第3章

Java语言基础

CHAPTER 3

本章学习目标：

（1）掌握Java数据类型、各种运算符、表达式的使用。

（2）掌握if语句、switch分支语句，for、while、do-while循环语句等流程控制语句的使用。

（3）熟悉break、continue、return等跳转控制语句的作用。

（4）掌握一维数组、二维数组的声明、建立与使用方法。

（5）熟悉常用的排序、查找算法。

重点：Java数据类型、各种运算符、表达式、控制语句、数组。

难点：二维数组的使用、常用算法的使用。

程序的本质就是对数据进行加工运算，以便得到需要的结果。任何一门语言都有它的基本组成元素，就像中文有字、词、句、章；英文有字母、单词、语句、文章一样。Java语言同样也有它的基本组成元素。不论用什么语言编程，要学好编程，首先要掌握这些编程的基石，掌握命令的语法格式。学习Java语言同样如此，学好Java语言的基本语法是学会Java编程的第一步。由于Java数据类型的设置与C语言很相近，所以在学习本章知识时，如果已有C语言基础，那么学习过程中，一定要与C语言进行比较，要多上机调试程序，通过采用C和Java实现案例来比较语法规则和程序的优劣。通过比较整理出Java基本语法中不同于C语言中的部分，找出这样改进或设置的优劣，重点整理出不同于C语言的部分，使繁多重复的知识得以精炼，方便记忆。如果学习Java前并没有C语言基础，那学习这部分知识就要细学、勤练、多动手、多举例，达到举一反三，融会贯通的效果。


视频讲解

3.1 Java 程序的构成

通过前面内容的学习，已经掌握了配置 Java 编程环境、Java 程序的编译与运行、保存等。Java 语言是一种面向对象的语言，本章将学习 Java 语言的编程基础，如程序的编码规则、Java 数据类型、运算符和表达式、程序流程控制语句等内容。

3.1.1 Java 程序的基本结构

一个 Java 程序的基本结构大体上可以分为包、类、方法(包括 main()主方法)、标识符、关键字、语句和注释等，如例 3-1 所示。

【例 3-1】 一个简单的 Java 程序。

```
import Java.util. * ;                                  //导入类
public class TestStructure {                           //创建类"TestStructure"(主类)
    public static void main(String[ ] args){           //定义主方法
      Cal cal = new Cal(8,4) ;
      System.out.println(" 8 + 4 = " + cal.add());
      System.out.println(" 8 - 4 = " + cal.sub());
      System.out.println(" 8 * 4 = " + cal.mul());
      System.out.println(" 8/4 = " + cal.div());
        }
}
class Cal {                                            //类定义
       int x,y ;                                       //定义局部变量
       Cal(int a, int b){                              //构造函数
          x = a ;
          y = b ;
       }
     int add(){                                        //方法 1
        return x + y;
       }
      int sub(){                                       //方法 2
            return x - y;
        }
        int mul(){                                     //方法 3
            return x * y;
        }
        int div(){                                     //方法 4
             return x/y;
        }
}
```

程序运行结果如图 3-1 所示。

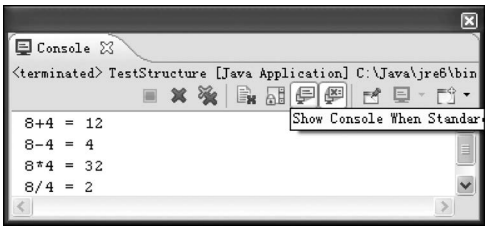

图 3-1 Java 程序结构运行结果

分析以上程序可知：一个程序可包括若干类，各类之间存在并列、继承和包含关系，这些类通常是在一起协同工作的。例如一个类的方法中需要创建其他类的对象，并操作这个对象。编程只能在方法中。Java 程序每条语句必须以分号结尾。代码中的所有标点符号必须是半角的，在英文输入法下输入的符号(如逗号、分号、双引号等)，否则程序会出错。

由此可得，一个完整的 Java 源程序应该包括下列部分：

```
package 语句;      //该部分至多只有一句,必须放在源程序的第一句
import 语句;     /* 该部分可以有若干 import 语句或者没有,必须放在所有的类定义之前 */
public classDefinition;      /* 公共类定义部分,至多只有一个公共类的定义,Java 语言规定该
                             Java 源程序的文件名必须与该公共类名完全一致 */
classDefinition;      //类定义部分,可以有 0 个或者多个类定义
interfaceDefinition;      /* 接口定义部分,可以有 0 个或者多个接口定义,例如一个 Java 源程序
可以是如下结构,该源程序命名为 HelloWorldApp.java: */
package Javawork.helloworld;      /* 把编译生成的所有.class 文件放到包 Javawork.helloworld
中 */
import Java.awt.*;      //告诉编译器本程序中用到系统的 AWT 包
import Javawork.newcentury;      /* 告诉编译器本程序中用到用户自定义的包 Javawork.
                                 newcentury */
public class HelloWorldApp
{......}      /* 公共类 HelloWorldApp 的定义,名字与文件名相同 */
class TheFirstClass{......}      //第一个普通类 TheFirstClass 的定义
class TheSecondClass{......}      /* 第二个普通类 TheSecondClass 的定义 */
......      //其他普通类的定义
interface TheFirstInterface{......}      /* 第一个接口 TheFirstInterface 的定义 */
......      //其他接口定义
```

以上 Java 结构也可简单描述如下：

```
package
import ---
class   类名 1 {
    属性定义
    方法名 1() {
        -----
    }
    方法名 2() {
        -----
    }
}
class  类名 2 {
      -------
}
```

Java 语言的源程序代码由一个或多个编译单元组成，每个编译单元可包含三个要素。

(1) 包声明(package statement，可选)。

package 语句：由于 Java 编译器为每个类生成一个字节码文件，且文件名与类名相同，因此同名的类有可能发生冲突。为了解决这一问题，Java 提供包来管理类名空间，包实际提供了一种命名机制和可见性限制机制。而在 Java 的系统类库中，把功能相似的类放到一个包(package)中，即一个包是一组相关类的集合。类库由若干包组成(class library-package)。

而用户自己编写的类(指.class 文件)也应该按照功能放在由程序员自己命名的相应的

包中，例如上例中的 Javawork. helloworld 就是一个包。包在实际的实现过程中是与文件系统相对应的。

以 package 开始的包声明语句，必须放在文件开始，作用是把当前文件(类)放入所指向的包中。为可选语句，若有，只能有一个 package 语句且只能是源程序文件的第一个语句；若没有，此文件将放到默认的当前目录下。

(2) 类引入声明语句(import statements)。

import 语句：如果在源程序中用到了除 Java. lang 这个包以外的类，无论是系统的类还是自己定义的包中的类，都必须用 import 语句标识，以通知编译器在编译时找到相应的类文件。例如上例中 import Java. awt. *；导入的是系统的包，而 import Javawork. newcentury；导入的是用户自定义的包。如果要从一个包中引入多个类则在包名后加上". *"表示，如 import Java. io. *；以 import 语句开头的类引入声明语句，引入声明语句必须放在所有类定义之前，用来引入标准类或已有类。

(3) 类的声明(class declarations)和接口声明(interface declarations)。

由 public 开始的类定义如果源程序文件中有主方法 main()，它应放在 public 类中。

public classDefinition，0～1 句，文件名必须与类的类名完全相同。

classDefinition，0～n 句，类定义的个数不受限制。

interfaceDefinition，0～n 句，接口定义的个数不受限制。

说明：

① 以上三个要素必须严格按照以上述顺序出现。也就是说任何引入语句出现在所有类定义之前；如果使用包声明，则包声明必须出现在类和引入语句之前。

② Java 语言源程序是由类定义组成的，每个 Java 的编译单元可包含多个类或接口，但是每个编译单元最多只能有一个类或者接口是公共的，即能被 public 修饰的主类。在 Java Application 中，这个主类是指含有 main 方法的类；在 Java Applet 中，这个主类是被定义为系统 Applet 子类的类。主类是 Java 程序执行的入口点。

③ Java 程序中定义类的关键字是 class，每个类的定义都由类头定义和类体定义两部分组成。类头定义的格式前面已经说明，类体定义部分主要是定义静态属性变量和动态属性方法两种类的成员。

④ 语句是构成 Java 程序的基本单位之一。每一条语句都以分号结尾，语句的构成应符合 Java 程序的语法规定。

⑤ 比语句更小的语言单位是表达式、变量、常量和关键字等。它们构成了 Java 程序的语句。

3.1.2 Java 程序的编码规则

软件开发是一个集体协作的过程，程序员之间的代码经常要进行交换阅读，因此，Java 源程序有一些约定成俗的编码规则，主要目的是提高 Java 程序的可读性和正确性。

1. 编码规则

在学习开发的过程中要养成良好的编码规范，因为规整的代码格式是提高程序可读性和维护性的一种手段，便于代码的重复使用。Java 语言的编码规则如下。

(1) 每条语句要单独占一行。

(2) 每条命令都要以分号结束,分别必须是英文状态的分号。

(3) 声明变量时要分行声明,即使是相同数据类型,也最好分行声明,以便添加注释。

(4) Java 语句中多个空格看成一个。

(5) 不要使用技术性很高、难懂、易混淆判断的语句。

(6) 对于关键的方法要多加注释,以增加可读性。

2. 标识符与关键字

标识符、关键字与后面的变量、常量,是构成 Java 程序的基本元素,是 Java 语言的编程基础。

1) 标识符

标识符可以简单地理解为一个名字,用来标识类名、变量名、方法名、数组名、文件名的有效字符序列。标识符由用户自由指定,但一般遵循见名知义的原则,Java 对于标识符的定义有如下规定:

(1) 标识符由字母、数字、下画线“_”和美元符号“$”组成。

(2) 标识符第一个字符可以是字母、下画线“_”和美元符号“$”,但不能是数字。

(3) Java 中的标识符区分大小写,无长度限制。如 class 和 Class,system 和 System,money 和 Money 分别代表不同的标识符。

(4) 标识符不能是 Java 的关键字和保留字。

标识符的命名规范直接影响着代码的正确性、可读性和可维护性。在 Java 中,对标识符通常有以下约定:

(1) 包名:包名是全小写的名词,中间用点分隔开,例如 Java.awt.event。

(2) 类名、接口名:首字母大写,通常由多个单词合成一个类名或接口名,要求每个单词的首字母也要大写,例如 class HelloWorldApp 或 interface Collection。

(3) 方法名:往往由多个单词合成,第一个单词通常为动词,首字母小写,中间的每个单词的首字母都要大写,例如 balanceAccount 或 isButtonPressed。

(4) 变量名:全小写,一般为名词,例如 length。

(5) 常量名:基本数据类型的常量名为全大写,如果是由多个单词构成,可以用下画线隔开,例如 int YEAR 或 int WEEK_OF_MONTH;如果是对象类型的常量,则是大小写混合,由大写字母把单词隔开。

表 3-1 列出了合法与不合法标识符的对照表。

表 3-1　合法与不合法标识符的对照表

合法标识符	不合法标识符	合法标识符	不合法标识符
MyJavaApp	1MyJavaApp	$ theFirstName	Java Applet
YourSalary34	Three&Cups	nVariable	Vari＃able
_isTrue	－istrue	Boy_number	switch

