第3章 数组与方法

本章学习目标

- 了解 Java 数组的定义。
- 掌握 Java 数组的常用操作。
- 掌握 Java 的方法定义与使用。
- 掌握 Java 方法重载与递归。
- 理解 Java 数组的引用传递。

数组能够用来存储固定大小的同类型元素，当在 Java 开发的过程中遇到需要定义多个相同类型的变量时，使用数组将会是一个很好的选择。例如，要存储 80 名学生的成绩，如果定义 80 个变量，会耗费大量的时间和精力，而此时如果使用数组不仅能够达到异曲同工之妙，还会提高代码的简洁性和扩展性。Java 中的方法是代码语句的集合，把这些语句组合在一起能够执行某个特定的功能，而且当遇到有些代码需要反复使用的情况时，也可以将代码声明成一个方法，以供程序反复调用，本章将对数组和方法的使用进行详细讲解。

3.1 数　　组

数组是一种数据结构，可以用来存储一系列的数据项，它是按照一定顺序排列的同种类型元素的集合。数组中的每一个元素都可以通过数组名和下标来确定，根据数组的维度可以分为一维数组、二维数组和多维数组等。使用数组时，可以通过数组元素的索引（下标）来访问数组元素，如数组元素的赋值和取值。

3.1.1 数组的定义

在 Java 中数组是相同类型元素的集合，可以存放上千万个数据，在一个数组中，数组元素的类型是唯一的，即一个数组中只能存储同一种数据类型的数据，而不能存储多种数据类型的数据。数组一旦定义完成就不能再修改数组长度，因为数组在内存中所占的大小是固定的，所以数组的长度不能改变，如果要修改就必须重新定义一个新数组或者引用其他的数组，因此数组的灵活性较差。

数组是可以保存一组数据的一种数据结构，它本身也会占用一个内存地址，因此数组是引用类型。定义数组的语法格式如下。

```
数据类型[] 数组名;
```

对于数组的声明也可用另外一种形式，其语法格式如下。

```
数据类型 数组名[];
```


上述两种不同语法格式声明的数组中,“[ ]”是一维数组的标识,从语法格式可以看出,它既可放置在数组名前面,也可以放在数组名后面。面向对象程序设计更侧重放在前面,保留放在后面是为了迎合 C 程序员的使用习惯,在这里推荐使用第一种格式。下面演示不同数据类型的数组声明,具体示例如下。

```
int[] a;                          // 声明一个 int 类型的数组
double b[];                       // 声明一个 double 类型的数组
```

上述示例中声明了一个 int 类型的数组 a 与一个 double 类型的数组 b,数组名是用来统一这组相同数据类型的元素名称,数组名的命名规则和变量相同。

3.1.2 数组的初始化

在 Java 程序开发中,使用数组之前都会对其进行初始化,这是因为数组是引用类型,声明数组只是声明一个引用类型的变量,并不是数组对象本身,只要让数组变量指向有效的数组对象,程序中就可使用该数组变量来访问数组元素。数组初始化,就是让数组名指向数组对象的过程,该过程主要分为两个步骤:一是对数组对象进行初始化,即为数组中的元素分配内存空间和赋值;二是对数组名进行初始化,即将数组名赋值为数组对象的引用。

通过两种方式可对数组进行初始化,即静态初始化和动态初始化。下面将演示这两种方式的具体语法。

1. 静态初始化

静态初始化是指由程序员在初始化数组时为数组每个元素赋值,由系统决定数组的长度。

数组的静态初始化有两种方式,具体示例如下。

```
Int[] array;                          //声明一个 int 类型的数组
array = new int[]{1,2,3,4,5};         //静态初始化数组
int[] array = new int[]{1,2,3,4,5};   //声明并初始化数组
```

对于数组的静态初始化也可简写,具体示例如下。

```
Int[] array ={1,2,3,4,5};             //声明并初始化一个 int 类型的数组
```

上述示例中静态初始化了数组,其中大括号包含数组元素值,元素值之间用逗号“,”分隔。此处注意,只有在定义数组的同时执行数组初始化才支持使用简化的静态初始化。

2. 动态初始化

动态初始化是指由程序员在初始化数组时指定数组的长度,由系统为数组元素分配初始值。

数组动态初始化的具体示例如下。

```
int[] array = new int[10];            // 动态初始化数组
```

上述示例会在数组声明的同时分配一块内存空间供该数组使用,其中数组长度是 10,

由于每个元素都为 int 型，因此上例中数组占用的内存共有 10×4＝40B。此外，动态初始化数组时，其元素会根据它的数据类型被设置为默认的初始值。本例数组中每个元素的默认值为 0，其他常见的数据类型默认值如表 3.1 所示。

表 3.1　数据类型默认值

成员变量类型	初　始　值	成员变量类型	初　始　值
byte	0	double	0.0D
short	0	char	空字符，'\u0000'
int	0	boolean	false
long	0L	引用数据类型	null
float	0.0F		

