现在一个字符串中搜索另一个字符串。两层嵌套循环中,外层循环处理被搜索的字符串,而内层循环处理要寻找的字符串。程序中使用了带标签的 continue 语句来结束外层循环的当前循环。

例 3-14 在一个字符串中搜索另一个字符串。

```
public class ContinueWithLabelDemo {
    public static void main(String[] args) {
        String searchMe = "Look for a substring in me";
        String substring = "sub";
        boolean foundIt = false;
```

```
        int max = searchMe.length() - substring.length();

    test:
        for (int i = 0; i <= max; i++) {
            int n = substring.length();
            int j = i;
            int k = 0;
            while (n-- != 0) {
                if (searchMe.charAt(j++) != substring.charAt(k++)) {
                    continue test;
                }
            }
            foundIt = true;
            break test;
        }
        System.out.println(foundIt ? "Found it" : "Didn't find it");
    }
}
```

例 3-14 的运行结果如下：

```
Found it
```

3.4　数　　组

数组中的元素都是同一种类型。数组的长度在创建时确定，并且在创建后不变。如果需要建立存储不同类型数据的集合，或者要求这种数据存储结构的长度可以动态变化，可以使用 java.util 包中的各种集合(collection)类，如 Vector 等。

3.4.1　数组的声明

可以声明任何类型的数组，包括基本类型和类类型的数组。数组声明和其他类型的变量声明一样，包括两个部分：数组的类型和数组的名字。数组声明语句有如下两种格式。

(1) 将表示数组的[]跟随在数组变量之后。

这种格式是 C、C++和 Java 的标准格式。例如：

```
char   s[];
Point  p[];  //Point 是一个类
```

(2) 将表示数组的“[]”跟随在数组类型之后。

这种格式是 Java 中特有的。例如：

```
char[]  s;
Point[]  p;
```

这种格式中，在数组标志“[]”后出现的所有变量都将是数组变量。例如：

```
char[]  s,m,n;          //声明了 3 个字符数组变量 s,m 和 n
```

注意:在数组的声明中不指定数组的长度。

在 Java 中数组是作为类来处理的。在 Java 中类类型变量的声明并不创建该类的对象。所以数组的声明并不创建实例数组的数据空间,而是给该数组变量分配了一个可用来引用该数组的引用空间。

3.4.2 数组的创建与初始化

1. 用 new 创建数组

数组元素所需的内存空间是通过 new 运算符或通过数组初始化分配的。通过 new 运算符创建数组的格式如下:

```
new elementType[arraySize]
```

例如,对于前面已经声明的字符串数组 s,可以利用下列语句创建它的数据空间,从而创建一个能够容纳 20 个字符的数组:

```
s = new char[20];
```

数组在创建后,其元素是被系统自动进行初始化的。对于字符数组,每个元素都被初始化为'\u0000',而对于对象数组,每个元素都被初始化为 null。在通过 new 实例化一个数组后,就可以对数组元素进行赋值并进行其他操作。例如对于上面创建的数组 s 赋值:

```
s[0] = 'a';
s[1] = 'b';
```

对数组元素操作时,要注意不要产生数组越界错误。Java 数组下标的取值范围是 0 到数组长度减 1。数组 s 在创建后内存中的布局如图 3-8 所示。

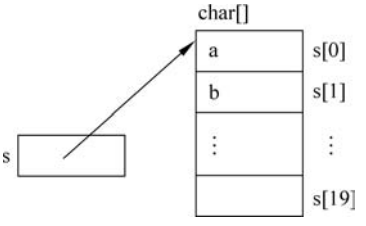

图 3-8 基本类型数组的内存布局示例

2. 数组的长度

Java 中为所有数组设置了一个表示数组元素个数的特性变量 length,它作为数组的一部分存储起来。Java 用该变量在运行时进行数组越界检查,应用程序中也可以访问该变量获取数组的长度,格式如下:

```
arrayname.length
```

例如:s.length。

数组在创建后长度是不可变的。因此不能改变数组的长度,但可以使用该数组变量指向另一个全新的数组,则通过该变量将不能访问到原来数组的值。例如:

```
int a[] = new int[6];
a[0] = 10;
…
a = new int[10];  //不用再次声明 a,此时 a[0]的值为 0,而不是 10
…
```

下面给出一个数组操作的简单例子。

例 3-15 创建整型数组并将其值打印输出。

```
public class ArrayDemo {
    public static void main(String[ ] args) {
        int[ ] anArray;                    //声明一个整型数组
        anArray = new int[10];             //创建数组

        //给数组每个元素赋值并打印输出
        for (int i = 0; i < anArray.length; i++ ) {
            anArray[i] = i;
            System.out.print(anArray[i] + " ");
        }
        System.out.println();
    }
}
```