说明:程序开发中,虽然可以使用汉字、日文等作为标识符,但为了避免出现错误,最好不要使用。

2）关键字

关键字是 Java 保留某些词汇作特殊用途的字符序列，故也称保留字。变量标识符不能与关键字同名，否则编译会出错。Java 中常用的关键字如表 3-2 所示。

表 3-2 Java 中常用的关键字

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

在 Java 中，常量 true、false、null 都是小写的，不像 C++ 中都是大写的。Java 中没有 sizeof 符，所有基本数据类型的长度都是固定的，与平台无关。体现了 Java 的跨平台性。Java 中也没有 const 和 goto 关键字，但也不可以使用。这两个词可能会在以后的升级版本中被使用。

3. 代码注释

软件编程规范中提到“可读性第一、效率第二”。在开发 Java 程序的过程中，经常需要在适当的地方加上注释语句，以便其他人阅读你的程序，通过在程序代码中添加注释可提高程序的可读性，注释中包含了程序的信息，可以帮助程序员更好地阅读和理解程序。在 Java 源程序文件的任意位置都可添加注释语句。注释中的文字 Java 编译器并不进行编译，所有代码中的注释文字并不对程序产生任何影响。一个好的程序应该在其需要的地方适当地加上一些注释，程序注释一般占程序总代码的 20%～50%。

注释语句有以下三种格式。

1）单行注释

//注释内容用于注释一行语句，单行注释一般用于描述代码的实现细节，如代码行的功能、变量的用途、方法存在的缺陷等。

2）多行注释（块注释）

/ * 注释内容 * /用于注释一行或多行语句，注释中的内容可以换行。

关于行注释和块注释，例如：

```
int x;
void nn(){
    int y;
    System.out.println(x);                    //x 的值为 0
    System.out.println(y);                    //y 未赋值，编译无法通过
```

在多行注释中可以嵌套单行注释，例如：

```
/ *
        程序名称:HelloJava                  // 开发时间:2013 - 7 - 28
 * /
```

但多行注释中不可以嵌套多行注释，非法代码如下：

```
/*
                程序名称:HelloJava
/* 开发时间:2013-7-28
开发者:李杰
*/
*/
```

3）文档注释

/** 注释内容 **/当文档注释出现在任何声明（如类的声明、类的成员变量的声明、类的成员方法声明等）之前时，会被 Javadoc 文档工具读取作为 Javadoc 文档内容，生成 API 文档，实现文档与程序同步实现的功能。文档注释对于初学者不是很重要，了解即可。

3.2　Java 数据类型、常量和变量

Java 数据类型的设置与 C 语言很相近。其不同之处在于：首先，Java 的各种数据类型占用固定的内存长度，与具体的软硬件平台环境无关；其次，Java 的每种数据类型都对应一个默认的数值，使得这种数据类型的变量的取值总是确定的。这两点分别体现了 Java 的跨平台特性和安全稳定性。

3.2.1　数据类型

程序中任一数据都属于某一特定类型，数据类型决定了数据的表示方式、取值范围及可进行的操作。同一类型的数据有相同的表示形式、取值范围和可进行的操作。例如，Java 中 int 类型的数据取值范围为 $-2^{31}\sim2^{31}-1$ 的整数，整数可以进行加、减、乘、整除和赋值等操作。与其他高级语言类似，Java 的数据类型可分为基本数据类型和引用数据类型两大类。基本数据类型共 8 种，用以实现基本的数据运算，其变量中保存数据值；而引用数据类型共 3 种，是用户根据自己的需要定义并实现的类型，是由基本数据类型组合而成的，其变量中保存的是地址。Java 数据类型如表 3-3 所示。

表 3-3　Java 数据类型

分　类	数 据 类 型		占用字节数	取 值 范 围
基本数据类型	整数类型	字节(byte)	1	−128～127
		短整型(short)	2	−372 768～3276
		整型(int)	4	$-2^{31}\sim2^{31}-1$
		长整型(long)	8	$-2^{63}\sim2^{63}-1$
	浮点类型	float	4	±3.4E−38～±3.4E38
		double	8	±1.7E−308～±1.7E308
	逻辑类型(Boolean)		1	true,false
	字符类型(char)		2	'\u0000'～'\uffff'
引用数据类型	类(class)			
	接口(interface)			
	数组(array)			

说明：字符串在 Java 中不是一种基本数据类型，而是被当作对象来处理，String 和 StringBuffer 对象都可以用来表示一个字符串。

1. 整数类型

整数类型又分为字节(byte)、短整型(short)、整型(int)、长整型(long)四种数据类型。其默认初始值为 0，但在程序中局部变量无默认初始值，必须赋值。一个整数的默认类型为 int，要表示一个 long 型整数，需加后缀 l 或 L，如 789L。

以上 4 种整数类型在 Java 中有 3 种表示形式：

十进制数。用 0～9 的数字表示，其首位不能为 0，如 400、37 等。

八进制数。用 0～7 的数字表示，以 0 为前缀，如 049、037 等。

十六进制数。用 0～9 的数字或 a～f、A～F 的数字表示，以 0x 或 0X 为前缀，如 0x49、0XA1 等。

2. 浮点类型

浮点类型指带小数点的数。按照表示范围和精度，分为单精度浮点(float)和双精度浮点(double)两种类型，它们分别以 32 位和 64 位形式存放，分别用后缀 f/F 和 d/D 来标志数据的浮点类型，如 67.45f、－345.129D 等。Java 中的默认浮点类型为 double。

浮点型有两种表示形式。

标准记数法。由整数、小数点和小数部分组成，如－0.234、12.346 等。

科学记数法，或称指数形式。由尾数、E 或 e 及阶码组成，如 3.65E－5 表示 3.65×10^{-5}。

3. 字符类型

字符类型(char)表示 Unicode 字符，即一个字符采用 16 位无符号整数来表示，用单引号括起来单个字符或转义字符表示，如 char x＝'a'；或用'\u0044'表示'C'；而用双引号括起来的是字符串，如"blue"。

4. 逻辑类型

逻辑类型也称布尔类型，只有 true、false 两个值，布尔类型占一字节。

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

引用数据类型在后面章节中进行介绍。

3.2.2 常量

常量就是在程序运行期间值不变的量。常量分为普通常量与标识符常量。Java 中的普通常量有整型常量、浮点常量、字符常量、字符串常量和布尔常量。

1. 整型常量

在前面的整型数据类型中提到，整型常量包括 byte、short、int、long 等 4 种数据类型。如 38 为整型常量。

2. 浮点常量

浮点常量分单精度和双精度浮点常量，它们分别以“f 或 F”“d 或 D”作为后缀来表示，双精度后面的“d 或 D”可以省略，如 6.46 f、2.366E－5D、3.1415 等。

3. 字符常量

字符常量用单引号括起来的单个字符表示，如‘&’和‘F’等，而“&”和“F”表示单个字符的一个字符串，二者是有区别的。字符常量也可以是转义字符，如表 3-4 所示。

表 3-4　转义字符

转 义 字 符	含　　义	Unicode 值
‘\’’	单引号字符	‘\u0027’
‘\’’’	双引号字符	‘\u0022’
‘\\’	反斜杠	‘\005c’
‘\r’	回车	‘\u000d’
‘\n’	换行	‘\u000a’
‘\f’	走纸换页	‘\u003d’
‘\t’	横向跳格，水平制表符 Tab	‘\u0009’
‘\b’	退格	‘\u0008’

4. 字符串常量

字符串常量用单引号括起来的若干字符，如“Hello?”。

5. 布尔常量

布尔常量只有 true 和 false 两个值，占一字节。

Java 中的标识符常量也称符号常量，由 final 关键字来定义，定义格式为

[修饰符]final type name＝value;

```
如 public static final float FPI = 3.1415926F;
    final char SEX = 'M';
```

说明：

符号常量必须先声明(定义)，后使用。

修饰符是表示该常量使用范围的权限修饰符，如 public、private、protected 或缺省。“[]”表示其中的内容可以省略。

符号常量全部大写，命名时要“见名知义”。

声明符号常量的优点如下：

增加了程序的可读性，由常量名可知常量的含义。

增强了程序的可维护性，只要在常量的声明处修改常量的值，就自动修改了程序中所有地方所使用的常量值，起到了“一改全改”的作用。

【例 3-2】 定义一个 Circle 类,其中 PI 为常量。

```
//Circle.java
public class Circle {
    public static final double PI = 3.14;              //定义常量 π 的值
    public double radius;                              //定义成员变量 radius

    public Circle(double radius){                      //类的构造方法
        this.radius = radius;
    }

    public double getArea(){                           //类的成员方法
            return radius * radius * PI;
    }
    public static void main(String[] args) {
        Circle aCircle = new Circle(2.0);
        double area = aCircle.getArea();
        System.out.println("圆的面积是:" + area);
    }
}
```

其运行结果如图 3-2 所示。

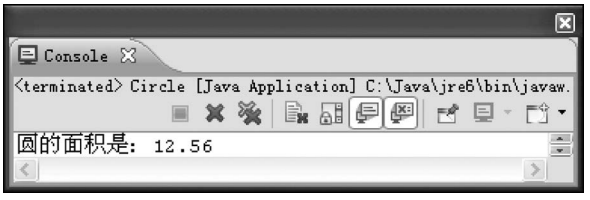

图 3-2 符号常量例

3.2.3 变量

1. 变量的声明

变量是在程序运行期间数值可变的数据,是 Java 中的基本存储单元,常用来记录运算中间结果或保存数据。变量用标识符来命名,Java 中的变量名区分大小写,变量需先声明后使用,定义格式如下:

```
type varName = value;
```

如 double d1,d2=2,3;。

说明:

① 变量需先声明后使用,声明后以分号结束,表示一条完整的 Java 语句。

② 变量名必须以一个字母开头,是一系列字母和数位的组合,空格不能在变量名中使用;变量名习惯用小写字母,如果变量名由多个单词构成,则首字母小写,其后单词的首字母大写,其余字母小写,取名时“见名知义”。变量名也最好不要起成汉字。

③ 变量名不能使用 Java 中的关键字,且区分大小写。

④ 与 C 语言相似,声明多个变量时,变量间用逗号分隔。

【例 3-3】　一个关于变量赋值的例子。

```
public class TestVariable {
     public static void main(String[] args) {
          boolean b = true;
          int i = 8,j = -99;
          long l = 1234567891;
          char chc = '中';
          double d = -1.04E-5;
          System.out.println("逻辑变量 b = " + b);
          System.out.println("整型变量 j = " + j);
          System.out.println("字符型变量 chc = " + chc);
          System.out.println("长整型变量 l = " + l);
          System.out.println("双精度变量 d = " + d);
          }
}
```

运行结果如图 3-3 所示。

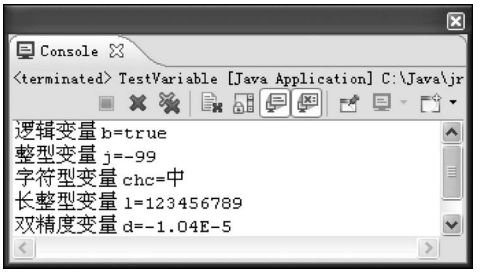

图 3-3　变量赋值

2. 变量的有效范围

变量的有效范围是指程序代码能够访问该变量的区域，若超出变量所在区域访问变量则编译时会出现错误。在 Java 程序中，变量分为成员变量和局部变量。

1）成员变量

在类体中定义的变量被称为成员变量，成员变量在整个类中都有有效。类的成员变量又可以分为静态变量和实例变量。

声明静态变量和实例变量的示例如下。

```
Class var{
int a = 34;                    //声明实例变量
static int b = 56;             //声明静态变量
…
```