3.1.3　数组的常用操作

1. 访问数组

在 Java 中，数组对象有一个 length 属性，用于表示数组的长度，所有类型的数组都是如此。

获取数组的长度的语法格式如下。

```
数组名.length
```

接下来用 length 属性获取数组的长度，具体示例如下。

```
int[] list = new int[10];                    // 定义一个 int 类型的数组
int size = list.length;                      // size = 10,数组的长度
```

数组中的变量又称为元素，考虑到一个数组中的元素可能会很多，为了便于区分它们，每个元素都有下标(索引)，下标从 0 开始，如在 int[] list = new int[10]中，list[0]是第 1 个元素，list[1]是第 2 个元素，……，list[9]是第 10 个元素，也就是最后一个元素。因此，假如数组 list 有 n 个元素，那么 list[0]是第 1 个元素，而 list[n－1]则是最后一个元素。

如果下标值小于 0，或者大于或等于数组长度，编译程序不会报任何错误，但运行时将出现异常：ArrayIndexOutOfBoundsException:N，即数组下标越界异常，N 表示试图访问的数组下标。

2. 数组元素的存取

通过操作数组的下标可以访问到数组中的元素，也可以实现数组元素的存取。接下来演示数组元素的存取操作，如例 3-1 所示。

例 3-1　TestArray.java

```
1  public class TestArray {
2      public static void main(String[] args) {
3          //声明数组
4          int[] a = new int[5];
5          //存入数组元素
```

```
6           a[0] = 5;                    // 往数组的第一个元素中存入数据 5
7           a[1] = 10;                   // 往数组的第二个元素中存入数据 10
8           a[4] = 9;                    // 往数组的第五个元素中存入数据 9
9           //读取数组元素
10          System.out.print("数组中的元素为:");
11          System.out.println(a[0] + "," + a[1] + "," + a[2] + ", " + a[3] + ", " + a[4]);
12      }
13  }
```

程序运行结果如图 3.1 所示。

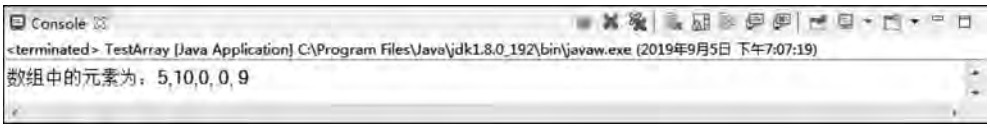

图 3.1　例 3-1 运行结果

从图 3.1 中可以看出,数组中的元素已经存取成功,而且在例 3-1 中,数组下标为 2、3 的位置中并未存入数据,但是却能取到数据为 0 的元素,可见声明为 int 类型的数组元素的默认值为 0。

3. 数组遍历

数组的遍历是指依次访问数组中的每个元素。接下来演示循环遍历数组,如例 3-2 所示。

例 3-2　TestArrayTraversal.java

```
1  public class TestArrayTraversal {
2      public static void main(String[] args) {
3          int[] list = {1, 2, 3, 4, 5};                  // 定义数组
4          for (int i = 0; i < list.length; i++) {        // 遍历数组元素
5              System.out.println(list[i]);               // 索引访问数组
6          }
7      }
8  }
```

程序的运行结果如图 3.2 所示。

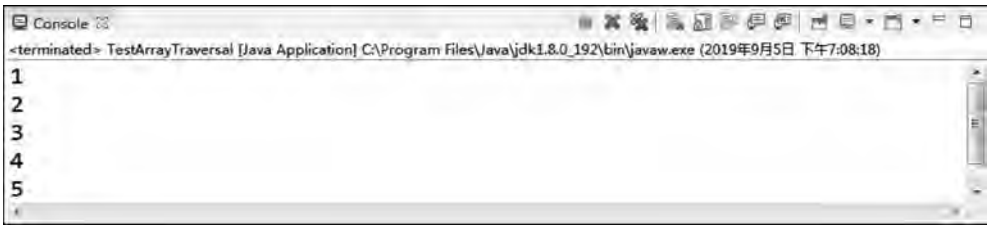

图 3.2　例 3-2 运行结果

在例 3-2 中,声明并静态初始化一个 int 类型的数组,然后利用 for 循环中的循环变量充当数组的索引,依次递增索引,从而遍历数组元素。

4. 数组最大值和最小值

通过前面已经掌握的知识，用数组的基本用法与流程控制语句的使用来实现得到数组中的最大值和最小值，首先把数组的第一个数赋值给变量 max 和 min，分别表示最大值和最小值，再依次判断数组的其他数值的大小，判断当前值是否是最大值或最小值，如果不是则进行替换，最后输出最大值和最小值。接下来通过一个案例来获取数组的最大值和最小值，如例 3-3 所示。