例 3-15 的运行结果如下：

```
0 1 2 3 4 5 6 7 8 9
```

3. 对象数组

除了创建基本类型数组，还可以创建对象数组。例如，下列语句声明并创建了一个 Point 类型的数组：

```
Point  p[ ];
p = new Point[10];                         //创建包含 10 个 Point 类型对象引用的数组
//创建 10 个 Point 对象
for(int i = 0; i<10; i++ ){
    p[i] = new Point(i,i + 1);
}
```

上述语句执行后，数组 p 在内存中的布局如图 3-9 所示。

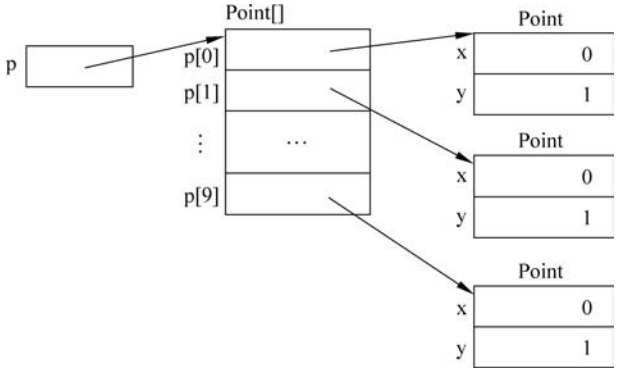

图 3-9　对象数组的内存布局示例

4. 通过初始化创建数组

Java 中可以在数组声明中直接给数组赋初值。这样可以通过一条语句完成数组的声明、创建与初始化 3 项功能，因此是一种数组定义中的简捷方法。例如：

```
//声明并创建一个有 3 个元素的字符串数组 names
String names[ ] = {"Jack","Wang","Lee"};
//声明并创建一个有两个元素的 Date 类型的数组 dates
```

```
Date dates[] = {
          new Date(10,9,2000),
          new Date(10,9,2001)
          };
```

数组 names 的定义等价于如下语句：

```
String names[];
names = new String[3];
names[0] = "Jack";
names[1] = "Wang";
names[2] = "Lee";
```

数组 dates 的定义等价于如下语句：

```
Date dates[];
dates = new Date[2];
dates[0] = new Date(10,9,2000);
dates[1] = new Date(10,9,2001);
```

下面给出一个对象数组操作例子。

例 3-16 创建字符串数组并将其元素转换为小写输出。

```
public class ArrayOfStringsDemo {
    public static void main(String[] args) {
        String[] anArray = { "String One", "String Two", "String Three" };
        for (int i = 0; i < anArray.length; i++) {
            System.out.println(anArray[i].toLowerCase());
        }
    }
}
```

例 3-16 的运行结果如下：

```
string one
string two
string three
```

3.4.3 多维数组

Java 语言的多维数组可以看作是数组的数组，即 n 维数组是 $n-1$ 维数组的数组。下面主要以二维数组为例进行说明。

1. 多维数组的声明

多维数组的声明格式与一维数组相类似，只是要用多对[]表示数组的维数，一般 n 维数组要用 n 对[]。例如：

```
int a[][];
int[][] a;
```

上述两条语句是等价的，声明了一个二维 int 型数组 a。

2. 多维数组的实例化

多维数组的实例化可以采用多种方式，并可以构造规则数组和不规则数组。

(1) 直接为每一维分配内存，创建规则数组。例如：

```
int a[][];
a = new int [4][4];
```

上述第一条语句声明了一个 int 型二维数组 a，第二条语句创建了一个有 4 行 4 列元素的数组。由于 Java 中二维数组是数组的数组，所以创建二维数组 a 实际上是分配了 4 个 int 型数组的引用空间，它们分别指向 4 个能容纳 4 个 int 型数值的空间，如图 3-10 所示。

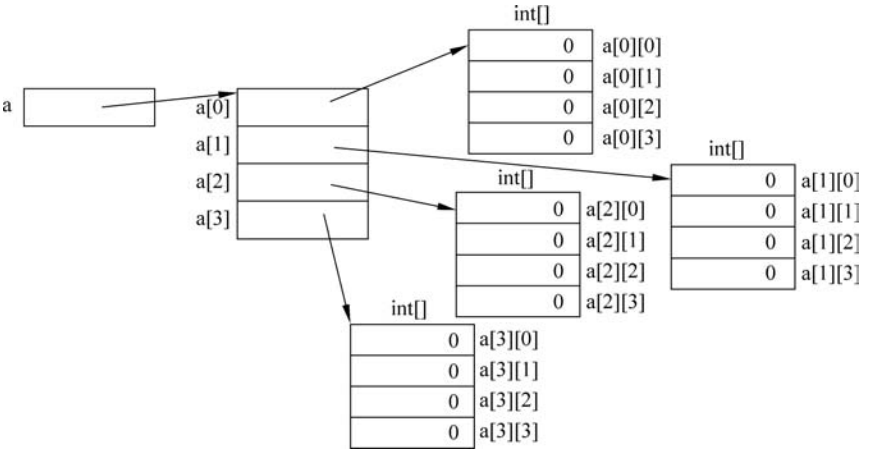

图 3-10　二维 int 型数组的存储结构示例

(2) 从最高维起，分别为每一维分配空间。这种方式可以构造不规则数组。例如：

```
int a[][] = new int[2][ ]; //只有最后维可以不给值,其他都要给
a[0] = new int[10];
a[1] = new int[5];
```