其中 b 变量前加上了 static 关键字，被称为静态变量。静态变量的有效范围还可以跨类，甚至可达到整个应用程序之内。使用静态变量时，除了能在定义它的类内存取，还能直接以“类名·静态变量”的方式在其他类内使用。

2）局部变量

在类的方法体中定义的变量，称为局部变量。局部变量只在当前代码块(定义它的花括号内)中有效。在其他类体中不能调用该变量。

【例 3-4】　成员变量与局部变量示例。

```
public class Val {
     static int numbers = 3;                          //定义成员变量 numbers
     public static void main(String[] args) {         //主方法
          int numbers = 4;                            //定义局部变量 numbers
          System.out.println("局部 numbers 的值为:" + numbers); /*输出局部变量*/
          System.out.println("成员 numbers 的值为:" + Val.numbers); /*输出静态变量*/
    }
}
```

运行结果如图 3-4 所示。

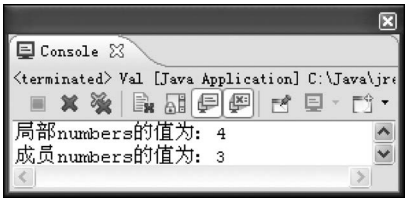

图 3-4　成员变量与局部变量示例

3.3　Java 运算符、表达式、控制结构

对各种类型的数据进行加工的过程称为运算。表示各种不同运算的符号称为运算符。由操作数和运算符按一定的语法形式组成的符号序列称为表达式。本节介绍 Java 运算符、表达式以及 Java 程序的控制结构。

3.3.1　运算符

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、条件运算符、其他运算符。

1. 算术运算符

算术运算符是针对数值型数据进行的运算，根据操作数的不同，可分为双目运算符和单目运算符。

1）双目运算符

双目运算符有 5 个，即＋(加)、－(减)、＊(乘)、/(除)和％(求余)。其中，“％”仅用于整数类型数据，可求得两个整数相除的余数。当“/ ”用于两个浮点数类型操作数时，得到的是其商；当“/ ”用于两个整数类型操作数时，得到的是其商的整数部分，称为整除。例如：

```
10 % 8                  //结果是 2
-13 % 6                 //结果是 -1,结果和符号与被除数符号相同
46/4                    //结果是 11
```

2）单目运算符

单目运算符有 3 个，即＋＋(自增)、－－(自减)、－(求相反数)。前两个运算符与 C 语言中运算类似，分别起变量加 1 和减 1 的作用，需认真区分“a＋＋”和“＋＋a”及“a－－”和“－－a”的不同，运算符在变量前是“先运算，后引用”，运算符在变量后是“先引用，后运算”。

例如：

```
int i = 6;
j = i++;                //结果,i = 7,j = 6
k = --i;                //结果,i = 6,k = 6
```

2. 关系运算符

关系运算是比较两个数据大小的运算，关系运算符有 6 个，即＞(大于)、＜(小于)、＞＝

(大于或等于)、<=(小于或等于)、==(等于)、!=(不等于)。关系运算的结果是布尔值,即 true 和 false。在表达式中需注意区分等于号与赋值号。

例如:

```
int a = 3,b = 4;
boolean c = (a == b);                    //结果 b 为 false
```

3. 逻辑运算符

逻辑运算符用于布尔类型的数据运算,运算结果仍然是布尔型。逻辑运算符有 6 个,如表 3-5 所示。

表 3-5　逻辑运算符

运算符	运　　算	用　　例	运 算 规 则
&	逻辑与(非简洁与)	x & y	x,y 都为真时,结果才为真
|	逻辑或(非简洁或)	x | y	x,y 都为假时,结果才为假
!	非	! x	x 真时结果为假,x 为假时结果为真
^	异或	x ^ y	x, y 同真假时结果为假
&&	条件与(简洁与)	x && y	x,y 都为真时,结果才为真;只要 x 为假,不再计算 y,结果为假
||	条件或(简洁或)	x || y	x,y 都为假时,结果才为假;只要 x 为真,不再计算 y,结果为真

以上运算符,只有!(逻辑非)是单目运算符。

比较 & 与 &&、|与||的不同,如下:

条件与 (&&)、条件或(||)的功能与逻辑与 (&)、逻辑或(|)的功能类似,但 && 和|| 有短路计算功能。条件运算可能只计算左边表达式的值而不计算右边表达式的值。例如,对于 && 运算符,只要左边操作数的值为 false,就不计算右边表达式的值,整个表达式的值为 false;对于||运算符,只要左边操作数的值为 true,就不计算右边表达式的值,整个表达式的值为 true。

【例 3-5】 逻辑运算符的使用。

```
public class LogicOperator {
   public static void main(String[] args) {
        int x = 3,y = 5;
        int a = 3,b = 5;
        boolean z = x > y && x++ == y--;          //条件与运算,短路计算
        boolean c = a > b & a++ == b--;           //逻辑与运算
        System.out.print("x = " + x);
        System.out.print(" y = " + y);
        System.out.println(" z = " + z);
        System.out.print("a = " + a);
        System.out.print(" b = " + b);
        System.out.println(" c = " + c);
        }
}
```

运行结果如图 3-5 所示。

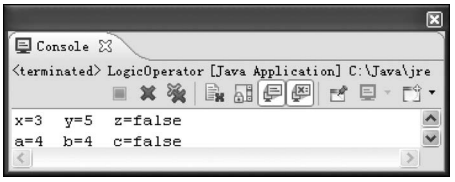

图 3-5 逻辑运算符例子

4. 位运算符

位运算是对整数类型和字符型的操作数按二进制的位进行运算，运算结果仍然是整数值。位运算符有 7 个，即～(位反)、&(位与)、|(位或)、^(位异或)、<<(左移位)、>>(右移位)、>>>(无符号右移位)。位运算符的运算规则如表 3-6 所示。

表 3-6 位运算符的运算规则

运 算 符	运 算	用 例	功 能
～	位反	～a	将 a 按位取反
&	位与	a & b	a、b 逐位进行与操作
|	位或	a | b	a、b 逐位进行或操作
^	异或	a ^ b	a、b 逐位进行异或操作
<<	左移	a << b	a 向左移动 b 位
>>	右移	a >> b	a 向右移动 b 位
>>>	不带符号的右移	a >>> b	a 向右移动 b 位，移动后的空位用 0 填充

【例 3-6】 位运算符例子。

```
public class BitOperator {
    public static void main(String[ ] args) {
        int i = 8;
        int j = 9;
        char c = 'a';
System.out.println("～8 的值是:" + (～i)); /* ～00001000 = 11110111,输出 - 9 */
        System.out.println("8&9 的值是:" + (i&j));
        System.out.println("8^'a'的值是:" + (i^c));
        System.out.println("8 >> 2 的值是:" + (i >> 2)); /* 1000 右移 2 位为 0010,输出 2 */
        System.out.println("8 >>> 2 的值是:" + (i >>> 2));
                                        /* 1000 无符号右移 2 位为 0010,输出 2 */
        System.out.println("8 << 2 的值是:" + (i << 2));
                                        /* 1000 左移 2 位为 100000,输出 32 */
    }
}
```

运行结果如图 3-6 所示。

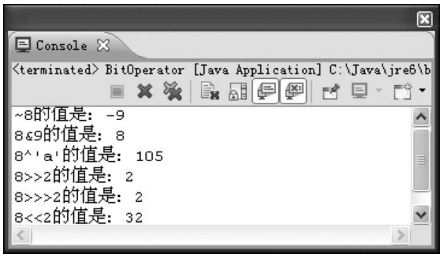

图 3-6 位运算符例子

说明：(1) 位运算得到的二进制值是补码的形式，如果首位是 1，表示这个数是负数，需要按位取反，末位加 1 的规则计算输出值。

(2) 带符号的右移“>>”中，右移后左边留下的空位中填入的是原数的符号位，即正数为 0，负数为 1。不带符号的右移“>>>”中，右移后左边的空位一律填 0。

(3) 移位能实现整数除以 2 或乘以 2 的 n 次方的效果，即 a << n，等价于 $a * 2^n$；反之，a >> n，等价于 $a/2^n$。

5. 赋值运算符

与 C 语言类似，赋值运算符用于将右边表达式的值赋给左边变量。需区分赋值运算符与数学中的等号的不同。例如 i=i+3；是正确的赋值运算，但在数学上是不成立的。

赋值运算符还可以与算术、逻辑、位运算符组合成复合赋值运算符，如+=、*=、%=、&=、|=、<<=、>>>=等，它们是先运算后，再把结果做赋值。

6. 条件运算赋

条件运算赋是一种三目运算符，与 C 语言中的含义完全相同，使用形式如下：

x ? y : z 表示 x 表达式值为真时，运算取 y；若 x 表达式值为假时，运算取 z。

例如：

```
int max,a = 5,b = 10;
max = a > b?a:b;                    //max 取较大者 10
```

7. 其他运算符

(1) 括号运算符：括号运算符()在所有的运算符中优先级最高，用来改变表达式运算的先后顺序，先进行括号内的运算，再进行括号外的运算；在有多层括号的情况下，优先进行最内层括号内的运算，再依次从内向外逐层运算。还可以表示方法或函数的调用。

(2) new 运算符：new 运算符用于建立类的实例或类的数组。

(3) 分量运算符：分量运算符用“.”表示，用于访问对象的成员，或者访问类的静态成员。

(4) 对象运算符：对象运算符(instanceof)用来判断一个对象是否是某个类或子类的实例(对象)，若是则返回 true，否则返回 false。

【例 3-7】 演示各种运算符的使用。

```
class Test{
}
public class OtherOperator {
   public static void main(String[] args) {
        Test t1 = new Test();                                    //new 运算符
        if (t1 instanceof Test){                                 //instanceof 对象运算符
              System.out.println("t1 是 Test 类的实例");        //.是分量运算符
        }
        String s = null;
        s = (t1 == null)?"t1 是空对象":"t1 已创建!";           //条件运算符
```

```
            System.out.println(s);
        }
    }
```

运行结果如图 3-7 所示。

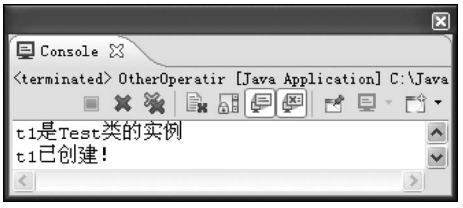

图 3-7　其他类运算符示例

3.3.2　表达式

表达式是用运算符将操作数连接起来的符合语法规则的运算式。操作数可以是常量、变量及方法调用。在表达式中，操作数的数据类型必须与操作符相匹配，变量必须已被赋值。

例如：

```
int a = 2,b = 4,c;
c = (73 - 6 * b) + a;
```

1. 运算符的优先级

运算符的优先级决定了表达式中运算执行的先后顺序，各运算符的大致优先级由高到低为“自增和自减运算→算术运算→比较运算→逻辑运算→赋值运算”。

运算符的优先级与结合性如表 3-7 所示。

表 3-7　运算符的优先级与结合性

优　先　级	运　算　符	描　　述	结　合　性
1	. [] ()	域，数组，括号	从左至右
2	++　--　-　!　~	一元运算符	从右至左
3	*　/　%	乘，除，取余	从左至右
4	+　-	加，减	从左至右
5	<<　>>　>>>	位运算	从左至右
6	<　<=　>　>=	逻辑运算	从左至右
7	==　!=	逻辑运算	从左至右
8	&	按位与	从左至右
9	^	按位异或	从左至右
10	|	按位或	从左至右
11	&&	逻辑与	从左至右
12	||	逻辑或	从左至右
13	?　:	条件运算	从右至左