例 3-3 TestMostValue.java

```
1  public class TestMostValue {
2      public static void main(String[] args) {
3          // 定义数组
4          int[] score = {88, 62, 12, 100, 28};
5          int max = 0;                        // 最大值
6          int min = 0;                        // 最小值
7          max = min = score[0];               // 把第一个元素值赋给 max 和 min
8          for (int i = 1; i < score.length; i++) {
9              if (score[i] > max) {           // 依次判断后面元素值是否比 max 大
10                 max = score[i];             // 如果大，则修改 max 的值
11             }
12             if (score[i] < min) {           // 依次判断后面元素值是否比 min 小
13                 min = score[i];             // 如果小，则修改 min 的值
14             }
15         }
16         System.out.println("最大值:" + max);
17         System.out.println("最小值:" + min);
18     }
19 }
```

程序的运行结果如图 3.3 所示。

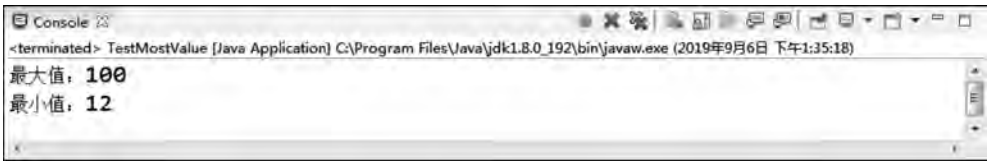

图 3.3 例 3-3 运行结果

在例 3-3 中，main()方法声明并静态初始化了 score 数组，并定义了两个变量 max 与 min，分别用来存储最大值与最小值。接着把 score 数组第一个元素 score[0]分别赋值到 max 与 min 中，然后使用 for 循环对数组进行遍历。下面通过一个图例来分析 min 和 max 的比较过程，如图 3.4 所示。

在图 3.4 中，max 与 min 最初存储的数值都是 score 数组的第一个元素 88，在遍历过程中只要遇到比 max 值还大的元素，就将该元素赋值给 max，遇到比 min 还小的元素，就将该元素赋值给 min。

5. 数组排序

数组排序是指数组元素按照特定的顺序排列。在实际应用中，经常需要对数据排序，如

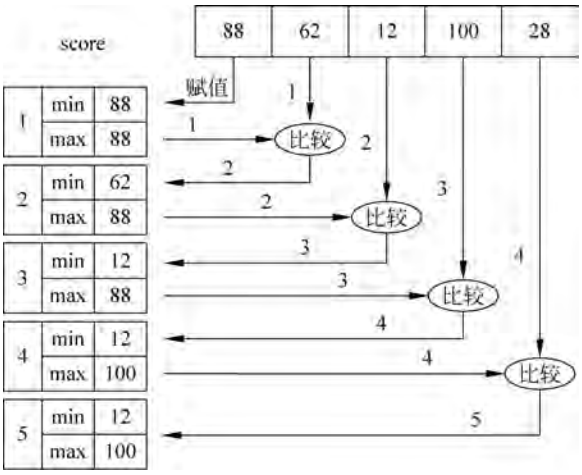

图 3.4　数组最大值和最小值比较过程

老师对学生的成绩排序。数组排序有多种算法,本节介绍一种简单的排序算法——冒泡排序。这种算法是不断地比较相邻的两个元素,较小的向上冒,较大的向下沉,排序过程如同水中气泡上升,即两两比较相邻元素,反序则交换,直到没有反序的元素为止,如例 3-4 所示。

例 3-4　TestBubbleSort.java

```
1  public class TestBubbleSort {
2      public static void main(String[] args) {
3          int[] array = {88, 62, 12, 100, 28};              // 定义数组
4          // 外层循环控制排序轮数
5          // 最后一个元素,不用再比较
6          for (int i = 0; i < array.length - 1; i++) {
7              // 内层循环控制元素两两比较的次数
8              // 每轮循环沉底一个元素,沉底元素不用再参加比较
9              for (int j = 0; j < array.length - 1 - i; j++) {
10                 // 比较相邻元素
11                 if (array[j] > array[j + 1]) {
12                     // 交换元素
13                     int tmp = array[j];
14                     array[j] = array[j + 1];
15                     array[j + 1] = tmp;
16                 }
17             }
18             // 打印每轮排序结果
19             System.out.print("第" + (i + 1) + "轮排序:");
20             for (int j = 0; j < array.length; j++) {
21                 System.out.print(array[j] + "\t");
22             }
23             System.out.println();
24         }
25         System.out.print("最终排序 :");
26         for (int i = 0; i < array.length; i++) {
```

```
27                System.out.print(array[i] + "\t");
28            }
29            System.out.println();
30        }
31 }
```