上述语句声明并创建了一个二维不规则数组。

注意：上述第一条语句的 new 表达式中，数组的最高维的长度必须指定，其他维可以不给出。

下面给出两个多维数组操作的例子。

例 3-17　利用二维 int 型数组表达一个矩阵，创建该数组并将其元素打印输出。

```
public class ArrayOfArraysDemo1 {
    public static void main(String[ ] args) {
        int[ ][ ] aMatrix = new int[4][ ];       //aMatrix 指向由 4 个引用组成的数组

        //创建每个 int 数组,并进行赋值
        for (int i = 0; i < aMatrix.length; i++ ) {
            aMatrix[i] = new int[5];
            for (int j = 0; j < aMatrix[i].length; j++ ) {
                aMatrix[i][j] = i + j;
```

```
            }
        }

        //将数组打印输出
        for (int i = 0; i < aMatrix.length; i++ ) {
            for (int j = 0; j < aMatrix[i].length; j++ ) {
                System.out.print(aMatrix[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

例 3-17 的运行结果如下:

```
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
```

例 3-18 通过初始化创建二维数组,并将数组元素打印输出。

```
public class ArrayOfArraysDemo2 {
    public static void main(String[ ] args) {
        String[ ][ ] cartoons = {
        { "Flintstones", "Fred", "Wilma", "Pebbles", "Dino" },
            { "Rubbles", "Barney", "Betty", "Bam Bam" },
            { "Jetsons", "George", "Jane", "Elroy", "Judy", "Rosie", "Astro" },
            { "Scooby Doo Gang", "Scooby Doo", "Shaggy", "Velma", "Fred", "Daphne" }
        };

        for (int i = 0; i < cartoons.length; i++ ) {
            System.out.print("cartoons[" + i + "]" + ", " + cartoons[i].length + " elements: ");
            for (int j = 0; j < cartoons[i].length; j++ ) {
                System.out.print(cartoons[i][j] + (j < cartoons[i].length - 1?", ":" "));
            }
            System.out.println();
        }
    }
}
```

例 3-18 的运行结果如图 3-11 所示。

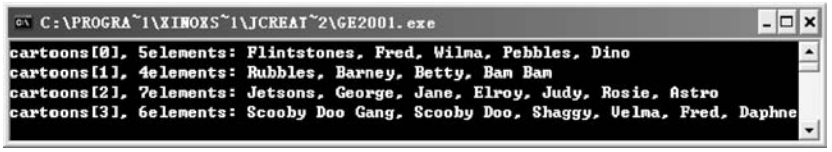

图 3-11 例 3-18 的运行结果

3.4.4　增强的 for 循环

在 JDK 5.0 以后，为了便于对数组和集合(collections)中的元素进行迭代处理，Java 中引入了一种增强的 for 循环形式，其定义如下：

```
for (类型 标识符: 可迭代类型的表达式)  语句;
```

其中，括号中的“类型 标识符”指定了一种类型的标识符，该标识符的类型应与冒号后面的表达式的类型兼容。“可迭代类型的表达式”一般是数组或集合，“:”可理解为 in。例如：

```
int[] numbers = {1,2,3,4,5,6,7,8,9,10};
for ( int element : numbers ) {
     System.out.println(element);
}
```

上述 for 循环可被读为：

```
for each element in numbers do { … }
```

因此，这种形式的 for 循环使程序更加易于理解并更加紧凑，应该尽量采用。

对例 3-16 使用增强的 for 循环进行改写，结果如例 3-19 所示。

例 3-19　增强的 for 循环示例。

```
public class NewArrayOfStringsDemo {
    public static void main(String[] args) {
        String[] anArray = { "String One", "String Two", "String Three" };
        for (String s : anArray) {
            System.out.println(s.toLowerCase());
        }
    }
}
```

例 3-19 的运行结果与例 3-16 相同。

3.4.5　数组的复制

数组变量之间赋值是引用赋值，不能实现数组数据的复制，如下所示。