续表

优　先　级	运　算　符	描　　述	结　合　性
14	=　*=　/=　%= +=　-=　<<= >>=　>>>= &=　^=　|=	赋值运算	从右至左

表达式求值的运算规则如下：按照运算符优先级从高到低的顺序运算，同级运算符按运算符的结合性进行；当遇到圆括号时，先进行括号内的运算，再将括号内的结果与括号外面的运算符和操作数进行运算。在 Java 中，!（非）、+（正）、-（负）及赋值运算符的结合方向是“先右后左”，其他运算符的结合性是“先左后右”。

2. 表达式的数据类型

表达式的数据类型由运算结果的数据类型决定。表达式的数据类型可分为 3 类：算术表达式、布尔表达式和字符串表达式。

例如：

```
int a = 4,b = 80;
boolean d;
d = (15 * a)> b;                    //布尔表达式
```

3. 数据类型转换

当将一种数据类型的值赋给另一种数据类型的变量时，出现了数据类型的转换。整型和字符型数据可以混合运算。在运算过程中，不同类型的数据先转换为同一类型，再进行计算。转换从低级到高级的优先顺序如下：

bye < short < int < long < float < double

转换规则如下：

(1) 将低级别的值赋给高级别的变量时，系统自动完成数据类型的转换（隐式类型转换）。

例如：

```
int x = 30;
float i;
i = x;                    //将 int 型值 30 转换成 float 型值,结果 i 的值是 30.0
```

(2) 将高级别的值赋给低级别的变量时，必须进行强制类型转换（显式类型转换），不推荐使用，因为会使数据精度降低。

例如：

```
int a = (int)56.38;                    //输出 a 的值为 56
long x = (long)678.4F;                 //输出 x 的值为 678
int y = (int)'c';                      //输出 y 的值为 99
```

当把整数赋值给一个 byte、short、int、long 型变量时，不可超出这些变量的取值范围，否则就会发生数据溢出。例如：

```
short a = 634;
byte b = (byte)a;
```

由于 byte 型变量的最大值是 127 634，已超过了其取值范围，因此发生数据溢出，会造成数据丢失。

当双目运算符的两个操作数不同时，系统先将低级别的值转换成高级别的值，再进行计算。有些情况需进行强制类型转换。

【例 3-8】 类型转换例子。

```
public class DivideNumber{
   public static void main(String args[]){
        int i = 16, j = 6, k;
        float f1, f2;
        k = i/j;                    //i,j,k 均为 int 型,i/j,整除得商为 2
        f1 = i/j;                   //先将 2 转换成 float 型值 2.0,再赋给 f1
        f2 = (float)i/j;            /* 先把 i 强制转换为 float 型 16.0,再把 j 也转换为 float 型
        //6.0,进行除法运算,得 float 型值 2.6666667
        system.out.println("k = " + k);
        system.out.println("f1 = " + f1);
        system.out.println("f2 = " + f2);
        }
}
```

运行结果如图 3-8 所示。

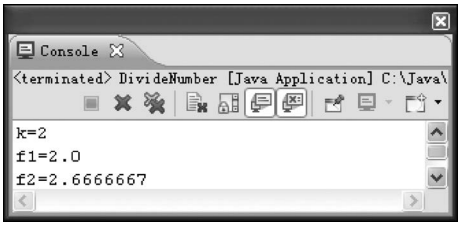

图 3-8 类型转换例子

3.3.3 Java 结构控制语句

大部分高级语言程序都是由若干语句组成的。语句大致可分为简单语句和构造语句两类。简单语句是以分号结束的单一语句，包括最常见的赋值语句、循环体内只有分号的空语句、转移语句和形如“system.out.println(“Hello!”)”的方法调用语句。构造语句包括由花括号“{}”括起来的复合语句、选择语句和循环语句。但是 Java 中没有 goto 语句。

尽管 Java 是完全面向对象的程序设计语言，但在其局部的程序块内，仍然需要借助结构化程序设计的基本流程结构，即顺序结构、分支结构和循环结构，完成相应的逻辑功能。结构化程序设计是遵循公认的面向过程编程的原则，采用“单入口单出口”的控制结构，按照自顶向下、逐步求精和模块化的原则进行程序的分析与设计，使得程序的逻辑结构清晰、层次分明，有效地提高了局部程序段的可读性和可靠性，显著提高了程序设计的质量和效率。三种结构控制语句的框架如图 3-9 所示。

顺序结构语句是三种结构中最简单的结构语句，即程序按照语句的书写次序顺序执行。分支结构语句又称选择结构语句，需根据表达式值来判断应选择执行哪一个流程分支；循

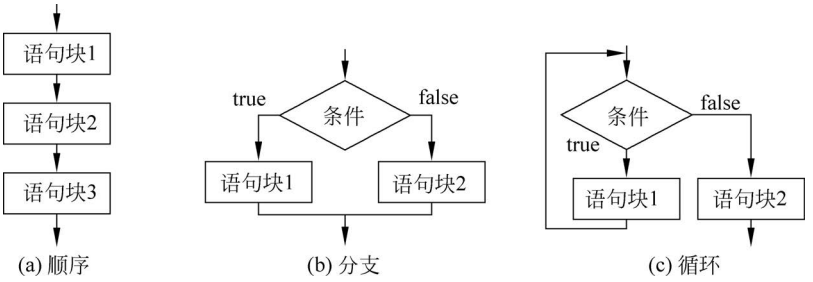

图 3-9　三种结构控制语句的框架

环结构语句则是在满足一定的条件下重复执行一段语句的流程结构。这三种结构构成了程序局部模块的基本框架。

1. 分支结构语句

分支结构语句也称为条件语句，有两种，分别是实现双分支的 if 语句和实现多分支的开关语句 switch 语句。

1）if 语句

if 语句的语法格式如下：

```
if (条件表达式)
     语句块 1;
else
     语句块 2;
```

其中，条件表达式用来选择程序的流程走向，在程序的执行过程中，如果条件表达式的值为真，则执行语句块 1，否则执行 else 分支的语句块 2。在编写程序时，也可以不写 else 分支，此时若条件表达式的值为假，则绕过 if 分支直接执行 if 语句后面的其他语句。语法格式如下：

```
if (条件表达式)
     语句块 1;
其他语句;
```

【例 3-9】 分段函数的分支结构示例。

```
public class FenDuanIf {
     public static void main(String[] args) {
          int x = 48;
          if(x >= 0)
               System.out.println(fun1(x));
          else
               System.out.println(fun2(x));
    }
    static double fun1(int a){
          return Math.sqrt(a) + 1;
    }
    static int fun2(int a){
          return a * a - 3 * a + 1;
    }
}
```

程序运行结果：7.928203230275509。

2）多分支 if 语句

多分支 if 语句也称为嵌套 if 语句，其语法格式如下：

```
if (条件表达式 1)
     语句块 1;
else if (条件表达式 2)
     语句块 2;
     else if (条件表达式 3)
               语句块 3;
```

【例 3-10】 从键盘任意输入三个数，求最大值。

```
public class MaxNum {
     public static void main(String args[])
          {
           int x,y,z, max;
           x = Integer.parseInt(args[0]);
           y = Integer.parseInt(args[1]);
           z = Integer.parseInt(args[2]);
           if(x > y)
                  if(x > z)
                     max = x;
                  else
                     max = z;
           else
               if(x > z)
                    max = y;
               else
                    max = z;
           System.out.println("x = " + x);
           System.out.println("y = " + y);
           System.out.println("z = " + z);
           System.out.println("max = " + max);
          } }
```

程序解析：main()方法中声明了 x、y、z、min 共 4 个 int 类型变量，分别表示通过键盘输入的三个整数和最大者，借助 main()方法中的参数 args[0]、args[1]和 args[2]分别接收按顺序输入的三个参数，通过 Integer. parseInt(args[0])、Integer. parseInt(args[1])和 Integer. parseInt(args[2])分别将三个参数转换成 int 类型值，再分别赋给 x、y、z。然后用嵌套 if 语句来得出三个数中的最大者 max，并输出 max。

说明：在 Eclipse 控制台中运行时，选择 run configurations，在打开的对话框中，再选择 Arguments 选项卡，在其下面的 program arguments 列表框中输入“10 5 12”，按 Enter 键，则运行结果如图 3-10 所示。

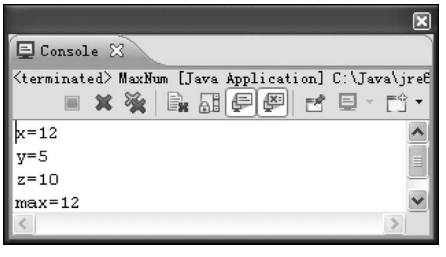

图 3-10　多分支 if 示例

3）switch 语句

switch 语句是一种多分支的开关语句，语法格式如下：

```
switch(表达式){
     case 判断值 1:语句块 1;break;                    //分支 1
     case 判断值 2:语句块 1;break;                    //分支 2
     ...
     case 判断值 n:语句块 n;break;                    //分支 n
     default: 语句块 n+1;                             //分支 n+1
}
```

switch 语句在执行时，首先计算表达式的值，这个表达式的值必须是 byte、char 和 int 类型，不允许是浮点数类型和 long 类型，同时应与各个 case 分支的判断值的类型相一致。如果表达式的值和某个 case 子句后面的常量值相等，就执行该 case 子句中的语句序列，直到遇到 break 语句为止。如果某个 case 子句没有 break 语句，一旦表达式的值与该 case 子句后面的常量值相等，在执行完该 case 子句中的语句序列后，继续执行后继的 case 子句中的语句序列，直到遇到 break 语句为止。如果没有一个常量值与表达式的值相等，则执行 default 子句中的语句序列；如果没有 default 子句，switch 语句不执行任何操作。

【例 3-11】 将百分制成绩转换为优秀、良好、中等、及格和不及格的五级制成绩。标准如下。

优秀：90～100 分；

良好：80～89 分；

中等：70～79 分；

及格：60～69 分；

不及格：60 分以下。

```
public class Level
{
   public static void main(String args[])
    {
      short newGrade, grade;
      grade = Short.parseShort(args[0]);
      switch (grade/10)
       {
           case 10:
           case 9: newGrade = 1; break;
           case 8: newGrade = 2; break;
           case 7: newGrade = 3; break;
           case 6: newGrade = 4; break;
           default: newGrade = 5;
       }
System.out.print(grade);
switch (newGrade)
 {
        case 1: System.out.println(",优秀"); break;
        case 2: System.out.println(",良好"); break;
        case 3: System.out.println(",中等"); break;
        case 4: System.out.println(", 及格"); break;
        case 5: System.out.println(",不及格");
 } } }
```

程序解析：本问题如果用多分支嵌套 if 语句来实现，可读性差，所以考虑用 switch 语句。利用 switch 语句的关键是要构造一个表达式，将各分支条件转换成对应的 char、byte、

short、int 类型的不同值。本例中，构造整数型表达式 grade/10 将分数转换成单个整数值。再利用另一个 switch 语句，将分数级别转换成相应汉字描述的分数级别。