程序的运行结果如图 3.5 所示。

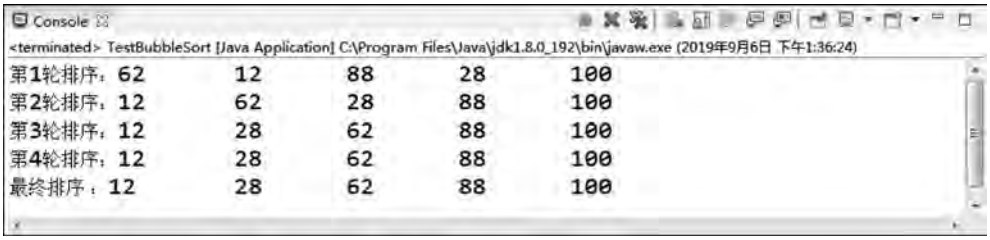

图 3.5　例 3-4 运行结果

在例 3-4 中，通过嵌套循环实现了冒泡排序。其中，外层循环是控制排序的轮数，每一轮可以确定一个元素位置，由于最后一个元素不需要进行比较，因此外层循环的轮数为 array.length－1。内层循环控制每轮比较的次数，每轮循环沉底一个元素，沉底元素不用再参加比较，因此，内层循环的次数为 array.length－1－i。内层循环的次数被作为数组的索引，索引循环递增，实现相邻元素依次比较，如果当前元素小于后一个元素，则交换两个元素的位置，如图 3.6 所示。

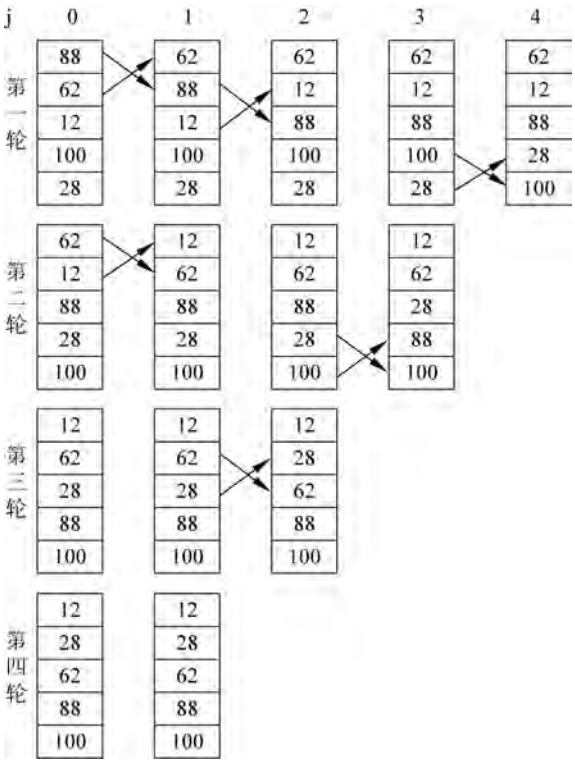

图 3.6　冒泡排序过程

在例 3-4 中,第 11～16 行代码实现了数组中两个元素的交换。首先定义一个临时变量 tmp 用于保存 array[j]的值,然后用 array[j+1]的值覆盖 array[j],最后将 tmp 的值赋给 array[j+1],从而实现了两个元素的交换,如图 3.7 所示。

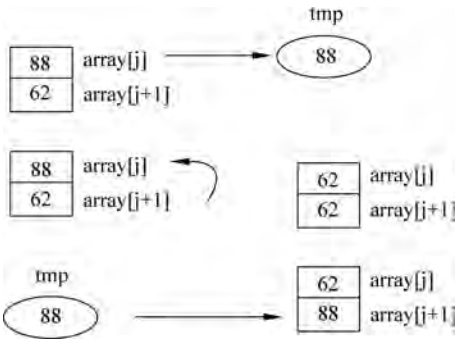

图 3.7　元素交换位置

3.1.4　数组的内存原理

数组是引用数据类型,因此数组变量就是一个引用变量,通常被存储在栈(Stack)内存中。数组初始化后,数组对象被存储在堆(Heap)内存中的连续内存空间,而数组变量存储了数组对象的首地址,指向堆内存中的数组对象。一维数组在内存中的存储原理如图 3.8 所示。

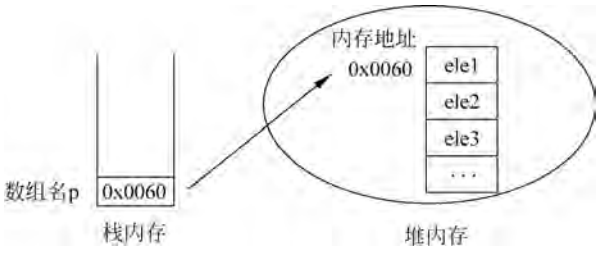

图 3.8　数组存储原理

在 Java 中,数组一旦初始化完成,数组元素的内存空间分配即结束,此后程序只能改变数组元素的值,而无法改变数组的长度。但程序可以改变一个数组变量所引用的数组,从而造成数组长度可变的假象。同理,在复制数组时,直接使用赋值语句不能实现数组的复制,这样做只是使两个数组引用变量指向同一个数组对象,如例 3-5 所示。