```
int a[] = new int[6];
int b[];
b = a;
```

上述程序的运行结果如图 3-12 所示。

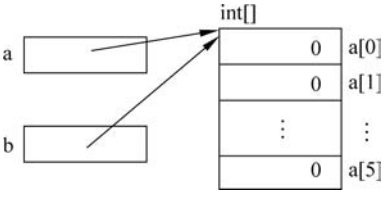

图 3-12　数组变量之间的赋值示例

进行数组数据的复制，要使用 Java 语言在 java.lang.System 类中提供的数组复制方法。该方法的定义如下：

```
public static void arraycopy(Object source,
                             int srcIndex,
                             Object dest,
                             int destIndex,
                             int length)
```

其中：source——源数组；srcIndex——源数组开始复制的位置；dest——目的数组；destIndex——目的数组中开始存放复制数据的位置；length——复制元素的个数。

例 3-20 是一个数组复制的例子。

例 3-20 将一个字符数组的部分数据复制到另一个数组中。

```
public class ArrayCopyDemo {
    public static void main(String[] args) {
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e', 'i', 'n', 'a', 't', 'e', 'd' };
        char[] copyTo = new char[7];

        System.arraycopy(copyFrom, 2, copyTo, 0, 7);
        System.out.println(new String(copyTo));
    }
}
```

例 3-20 的运行结果如图 3-13 所示。

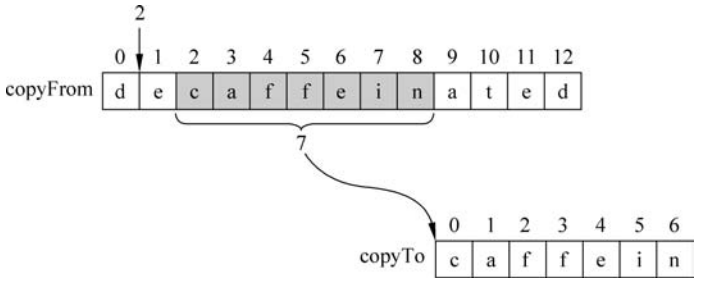

图 3-13 例 3-20 的运行结果

3.5 小 结

本章对 Java 语言的基础部分进行了介绍，包括标识符与数据类型、表达式与语句、程序流控制以及数组。在这些语言基本机制方面 Java 与 C/C++有很多地方是一致的，但也存在很多具体的差异，在学习中应该注意这些差异。关于数组的处理 Java 与 C/C++有很大的不同，在 Java 中数组作为类来对待和处理。基于这个原则可以正确理解数组的声明、创建、复制等操作的语义，并能够正确操作数组。

习 题 3

1. 下列标识符哪些是合法的？

$88，#67，num，applet，Applet，7#T，b++，--b

2. Java 有哪些基本数据类型？什么是复合数据类型？对于这两种类型的变量，系统的处理有什么不同？

3. 设变量 i 和 j 的定义如下，试分别计算下列表达式的值。

```
int i = 1; double d = 1.0
```

(1) 35/4；(2) 46%9+4*4-2；(3) 45+43%5*(23*3%2)；(4) 45+45*50%i--；(5) 45+45*50%(--i)；(6) 1.5*3+(++d)；(7) 1.5*3+d++；(8) i+=3/i+3。

4. 计算下列逻辑运算表达式的值。

(1) (true) && (3>4)；

(2) !(x>0) && (x>0)；

(3) (x>0)||(x<0)；

(4) (x!=0)||(x==0)；

(5) (x>=0)||(x<0)；

(6) (x!=1)==!(x==1)。

5. Java 中有哪些类型的程序流控制语句？

6. switch 语句与 if 语句可以相互转换吗？使用 switch 语句的优点是什么？

7. 试写出下列循环的运行结果。

```
int i = 1;
while(i<10){
    if ((i++)%2 == 0){
        System.out.println(i);
    }
}
```

8. while 循环和 do while 循环有什么区别？

9. 循环跳转语句 break 的作用是什么？试给出下列程序的运行结果。

```
int i = 1000;
while(true){
    if( i<10){
        break;
    }
    i = i-10;
}
System.out.println("The value of i is " + i);
```

10. 循环跳转语句 continue 的作用是什么？试给出下列程序的运行结果。

```
int i = 1000;
while(true){
    if( i<10){
        continue;
    }
    i = i-10;
}
```

11. 编写程序输出 1000 以内的所有奇数。

12. 编写程序输出下列结果。

```
1
1 2
1 2 3
```

```
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
```

13. 下列数组声明哪些是合法的？

(1) int i = new int(30);

(2) double d[] = new double[30];

(3) Integer[] r = new Integer(1..30);

(4) int i[] = (3, 4, 3, 2);

(5) float f[] = {2.3, 4.5, 5.6};

(6) char[] c = new char();

(7) Integer[][] r = new Integer[2];

14. 数组变量是基本类型变量还是引用型变量？数组的内存是在什么时候分配的？

15. 编写程序，创建一个整型 5×5 矩阵，并将其输出显示。