程序中，用 short 类型变量 grade 和 newGrade 分别表示百分制成绩和用 1～5 表示的对应成绩。通过 Short.parseShort(args[0])将键盘输入的参数转换成 short 类型的值，再赋给 grade。需要说明的是，当输入成绩 100 时，grade/10 的值为 10，在执行第一个 switch 语句时，因为 case10：后面没有 break 语句，所以程序继续执行 case9：结果，newGrade 的值为 1，然后跳出第一个 switch 语句。

如果输入 89，则程序运行结果如图 3-11 所示。

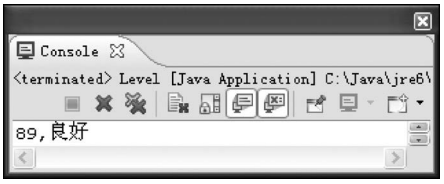

图 3-11　switch 语句转换成绩等级

2. 循环结构语句

循环语句的作用是在一定条件下，反复执行一段程序代码，被反复执行的程序称为循环体。Java 提供的循环语句有 while 语句、do-while 语句和 for 语句。它们的共同特点是，当循环条件满足时，反复执行循环体；否则，退出循环。三种循环语句流程如图 3-12 所示。

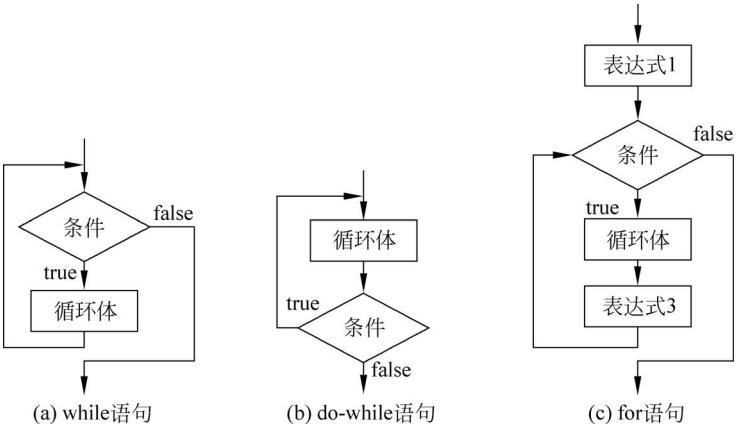

图 3-12　三种循环语句流程

1）while 语句

while 语句也称条件判断语句，它的语法格式如下：

```
while (条件表达式)
循环体
```

其中，条件表达式的值为布尔值，循环体可以是单个语句，也可以是复合语句块。其执行过程如下：当条件表达式的值为真时，执行循环体，并无条件转向条件表达式再做计算与判断；当条件表达式值为假时，跳出循环体执行 while 语句后面的语句。可见，while 语句的特点是：先判断，后执行。

【例 3-12】　求两个数的最小公倍数。

```
public class GongBeiShu {
   public static void main(String[] args) {
         int result;
         int m = Integer.parseInt(args[0]);
         int n = Integer.parseInt(args[1]);
         if (m>0 && n>0) {                    //检查输入的合法性
         result = m<n?n:m ;
         while(result % m!= 0 || result % n != 0) {
               result++;
         }
         System.out.println("最小公倍数为:" + result);
         }
         else                                 //输入错误
         System.out.println("没有输入两个正整数");
  } }
```

当从键盘输入 7　9 时，得到运行结果如下：

```
最小公倍数为: 63
```

说明：while 语句易犯的错误是在 while(条件表达式)之后加分号。例如：

```
while(x == 5);
```

这时程序会认为是一条空语句，而进入死循环，Java 编辑器又不会报错，可能会浪费很多时间去调试，需注意这个问题。

2）do-while 语句

do-while 语句的语法格式如下：

```
do
循环体
while (条件表达式);
```

do-while 语句的特点是：先执行，后判断，所以 do-while 语句的循环体至少要执行一次，这也正是 do-while 语句与 while 语句不同的地方。

【例 3-13】　计算 1～50 的奇数和与偶数和。

```
public static void main(String args[]){
       int i,oddSum,evenSum;
       i = 1;
       oddSum = 0;
       evenSum = 0;
       do {
          if(i % 2 == 0)                    //如果 i 是偶数
            evenSum += i;                   //求偶数和
          else                              //如果 i 是奇数
            oddSum += i;                    //求奇数和
        i++;
      }while(i <= 50);                      //判断 i 的值是否在 1～50
      System.out.println("Odd sum = " + oddSum);
      System.out.println("Even sum = " + evenSum);
      }
}
```

程序运行结果如图 3-13 所示。

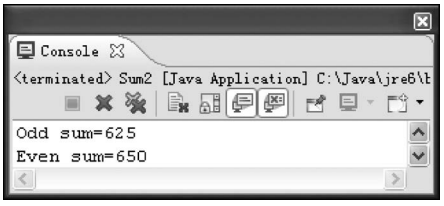

图 3-13　do-while 语句求奇偶数和

3）for 语句

for 语句是 Java 的三种循环语句中功能较强、使用较频繁的一种语句。

for 语句的语法格式如下：

```
for(表达式 1;表达式 2;表达式 3){
        循环体
}
```

表达式 1：循环变量赋初值。

表达式 2：循环条件。

表达式 3：循环变量修正。

for 语句的执行过程：先计算表达式 1，完成初始化工作；再判断表达式 2 的值；若为真，则执行循环体，执行完循环体后再返回表达式 3，计算并修改循环条件，则一轮循环结束。第二轮循环从计算并判断表达式 2 开始，若表达式的值仍为真，则继续循环，否则跳出整个 for 语句执行下面的语句。

for 语句的三个表达式均可为空，但若表达式 2 为空，则表示当前循环是一个无限循环，需要在循环体中增加另外的跳转语句终止循环。

【例 3-14】 从键盘输入一个数，判断是否为素数。

素数是指除 1 及自身外，不能被其他数整除的自然数。对于一个自然数 k，需要使用 2～sqrt(k)的每个整数进行测试，如果不能找到一个整数 i，使 k 能被 i 整除，则 k 是素数；如果能找到某个整数 i，使 k 能被 i 整除，则 k 不是素数。

```
import Java.util.*;
public class IsPrime {
     public static void main(String[] args) {
          Scanner input = new Scanner(System.in);
          int k,i,t;
          System.out.print("请输入一个数字:");
          k = input.nextInt();
          t = (int)Math.sqrt(k);          //Math.sqrt(k)是 double 型,需强制转换为 int 型
          for (i = 2; i <= t; i++)        // i 是除数
               if (k % i == 0)            // 求可以整除被除数的除数
                    break;
           if(i == t + 1)                 //i 是与(t + 1)比较,而不是与 t 比较
                 System.out.println(k + " is a prime.");
           else
                 System.out.println(k + " is not a prime.");
           }
       }
```

当在 Eclipse 控制台通过键盘输入 11 时，运行结果如图 3-14 所示。

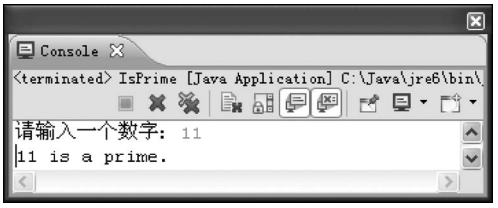

图 3-14　for 循环判素数运行结果

4）多重循环

多重循环又称嵌套循环，即在循环体中又嵌套循环。循环套时要求内循环完全包含在外循环之内，不能出现相互交叉现象。

例如：

```
for( ; ; )                              //外循环开始
{ …
     for( ; ; )                         //内循环 1 开始
     {…}                                //内循环 1 结束
     while(condition)                   //内循环 2 开始
     {…}                                //内循环 2 结束
}
```

【例 3-15】 求 2～50 的所有素数。

```
public class PrimeNumber1
{
   public static void main(String args[])
    {
       final int MAX = 50;
       int i, k;
       boolean yes;
       System.out.println("2～50 的素数为: ");
       for(k = 2; k < MAX; k++)
       {
            yes = true;
            i = 2;
            while (i <= Math.sqrt(k); && yes)
            {
            if (k  % i == 0)
                yes = false;
                  i++;
            }
           if (yes)
               System.out.print(k + " ");
               }
           }
}
```

运行结果如图 3-15 所示。

程序解析：本例用到了 for 语句和 while 语句实现二重循环。内循环用来判断 k 是否为素数。在内循环之前，给布尔变量 yes 赋初值 true。在内循环 while 语句中，如果在 2～Math.sqrt(k)中找到了能整除 k 的整数 I，将 yes 改为 false，结束内循环。若 yes 的值为

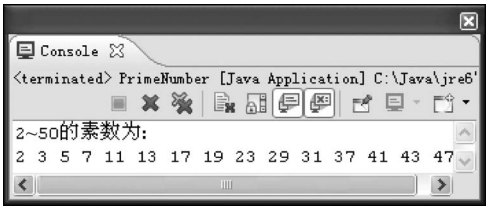

图 3-15　嵌套 for 循环找素数运行结果

true，则 k 为素数。

外循环的作用是对 2～50 中的每一个整数，判断其是否为素数。外循环共执行 48 次，每次对一个整数进行判断。

本例也可以用 for 语句的二重嵌套来实现。代码如下：

```
public class PrimeNumber2
{
     public static void main(String args[])
     {
     int i,j;
     boolean flag;
     System.out.println("2～50 的素数为: ");
     for(i = 2; i <= 50; i++)
          {
          flag = true;
               for(j = 2;j <= Math.sqrt(i);j++)
                    if(i % j == 0)              //如果该数可以整除其余数,则不可能是素数
                         {flag = false;
                             break;
                         }
        if(flag)
              System.out.print(i + " ");
      }
    }
}
```

其运行结果与图 3-15 完全相同。

3. 跳转语句

Java 语言中，跳转语句用于控制流程转移，但不支持无条件跳转的 goto 语句。

1）break 语句

break 语句使程序的流程从一个封闭的语句块内部跳转出来。通常在 switch 和循环语句 while、do-while、for 语句中使用。当程序执行到 break 语句，立即从 switch 语句或循环语句退出。

带标号的 break 语句：

```
break 标号名;
```

表示程序从标号语句块跳出，跳转到该语句后面的语句。

不带标号的 break 语句表示从其所在 switch 分支或最内层循环体中跳转出来，执行分支或循环体后的语句。

2）continue 语句

continue 语句只在 while、do-while、for 循环语句中使用，其作用是终止当前这一轮的循环，跳过本轮循环剩余的语句，直接进入下一轮循环。通常用于某外层循环。

区别如下：

break 语句：退出循环体，执行循环体后面的语句。

continue 语句：提前结束本次循环，忽略本循环体中 continue 语句后面的语句，回到循环的条件测试部分继续执行下一轮循环。

说明：从结构化程序设计的角度，不鼓励使用这两种跳转语句。

3）return 语句

return 语句的格式如下：

```
return 表达式;
```

return 语句的作用是使程序从方法调用中返回，表达式的值就是调用方法的返回值。若没有返回值，则表达式可省略。

3.4　数组

在实际应用中，经常需要处理具有相同性质的一批数据。例如要处理 100 个学生的考试成绩，如果要用基本类型变量（简单变量），将需要 100 个变量，极不方便。为此，在 Java 中，和 C 语言类似，除简单变量外，还引入了数组，即用一个变量表示相同性质的数据。例如，球类的集合——足球、篮球、羽毛球等；电器集合——电视机、洗衣机、电风扇等，就可以分别定义在一个数组中。Java 语言中，数组是一种最简单的复合数据类型，也叫引用数据类型。所谓数组是指名称相同、下标不同的一组变量，它用来存储一组类型相同的数据。数组可以用一个统一的数组名和下标来唯一地确定数组中的元素。按照下标个数不同，数组可分为一维数组和多维数组。