例 3-5　TestCopyArray.java

```
1  public class TestCopyArray {
2      public static void main(String[] args) {
3          int[] x = {88, 62, 12, 100, 28};
4          // 直接用赋值语句复制数组,赋的是数组的首地址
5          int[] y = x;
6          System.out.println(x);                    // 打印源数组名
7          System.out.println(y);                    // 打印目的数组名
```

```
8           x[0] = 22;                      // 修改源数组
9           System.out.println(y[0]);       // 访问目的数组
10      }
11  }
```

程序的运行结果如图 3.9 所示。

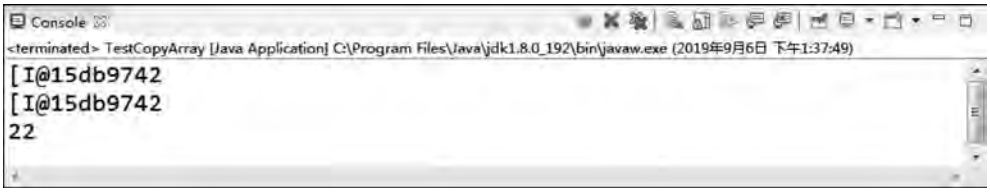

图 3.9　例 3-5 运行结果

在图 3.9 中，从程序运行结果可发现，数组变量保存的就是数组首地址。通过赋值运算符复制数组，复制的是数组的首地址，原数组名和目的数组名都指向实际的数组内存单元，因此它们操作的是同一数组，所以不能通过赋值运算符来复制数组，如图 3.10 所示。

要复制数组，可以使用循环来复制每一个元素，或者使用 System 类的 arraycopy()方法以及使用数组的 clone()方法来复制数组。

3.1.5　二维数组

虽然一维数组可以处理一些简单的一维模型，但在实际应用中模型却往往不止一维，例如棋盘，如图 3.11 所示。

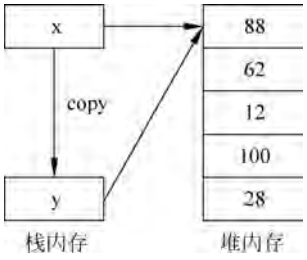

图 3.10　复制数组原理

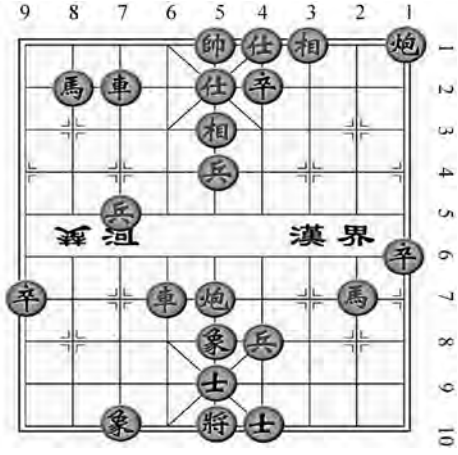

图 3.11　棋盘图

图 3.11 中有 10 行 9 列，因此它需要用二维模型来表示。Java 中用二维数组来模拟二维模型，因此，二维数组的第一维可以表示棋盘的 10 行，第二维可以表示棋盘的 9 列。假定数组名为 a，该棋盘用二维数组则可以表示为 a[10][9]，红帅的位置在第 1 行第 5 列，该位置就表示为 a[0][4]，其他位置以此类推。接下来详细讲解二维数组的声明及使用。

1. 二维数组

二维数组可以看成以数组为元素的数组,常用来表示表格或矩形。二维数组的声明、初始化与一维数组类似。

二维数组的声明,示例如下。

```
int[][] array;
int array[][];
```

二维数组动态初始化的示例如下。

```
array = new int[3][2];                    // 动态初始化 3×2 的二维数组
array[0] = {1, 2};                        // 初始化二维数组的第一个元素
array[1] = {3, 4};                        // 初始化二维数组的第二个元素
array[2] = {5, 6};                        // 初始化二维数组的第三个元素
```

上述示例定义了一个 3 行 2 列的二维数组,即二维数组的长度为 3,每个二维数组的元素是一个长度为 2 的一维数组。该二维数组元素的存储形式如图 3.12 所示。

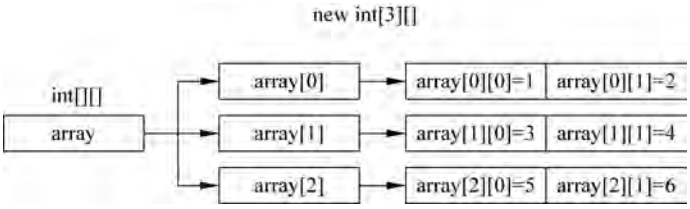

图 3.12　二维数组动态初始化

二维数组静态初始化的示例如下。

```
array = new int[][]{
    {1},
    {2, 3},
    {4}
};
```

对于二维数组的静态初始化也可用另一种形式,具体示例如下。

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

需要注意的是静态初始化由系统指定数组长度,不能手动指定,下面演示的是错误的静态初始化。

```
array = new int[3][3]{                    // 非法,静态初始化由系统指定数组长度,不能手动指定
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

以上静态初始化的二维数组元素的存储形式如图 3.13 所示。

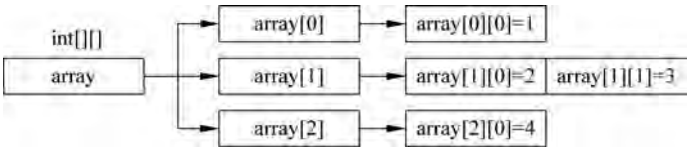

图 3.13　二维数组元素的存储形式

二维数组的每个元素是一个一维数组。二维数组 array 的长度是数组 array 的元素的个数，可由 array.length 得到；元素 array[i]是一个一维数组，其长度可由 array[i].length 得到。具体示例如例 3-6 所示。

例 3-6　TestTwoDimensionalArray.java

```
1  public class TestTwoDimensionalArray {
2      public static void main(String[ ] args) {
3          // 定义二维数组,3 行×3 列
4          int[ ][ ] array = new int[3][3];
5          // 动态初始化二维数组
6          // array.length 获取二维数组的元数个数
7          // array[i].length 获取二维数组元素指向的一维数组个数
8          for (int i = 0; i < array.length; i++) {
9              for (int j = 0; j < array[i].length; j++) {
10                 array[i][j] = 3 * i + j + 1;
11             }
12         }
13         // 打印二维数组
14         for (int i = 0; i < array.length; i++) {
15             for (int j = 0; j < array[i].length; j++) {
16                 System.out.print(array[i][j] + "\t");
17             }
18             System.out.println();
19         }
20     }
21 }
```

程序的运行结果如图 3.14 所示。

图 3.14　例 3-6 运行结果

在例 3-6 中定义了一个 3×3 的二维数组，用嵌套 for 循环为二维数组赋值和打印。由此可发现，每多一维，嵌套循环的层数就多一层，维数越多的数组其复杂度也就越高。该二维数组在内存中的存储原理如图 3.15 所示。

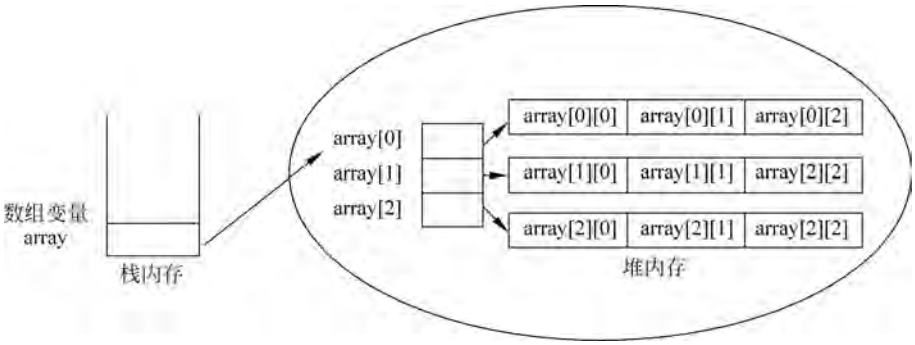

图 3.15　二维数组存储原理

2. 锯齿数组

二维数组中的每一行就是一个一维数组,因此,各行的长度就可以不同,这样的数组称为锯齿数组。创建锯齿数组时,可以只指定第一个下标,此时二维数组的每个元素为空,因此必须为每个元素创建一维数组,如例 3-7 所示。

例 3-7　TestJaggedArray.java

```
1  public class TestJaggedArray{
2      public static void main(String[] args) {
3          // 静态初始化锯齿数组
4          int[][] array = {
5                  {1, 2, 3, 4, 5},
6                  {2, 3, 4, 5},
7                  {3, 4, 5},
8                  {4, 5},
9                  {5}
10         };
11         // 动态初始化锯齿数组
12         int[][] x = new int[5][];
13         x[0] = new int[5];
14         x[1] = new int[4];
15         x[2] = new int[3];
16         x[3] = new int[2];
17         x[4] = new int[1];
18         // 为数组赋值
19         for (int i = 0; i < x.length; i++) {
20             for (int j = 0; j < x[i].length; j++) {
21                 x[i][j] = array[i][j];
22             }
23         }
24         // 打印二维数组
25         for (int i = 0; i < x.length; i++) {
26             for (int j = 0; j < x[i].length; j++) {
27                 System.out.print(x[i][j] + "\t");
28             }
29             System.out.println();
```