3.4.1　数组的声明和创建

1. 一维数组

一维数组是用一个下标来确定数组中的不同元素的。Java 程序中定义数组的操作与其他语言有一定差异。一般来说，创建一个 Java 的一维数组需要下列两个步骤。

1）一维数组的声明

声明一个数组就是要确定数组名、数组的维数和数组元素的数据类型。

一维数组声明的格式如下：

```
数组元素类型 数组名[];
```

或

```
数组元素类型[]数组名;
```

[]是数组的标志，可出现在数组名前或后。数组元素类型可以是任意的基本数据类型，也可以是引用数据类型。数组名是一个引用变量，其命名方法同简单变量。

例如：

```
int score[];
float []weight;
Employee []employees;
```

上面声明的数组，它们的元素类型分别为 int 型、float 型和 Employee 类型。在 Java 语言中，数组是引用数据类型，也就是说数组是一个对象，数组名就是对象名（或引用名）。数组声明实际上是声明一个引用变量，这一引用变量并没指向任何空间。如果数组元素为引用类型，则该数组称为对象数组，如上面的 employee 就是对象数组。

2）创建数组空间

声明数组后，仅为数组指定了数组名和元素的数据类型，并未指定数组元素的个数，系统还无法为数组分配内存空间。Java 不支持变长数组，所以要想开辟空间，创建数组，还必须指定数组元素的个数，以便实现数组的初始化。创建数组空间有两种方法。

（1）动态创建：动态创建是指用 new 运算符初始化数组，只指定数组元素个数，分配存储空间，并不给元素赋初值。

格式如下：

```
数组名 = new 数组元素类型[数组元素个数];
```

当用 new 运算符创建一个数组时，系统就为数组元素分配了存储空间，这时系统根据指定的长度创建若干存储空间并为数组每个元素指定默认值。对数值型数组元素默认值是 0、字符型元素的默认值是'\u0000'、布尔型元素的默认值是 false。如果数组元素是引用类型，则默认值是 null。

例如：

```
double score[];                    //数组声明
score = new int[5];                //创建数组空间
```

也可把数组声明与创建空间合二为一。例如：

```
int score = new int[10];
Employee []employees = new Employee[3];
```

在内存中，数组的下标从 0 开始，上面两个语句分别分配了 5 个 double 型和 3 个 Employee 类型的空间，并且每个元素使用默认值初始化。上面两个语句执行后效果如图 3-16 所示。数组 score 的每个元素都被初始化为 0.0，而数组 employees 的每个元素被初始化为 null。

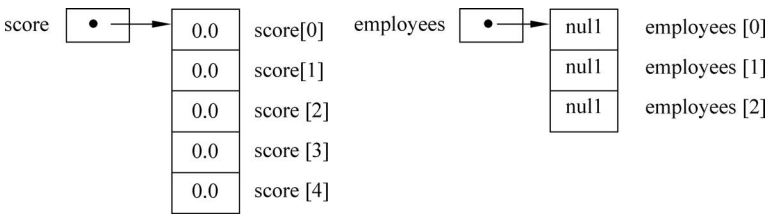

图 3-16　创建数组示例

对于引用类型数组（对象数组），它的每个元素初值为 null，因此，还需要创建数组元素对象。

```
employees[0] = new Employee(1002,"张三",3000.0);
employees[1] = new Employee(1006,"王五",5000.0);
employees[2] = new Employee(1008,"李四",8000.0);
```

上面语句执行后效果如图 3-17 所示。

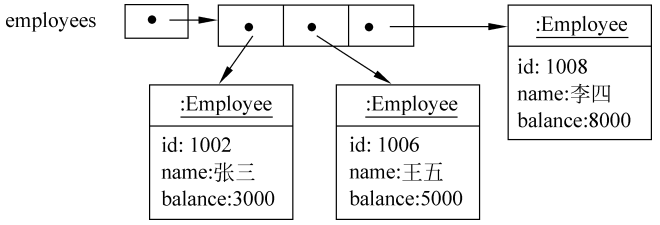

图 3-17　创建数组元素对象的效果

(2) 静态创建：声明数组同时可以使用初始化器对数组元素初始化，它是在一对花括号中给出数组的每个元素值。这种方式适合数组元素较少的情况，这种初始化也称为静态初始化。即在创建空间的同时，给各元素赋初值，这样省略了 new 运算，例如：

```
int score[] = {56,85,100,88,92};
String stringArray[] = {"abc","How","you"};
Employee employees[] = { new Employee(1002,"张三",3000.0), new Employee(1006,"王五",
5000.0), new Employee(1008,"李四",8000.0)}
```

用这种方法创建数组不能指定大小，系统根据元素个数确定数组大小。另外可以在最后一个元素后面加一个逗号，以方便扩充。

说明：无论用哪种方式创建空间，前面的方括号[]中都不能填写任何内容，否则编译出错。

2. 二维数组

Java 语言中，多维数组被看作数组的数组，即包括两个以上下标的数组。其中，二维数组最常用。

1) 二维数组的声明

二维数组声明的格式如下：

```
数组元素类型 数组名[][];
```

或

```
数组元素类型[][]数组名;
```

2) 二维数组空间的创建

(1) 动态创建。有以下两种方法。

① 直接为每一维分配空间，格式如下：

```
数组名 = new 元素类型[行数][列数];
例如 int a[][] = new int[2][3];                    //声明的同时创建空间
等价于 int a[][];
     a = new int[2][3];                            //先声明再创建空间
```

② 从最高维开始，分别为每一维分配空间。

例如：

```
int a[][] = new int[2][];
a[0] = new int[3];
a[1] = new int[5];
```

(2) 静态创建。

```
int intArray[][] = {{1,2},{2,3},{3,4,5}};
```

3.4.2 数组元素的引用

声明了一个数组，并使用 new 运算符为数组元素分配内存空间后，就可以使用数组中的每一个元素。

1. 一维数组元素的引用

数组元素的引用方式如下：

```
数组名[下标]
```

通过数组名和下标访问数组元素，数组下标可以为整型常数或表达式，下标从 0 开始，到数组的长度减 1。每个数组都有一个属性 length 指明它的长度，例如 intArray.length 指明数组 intArray 的长度。

【例 3-16】 简单数组复制数组的引用。

```
public class TestArray1 {
public static void main(String[] args) {
int a[] = {2, -8,5,30};                          //声明并创建整型数组
System.out.println("数组 a 的地址为:" + 啊);
System.out.println("数组 a 的长度为:" + a.length);
print(a);
int b[];
b = a;                                           //将数组 a 的引用赋值给数组 b,变量 b 指向 a 数组
System.out.println("数组 b 的地址为:" + a);
print(b);
b[2] = 100;                                      //数组 a 发生变化了吗?
print(a);
print(b);
}
static void print(int[] array){
for(int i = 0;i < array.length;i++){
System.out.print(array[i] + " ");
}
System.out.println();
}
}
```

运行结果如图 3-18 所示。

程序解析：从运行结果看出，数组 b 和数组 a，实际上是一个地址，当修改 b[2]的值时，数组 a 也发生了变化。

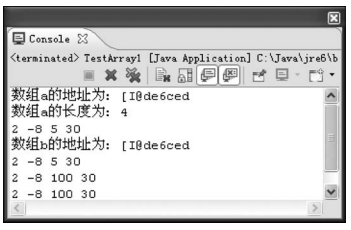

图 3-18 一维数组应用

2. 二维数组元素的引用

对二维数组中的每个元素，引用方式如下：

```
数组名[下标 1][下标 2]
```

例如下面代码给 matrix 数组元素赋值：

```
matrix[0][0] = 80;
matrix[0][1] = 75;
matrix[0][2] = 78;
matrix[1][0] = 67;
matrix[1][1] = 87;
matrix[1][2] = 98;
```

下面代码输出 matrix[1][2]元素值：

```
System.out.println(matrix[1][2]);
```

与访问一维数组一样，访问二维数组元素时，下标也不能超出范围，否则抛出异常。可以用 matrix.length 得到数组 matrix 的大小，结果为 2，用 matrix[0].length 得到 matrix[0]数组的大小，结果为 3。

对二维数组的第一维通常称为行，第二维称为列。要访问二维数组的所有元素，应该使用嵌套的 for 循环。如下面代码输出 matrix 数组中所有元素。

```
for(var i = 0; i < matrix.length; i++){
     for(var j = 0; j < matrix[0].length; j ++){
        System.out.print(matrix[i][j] + " ");
     }
     System.out.println();                    // 换行
}
```

同样，在访问二维数组元素的同时，可以对元素处理，例如计算行的和或者列的和等。

Java 的二维数组是数组的数组，对二维数组声明时可以只指定第一维的大小，第二维的每个元素可以指定不同的大小。例如：

```
var cities = new String[2][];                 // cities 数组有 2 个元素
cities[0] = new String[3];                    // cities[0]数组有 3 个元素
cities[1] = new String[2];                    // cities[1]数组有 2 个元素
```

这种方法适用于低维数组元素个数不同的情况，即每个数组的元素个数可以不同。对于引用类型的数组，除了为数组分配空间外，还要为每个数组元素的对象分配空间。

```
cities[0][0] = new String("北京");
cities[0][1] = new String("上海");
cities[0][2] = new String("广州");
cities[1][0] = new String("伦敦");
cities[1][1] = new String("纽约");
```

cities 数组元素空间的分配情况如图 3-19 所示，图中共有 8 个对象。

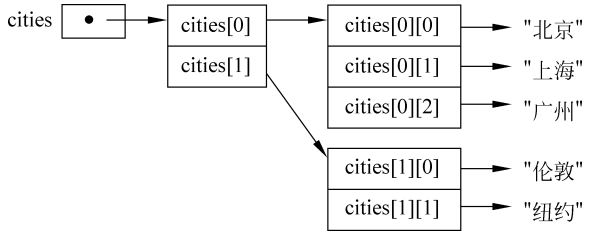

图 3-19　创建不规则二维数组

【例 3-17】 假设要打印输出前 10 行杨辉三角形，可以用不规则数组存储。

```
public class Triangle
{
     public static void main(String[ ] args)
     {
          int level = 10;
          int triangle[][] = new int[level][];
          for(int i = 0; i < triangle.length; i++) triangle[i] = new int[i + 1];
          // 为 triangle 数组的每个元素赋值
          triangle[0][0] = 1;
          for(int i = 1; i < triangle.length; i++)
          {
              triangle[i][0] = 1;
              for(int j = 1; j < triangle[i].length - 1; j++) triangle[i][j] = triangle
[i - 1][j - 1] + triangle[i - 1][j];
              triangle[i][triangle[i].length - 1] = 1;
          }
          // 打印输出 triangle 数组的每个元素
          for(int i = 0; i < triangle.length; i++)
          {
            for(int j = 0; j < triangle[i].length; j++) System.out.print(triangle[i][j] +
" ");
              System.out.println(); // 换行
          }
     }
}
```

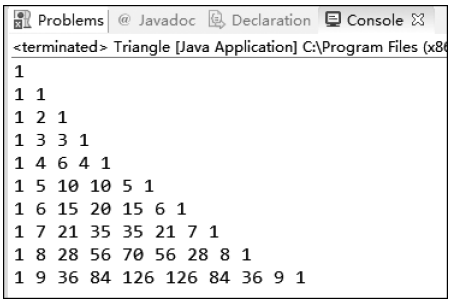

图 3-20 前 10 行杨辉三角形

程序运行结果如图 3-20 所示。

在使用数组参数时，需注意以下几点：

(1) 数组作形参时，形参表中数组名后的括号“[]”不能省略，括号个数和数组的维数相等。不需给出数组元素个数。

(2) 数组作实参时，实参表中，数组名后不需要加括号。

(3) 数组名作实参时，传递的是地址，而不是值，形参和实参具有相同的存储单元，所以形参数组的改变会影响到实参的改变。

(4) 数组元素作实参时，数组元素相当于是一个简单变量，其传递的是值，而不是地址。

3. 增强的 for 循环

如果程序只需顺序访问数组中每个元素，可以使用增强的 for 循环，它是 Java 5 新增功能。增强的 for 循环可以用来迭代数组和对象集合的每个元素。它的一般格式为

```
for(type identifier:expression) {
// 循环体
}
```

该循环的含义如下：对 expression(数组或集合)中的每个 type 类型的元素 identifier，执行一次循环体中的语句。这里，type 为数组或集合中的元素类型。expression 必须是一

个数组或集合对象。

下面使用增强的 for 循环实现求数组 marks 数组中各元素的和，代码如下：

```
double sum = 0;
 for(var score :marks){
      sum = sum + score;
}
System.out.println("总成绩 = " + sum);
```

提示：使用增强的 for 循环只能按顺序访问数组元素，并且只能使用元素而不能对元素进行修改。

3.4.3　数组应用

1. 排序

排序是一个数据序列的各元素按关键值大小进行升序或升序排列的过程。这里介绍冒泡排序、选择排序这两种常用的排序算法。

1）冒泡排序(Bubble Sort)

(1) 基本思想。

两两比较待排序数据元素的大小，发现两个数据元素的次序相反时即进行交换，直到没有反序的数据元素为止。

(2) 排序过程。

设想被排序的数组 R[1..N]垂直竖立，将每个数据元素看作有重量的气泡，根据轻气泡不能在重气泡之下的原则，从下往上扫描数组 R，凡扫描到违反本原则的轻气泡，就使其向上“漂浮”。如例 3-18 所示，第一趟排序后得出最大数 49，第二趟排序后，得出次大数 76，以此类推，如此反复进行，直至最后任何两个气泡都是轻者在上，重者在下为止。

【例 3-18】　冒泡升序排序。

初始关键字：49 38 65 97 76 13 27 49

排序具体过程：

```
49  13  13  13  13  13  13  13
38  49  27  27  27  27  27  27
65  38  49  38  38  38  38  38
97  65  38  49  49  49  49  49
76  97  65  49  49  49  49  49
13  76  97  65  65  65  65  65
27  27  76  97  76  76  76  76
49  49  49  76  97  97  97  97
```

可见，n 个数排序，需 n－1 趟比较。

代码如下：

```
public class BubbleSort
 {
    void bubble(int[ ] src)
```

```
        {
            int temp;
            int len = src.length;
            for(int i = 0;i < len;i++)
            {
               for(int j = i + 1;j < len;j++)
               {
                 if(src[i]> src[j])
                   {
                        temp = src[j];
                        src[j] = src[i];
                        src[i] = temp;
                   }
               }
           }
        }
       public static void main(String[ ] args)
       {
            int array[ ] = {49,38,65,97,76,13,27,49};
            System.out.print("原数组元素为:");
            for(int i = 0;i < array.length;i++)
                System.out.print(array[i] + " ");
            System.out.println();
            BubbleSort b = new BubbleSort();
            b.bubble(array);                          //调用函数
            System.out.print("冒泡升序排后数组元素为:");
            for(int i = 0;i < array.length;i++)
            System.out.print(array[i] + " ");
       }
  }
```

运行结果如图 3-21 所示。

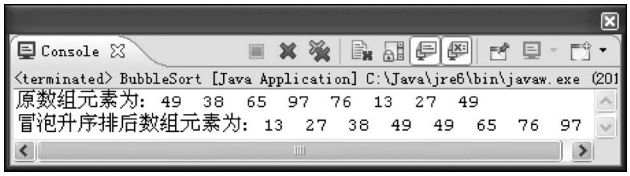

图 3-21 冒泡排序结果

2）选择排序(Selection Sort)

选择排序的基本思想：对待排序的记录序列进行 n－1 遍的处理，第 1 遍处理是将 L[1..n]中最小者与 L[1]交换位置，第 2 遍处理是将 L[2..n]中最小者与 L[2]交换位置，……，第 i 遍处理是将 L[i..n]中最小者与 L[i]交换位置。这样，每一趟从待排序的数据元素中选出最小(或最大)的一个元素，顺序放在已排好序的数列的最后，直到全部待排序的数据元素排完。当然，实际操作时，也可以根据需要，通过从待排序的记录中选择最大者与其首记录交换位置，按从大到小的顺序进行排序处理。

【例 3-19】 选择升序排序。

初始关键字 [49 38 65 97 76 13 27 49]

第一趟排序后 13 [38 65 97 76 49 27 49]

第二趟排序后 13 27 [65 97 76 49 38 49]
第三趟排序后 13 27 38 [97 76 49 65 49]
第四趟排序后 13 27 38 49 [49 97 65 76]
第五趟排序后 13 27 38 49 49 [97 97 76]
第六趟排序后 13 27 38 49 49 76 [76 97]
第七趟排序后 13 27 38 49 49 76 76 [97]
最后排序结果 13 27 38 49 49 76 76 97
代码如下：

```
public class SelectionSort {
     void doSelectionSort(int[] src)
     {
          int len = src.length;
          int temp;
          for(int i = 0;i < len;i++)
          {
              temp = src[i];
              int j;
              int samllestLocation = i;                    //最小数的下标
              for(j = i + 1;j < len;j++)
              {
                if(src[j]< temp)
                {
                    temp = src[j];                         //取出最小值
                    samllestLocation = j;                  //取出最小值所在下标
                }
             }
             src[samllestLocation] = src[i];
             src[i] = temp;
          }
       }
   public static void main(String[] args)
   {
             int array[] = {49,38,65,97,76,13,27,49};
             System.out.print("原数组元素为:");
             for(int i = 0;i < array.length;i++)
                  System.out.print(array[i] + " ");
             System.out.println();
             SelectionSort b = new SelectionSort();
             b.doSelectionSort(array);                     //调用函数
             System.out.print("选择升序排后数组元素为:");
             for(int i = 0;i < array.length;i++)
                  System.out.print(array[i] + " ");
    }
        }
```

运行结果如图 3-22 所示。

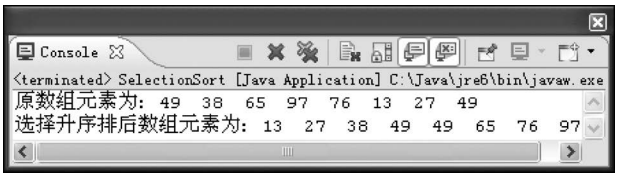

图 3-22　选择排序结果

2. 查找

查找是利用率给定的某个值，在一个数据集或数据序列中确定一个其关键字等于给定值记录或数据元素。若表中存在这样一个记录，则称查找是成功的；若表中不存在关键字等于给定值的记录，则称查找不成功。常用查找算法有顺序查找、二分查找等。

1）顺序查找

顺序查找又称线性查找。基本思想如下：从查找表的一端开始，向另一端逐个按给定值 K 与关键字进行比较，若找到，查找成功；并给出记录在表中的位置；若整表检测完，仍未找到与 K 值相同的关键字，则查找失败。其优点是：对表中数据的存储没有要求，数据序列可以是有序，也可以是无序；对于链表，只能进行顺序查找。顺序查找的缺点是：当 n 值很大时，平均查找长度较大，效率低。

2）二分查找

（1）算法思想。

二分查找又称折半查找，它是一种效率较高的查找方法。折半查找的算法思想是将数列按有序化（递增或递减）排列，查找过程中采用跳跃式方式查找，即先以有序数列的中点位置为比较对象，如果要找的元素值小于该中点元素，则将待查序列缩小为左半部分，否则为右半部分。通过一次比较，将查找区间缩小一半。折半查找是一种高效的查找方法。它可以明显减少比较次数，提高查找效率。但是，折半查找的先决条件是查找表中的数据元素必须有序。

折半查找法的优点是比较次数少，查找速度快，平均性能好；其缺点是要求待查表为有序表，且顺序存储，则导致插入删除困难。因此，折半查找方法适用于不经常变动而查找频繁的有序列表。

（2）算法步骤描述。

① 首先确定整个查找区间的中间位置 mid＝(left＋right)/2。

② 用待查关键字值与中间位置的关键字值进行比较：若相等，则查找成功；若大于，则在后（右）半个区域继续进行折半查找；若小于，则在前（左）半个区域继续进行折半查找。

③ 对确定的缩小区域再按折半公式，重复上述步骤。

最后，得到结果：要么查找成功，要么查找失败。折半查找的存储结构采用一维数组存放。

【例 3-20】 二分查找。

```
public class BinarySearch {
/**
     * 二分查找算法
     * @param srcArray 有序数组
     * @param key 查找元素
     * @return key 的数组下标，若没找到，则返回 -1
     */
    public static void main(String[] args)
    {
        int srcArray[] = {3,5,11,17,21,23,28,30,32,50,64,78,81,95,101};
        System.out.print("所在下标为:");
        System.out.println(binSearch(srcArray,0,srcArray.length-1, 81));
```

```
    }
     // 二分查找递归实现
    public static int binSearch(int srcArray[], int start, int end, int key)
    {
    int mid = (end - start) / 2 + start;
        if (srcArray[mid] == key)
        return mid;
        if (start >= end)
        return -1;
        else if (key > srcArray[mid])
             return binSearch(srcArray, mid + 1, end, key);
        else if (key < srcArray[mid])
             return binSearch(srcArray, start, mid - 1, key);
        return -1;
      }
    }
```

运行结果如图 3-23 所示。

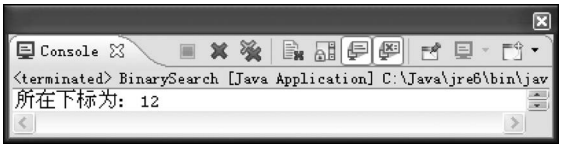

图 3-23　二分查找结果

3.4.4　数组 Array 类

Java. util. Arrays 类定义了若干静态方法对数组操作，包括对数组排序、在已排序的数组中查找指定元素、数组元素的复制、比较两个数组是否相等、将一个值填充到数组的每个元素中。

(1) public static int binarySearch (int[] a, int key)：根据给定的键值，查找该值在数组中的位置，如果找到指定的值，则返回该值的下标值。如果查找的值不包含在数组中，方法的返回值为(－插入点－1)。插入点为指定的值在数组中应该插入的位置。

(2) public static void sort(int[] a)：对数组 a 按自然顺序排序。

(3) public static void sort(int[] a, int fromIndex, int toIndex)：对数组 a 中的元素从起始下标 fromIndex 到终止下标 toIndex 之间的元素排序。

(4) public static double[] copyOf(double[] original, int newLength)：方法的 original 参数是原数组，newLength 参数是新数组的长度。