```
30          }
31      }
32 }
```

程序的运行结果如图 3.16 所示。

图 3.16　例 3-7 运行结果

在例 3-7 中，首先静态初始化锯齿数组 array 和动态初始化数组 x，然后通过嵌套 for 循环将锯齿数组 array 的数组元素值赋值给锯齿数组 x 的数组元素，最后通过嵌套 for 循环将锯齿数组 x 的数组元素打印出来。通过该示例，可以了解锯齿数据的基本形态和使用方法。

3.2　方　　法

方法(method)是一段可重用的代码，为执行一个操作组合在一起的语句集合，用于解决特定问题。在程序中多次重复使用相同的代码，重复地编写及维护比较麻烦，因此可以将此部分代码定义成一个方法，以供程序反复调用。

3.2.1　方法的定义

Java 中的方法定义在类中，一个类可以声明多个方法。方法的定义由方法名、参数、返回值类型以及方法体组成。

接下来说明如何定义方法，其语法格式如下。

```
修饰符 返回值类型 方法名([参数类型 参数名 1,参数类型 参数名 2,…]) {
    方法体
    return 返回值;
}
```

定义方法时需注意以下几点。

(1) 修饰符：方法的修饰符比较多，有对访问权限进行限定的，有静态修饰符 static，还有最终修饰符 final 等。

(2) 返回值类型：限定返回值的类型。

(3) 参数类型：限定调用方法时传入参数的数据类型。

(4) 参数名：是一个变量，用于接收调用方法时传入的数据。

(5) return：关键字，用于结束方法以及返回方法指定类型的值。

(6) 返回值：被 return 返回的值，该值返回给调用者。

接下来演示方法声明,如图 3.17 所示。

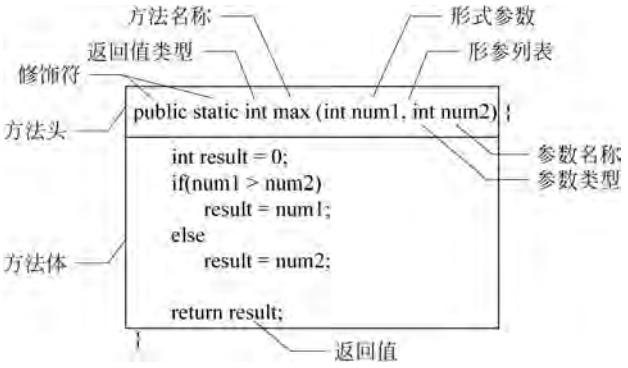

图 3.17　方法声明

在图 3.17 中,方法头中声明的变量称为形式参数,简称形参。当调用方法时,给参数传入的值称为实际参数,简称实参。形参列表是指形参的类型、顺序和数量。方法不需要任何参数,则形参列表为空。

方法可以有返回值,返回值必须为方法声明的返回值类型。如果方法没有返回值,则返回类型为 void,return 语句可以省略,如例 3-8 所示。

例 3-8　TestVoidMethod.java

```
1  public class TestVoidMethod {
2      public static void main(String[] args) {
3          int score = 78;
4          // 调用 void 方法
5          printGrade(score);
6          // 声明变量接收方法的返回值
7          char ret = getGrade(score);
8          System.out.println(ret);
9      }
10     // void 方法
11     public static void printGrade(double score) {
12         if (score < 0 || score > 100) {
13             System.out.println("成绩输入错误!");
14             return;
15         }
16         if (score >= 90.0) {
17             System.out.println('A');
18         } else if (score >= 80.0) {
19             System.out.println('B');
20         } else if (score >= 70.0) {
21             System.out.println('C');
22         } else if (score >= 60.0) {
23             System.out.println('D');
24         } else {
25             System.out.println('F');
26         }
```

```
    }
    // 带返回值的方法
    public static char getGrade(double score) {
        if (score >= 90.0) {
            return 'A';
        } else if (score >= 80.0) {
            return 'B';
        } else if (score >= 70.0) {
            return 'C';
        } else if (score >= 60.0) {
            return 'D';
        } else {
            return 'F';
        }
    }
}
```

程序的运行结果如图 3.18 所示。

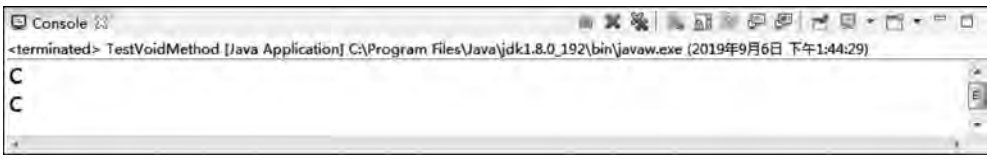

图 3.18　例 3-8 运行结果

例 3-8 中，定义了两个方法 printGrade()和 getGrade()，其中，printGrade()方法是用 void 修饰的，不返回任何值，而 getGrade()方法有返回值。用 void 修饰的方法不需要 return 语句，但它能用于终止方法返回到方法的调用者，控制程序的流程。当成绩不在 0～100，调用 printGrade()方法，程序将打印“成绩输入错误!”，执行 return 语句后，它后面的语句将不再执行，程序直接返回到调用者。

3.2.2　方法的调用

方法在调用时执行方法中的代码，因此要执行方法，必须调用方法。如果方法有返回值，通常将方法调用作为一个值来处理。如果方法没有返回值，方法调用必须是一条语句。具体示例如下。