(5) public static void fill (int[] a, int val)：用指定的 val 值填充数组 a 中的每个元素。

(6) public static boolean equals(boolean[] a, boolean[] b)：比较布尔型数组 a 与 b 是否相等。

(7) public static String toString(int[] a)：将数组 a 的元素转换成字符串，它有多个重载的版本，方便对数组的输出。

上述操作都有多个重载的方法，可用于所有的基本数据类型和 Object 类型。

【例 3-21】 用 Arrays 的 sort()方法排序数组。

```
package com.boda.xy;
import Java.util.Arrays;
public class SortDemo
{
     public static void main(String[] args)
     {
          int[] a = {
               75, 53, 32, 12, 46, 199, 17, 54
          };
          System.out.print("排序前:");
          System.out.println(Arrays.toString(a));
          Arrays.sort(a);
          System.out.print("\n 排序后:");
          System.out.println(Arrays.toString(a));
     }
}
```

程序运行结果如图 3-24 所示。

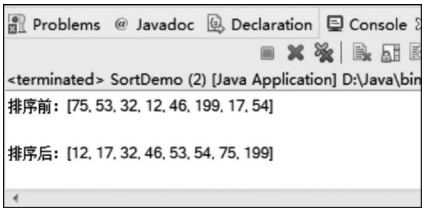

图 3-24 例 3-21 运行结果

3.5 典型案例

本节设计两个典型案例，练习 Java 语言基础知识的应用。

3.5.1 人脸识别

编写一个真正的人脸识别算法是一个相当复杂的任务，涉及图像处理、特征提取和机器学习等多个领域。但是，为了演示目的，我们可以编写一个非常简化的版本，其中使用数组来存储“人脸”数据，并编写一个简单的匹配算法。

在这个简化的例子中，我们将假设“人脸”数据是由两个整数数组表示的，每个数组包含一些简化的面部特征（例如，眼睛和鼻子的位置）。我们将编写一个函数来比较两个“人脸”数据数组，并返回一个表示它们是否匹配的布尔值。

请注意，这个例子只是为了教学目的，并不代表真实世界的人脸识别技术。

```
public class SimpleFaceRecognition {
// 假设的"人脸"数据，由两个整数数组表示
// 例如，第一个数组表示眼睛的位置，第二个数组表示鼻子的位置
public static int[][] face1 = { {10, 20}, {50, 50} };
public static int[][] face2 = { {15, 25}, {55, 55} };
//public static int[][] face1 = { {10, 20}, {50, 50} };
```

```
    //public static int[][] face2 = { {15, 25}, {65, 65} };
    // 人脸识别算法:比较两个人脸数据数组是否匹配
    public static boolean recognizeFace(int[][] face1, int[][] face2) {
    // 假设如果眼睛和鼻子的位置都在一定范围内(例如,±5 个单位),则认为它们是匹配的
        int eyeTolerance = 5;
        int noseTolerance = 5;
        // 比较眼睛的位置
        if (Math.abs(face1[0][0] - face2[0][0]) <= eyeTolerance &&
             Math.abs(face1[0][1] - face2[0][1]) <= eyeTolerance) {
           // 比较鼻子的位置
            if (Math.abs(face1[1][0] - face2[1][0]) <= noseTolerance &&
                Math.abs(face1[1][1] - face2[1][1]) <= noseTolerance) {
                return true;                    // 位置匹配,认为是同一张脸
             }
         }
           return false;                        // 位置不匹配,不是同一张脸
        }

        public static void main(String[] args) {
            // 测试人脸识别算法
            System.out.println("Matching face1 with face1: " + recognizeFace(face1, face1));
    // 应该返回 true
            System.out.println("Matching face1 with face2: " + recognizeFace(face1, face2));
    // 应该返回 true 或 false,取决于容忍度
        }
    }
```

运行程序得到如图 3-25 所示的结果,说明这两张图是匹配的。

若将语句 public static int[][] face1 = { {10, 20}, {50, 50} }; 和 public static int[][] face2 = { {15, 25}, {55, 55} }; 改为 public static int[][] face1 = { {10, 20}, {50, 50} }; 和 public static int[][] face2 = { {15, 25}, {65, 65} };,运行结果如图 3-26 所示,说明这两张图是不匹配的。

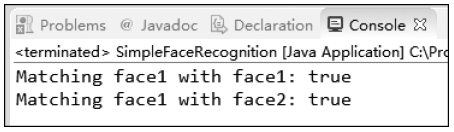

图 3-25　人脸识别匹配

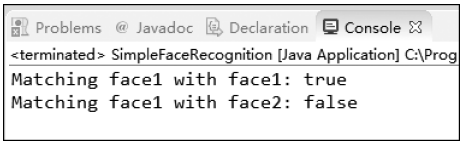

图 3-26　人脸识别不匹配

在这个例子中,我们定义了两个静态的二维整数数组 face1 和 face2 来表示两张人脸的特征位置。recognizeFace 方法接受两个人脸数组作为参数,并比较它们是否匹配。我们通过检查眼睛和鼻子的位置是否在定义的容忍度范围内来判断它们是否匹配。

请注意,这个简单的例子并没有使用任何图像处理或机器学习技术。在真实世界的应用中,人脸识别通常涉及从图像中提取复杂的特征,并使用这些特征训练机器学习模型来进行匹配。这通常需要使用像 OpenCV 这样的图像处理库,以及深度学习框架如 TensorFlow 或 PyTorch 等。

3.5.2　实现桥牌随机发牌

桥牌是一种文明、高雅、竞技性很强的智力游戏,由 4 个人分两组玩。桥牌使用普通扑

克牌去掉大小王后的 52 张牌，分为梅花(C)、方块(D)、红心(H)和黑桃(S)四种花色，每种花色有 13 张牌，从大到小的顺序为 A(最大)、K、Q、J、10、9、8、7、6、5、4、3、2(最小)。

打桥牌首先需发牌。本案例就是通过编程实现随机发牌。最后，按顺序显示输出 4 个玩家得到的牌。

案例代码如下：

```
import Java.util.Arrays;
public class BridgeCards
{
     public static void main(String[] args)
     {
          int[] deck = new int[52];
          String[] suits = {
              "♠", "♥", "♦", "♣"
          };
          String[] ranks = {
              "A", "K", "Q", "J", "10", "9", "8", "7", "6", "5", "4", "3", "2"
          };
          //初始化每一张牌
          for(int i = 0; i < deck.length; i++) deck[i] = i;
          // 打乱牌的次序
          for(int i = 0; i < deck.length; i++)
          {
              // 随机产生一个元素下标 0～51
              int index = (int)(Math.random() * deck.length);
              int temp = deck[i];                    // 将当前元素与产生的元素交换
              deck[i] = deck[index];
              deck[index] = temp;
          }
          // 对指定范围的数组元素排序
          Arrays.sort(deck, 0, 13);
          Arrays.sort(deck, 13, 26);
          Arrays.sort(deck, 26, 39);
          Arrays.sort(deck, 39, 52);
          // 显示所有 52 张牌
          for(int i = 0; i < 52; i++)
          {
              switch(i % 13)
              {
                  case 0 : System.out.print("玩家" + (i / 13 + 1) + ":");
              }
              String suit = suits[deck[i] / 13];     // 确定花色
              String rank = ranks[deck[i] % 13];    // 确定次序
              System.out.printf("%s%-3s", suit, rank);
              if((i + 1) % 13 == 0)
              {
                  System.out.println();
              }
          }
      }
}
```

程序运行结果如图 3-27 所示。

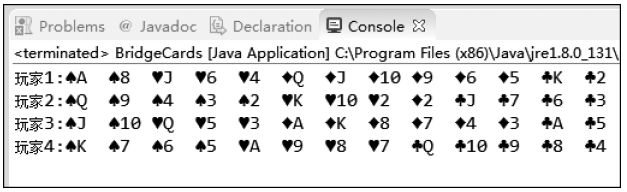

图 3-27　桥牌随机发牌结果

本案例的设计思路分析如下：

(1) 可以使用一个有 52 个元素的数组存储 52 张牌。为了区分 52 张牌，使用不同的元素值即可，为了方便这里使用 0～51。设元素值从 0～12 为黑桃，13～25 为红桃，26～38 为方块，39～51 为梅花。在创建数组后为每个元素赋值，如下所示。

```
int [] deck = new int[52];
for(var i = 0; i < deck.length; i++)            // 填充每个元素
    deck[i] = i;
```

(2) 为实现随机发牌，需要打乱数组元素的值，这里对每个元素，随机生成一个整数下标，将当前元素与产生的下标的元素交换，循环结束后，数组中的元素值被打乱。

```
for(var i = 0; i < deck.length;i++){
  // 随机产生一个元素下标 0～51
   int index = (int)(Math.random() * deck.length);
   int temp = deck[i];                    // 将当前元素与产生的下标的元素交换
   deck[i] = deck[index];
   deck[index] = temp;
}
```

(3) 牌的顺序打乱后，四个玩家的牌依次从 52 张牌取出 13 张牌。第 1 个 13 张牌属于第 1 个玩家，第 2 个 13 张牌属于第 2 个玩家，第 3 个 13 张牌属于第 3 个玩家，第 4 个 13 张牌属于第 4 个玩家。最后要求每个玩家的牌按顺序输出，因此需要对每个玩家的牌排序，这里使用 Arrays 类的 sort()方法，它可以对数组部分元素排序，例如对第 3 个玩家的牌排序，使用如下代码。

```
Arrays.sort(deck,26,39);
```

(4) 最后根据每张牌的数值转换为牌的名称(如，♥K)。为此，定义两个 String 数组，如下所示。

```
String[] suits = {"♠","♥","◆","♣"};
String[] ranks = {"A","K","Q","J","10","9","8","7","6","5","4","3","2"};
```

根据下面代码确定每张牌的花色和次序，然后输出。

```
String suit = suits[deck[i]/13];                // 确定花色
String rank = ranks[deck[i] % 13];              // 确定次序
System.out.printf("%s%-3s",suit,rank);
```

3.6　本章小结

本章详细介绍了 Java 编程的基础知识，包括 Java 程序的构成、Java 的基本数据类型、

变量和常量的定义与使用，Java 的运算符和表达式，Java 的流程控制语句，数组及 Java 常用的排序、查找算法。其中数据类型、变量和运算符是 Java 的基础，需要着重注意数据类型的转换规则，运算符中需分清|与||、& 与 &&；if/else、switch、while、do-while 和 for 等流程控制语句及二维数组的使用是本章的重点；常用的查找和排序算法可以以 C 语言为基础进行巩固学习。通过本章的学习，读者能够编写简单的 Java 程序，完成一些面向过程的基本操作。


习题答案

课后习题

1. Java 标识符命名的规则有哪些？
2. 下面哪些是合法的标识符？

```
$person   TwoUser   *point   this   endline   3person
```

3. Java 数据类型中包含哪些基本数据类型和哪些引用数据类型？
4. Java 运算符按功能分为哪些类型？其运算优先级大致如何？
5. 如何声明变量和常量？
6. 什么是强制类型转换？如何实现强制类型转换？
7. 在一个循环中使用 break、continue、return 语句有什么不同的效果？
8. 参照例 3-14 从键盘输入两个数据为上下限，然后输出上下限之间的所有素数。
9. 编写一个字符界面的 Java Application 程序，接受用户输入的字符，以“#”标志输入的结束。比较并输出按字典序排列的最小字符。

拓展阅读

人脸识别算法原理及应用，请扫描以下二维码查看。