```
int large = max(3, 4);                  // 将方法的返回值赋给变量
System.out.println(max(3,4));           // 直接打印方法的返回值
System.out.println("Hello World!");     // println 方法没有返回值,必须是语句
```

如果方法定义中包含形参，调用时必须提供实参。实参的类型必须与形参的类型兼容，实参顺序必须与形参的顺序一致。实参的值传递给方法的形参，称为值传递(pass by value)，方法内部对形参的修改不影响实参值。当调用方法时，程序控制权转移至被调用的方法。当执行 return 语句或到达方法结尾时，程序控制权转移至调用者，如例 3-9 所示。

例 3-9 TestCallMethod.java

```
1  public class TestCallMethod {
2      public static void main(String[] args) {
3          int n = 5;
4          int m = 2;
5          System.out.println("before main\t:n = " + n + ", m = " + m);
6          swap(n, m);
7          System.out.println("end main\t:n = " + n + ", m = " + m);
8      }
9      // 交换两个数
10     public static void swap(int n, int m) {
11         System.out.println("before swap\t:n = " + n + ", m = " + m);
12         int tmp = n;
13         n = m;
14         m = tmp;
15         System.out.println("end swap\t:n = " + n + ", m = " + m);
16     }
17 }
```

程序的运行结果如图 3.19 所示。

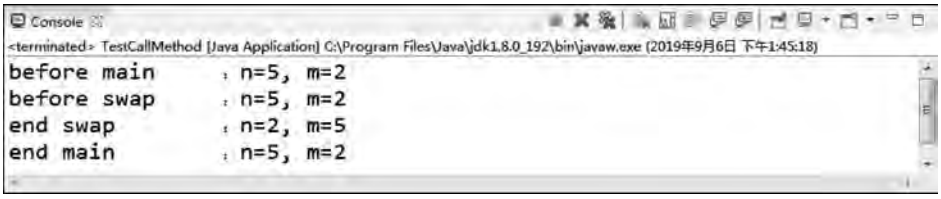

图 3.19 例 3-9 运行结果

在例 3-9 中,当调用 swap 方法时,程序将实参 n、m 的值传递给形参的 n、m,然后程序将控制流程转向 swap 方法。执行 swap 方法时,交换形参 n 和 m 的值,当 swap 方法执行完毕时,系统释放形参并将控制权返还给它的调用者 main 方法。因此,swap 方法不能交换实参 n 和 m 的值。

每当调用一个方法时,JVM 将创建一个栈帧,用于保存该方法的形参和变量。当方法调用结束返回到调用者时,JVM 释放相应的栈帧。每一个方法从调用开始到执行完成的过程,就对应着一个栈帧在 JVM 中从入栈到出栈的过程。接下来演示堆栈中调用方法的栈帧,如图 3.20 所示。

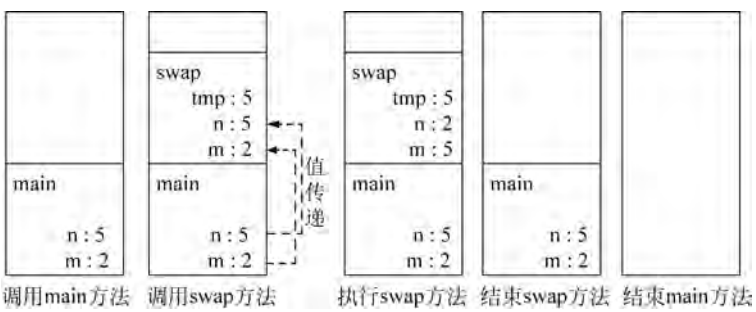

图 3.20 栈帧

在图 3.20 中，main 方法调用 swap 方法时，调用者 main 方法的栈帧不变，程序先为 swap 方法创建一个新的栈帧，用于保存形参 n、m 和局部变量 tmp 的值，再将实参值传递给形参，并保存在该栈帧中，方法内部操作的都是该方法栈帧中的值。当 swap 方法执行结束时，其对应的栈帧将被释放。

3.2.3 方法的重载

方法重载(overloading)是指方法名称相同，但形参列表不同的方法。调用重载的方法时，Java 编译器会根据实参列表寻找最匹配的方法进行调用，如例 3-10 所示。

例 3-10 TestOverload.java

```
1  public class TestOverload {
2      public static void main(String[] args) {
3          // 调用 max(int, int)方法
4          System.out.println("3 和 8 的最大值:" + max(3, 8));
5          // 调用 max(double, double)方法
6          System.out.println("3.0 和 8.0 的最大值:" + max(3.0, 8.0));
7          // 调用 max(double, double, double)方法
8          System.out.println("3.0、5.0 和 8.0 的最大值:" + max(3.0, 5.0, 8.0));
9          // 调用 max(double, double)方法
10         System.out.println("3 和 8.0 的最大值:" + max(3, 8.0));
11     }
12     // 返回两个整数的最大值
13     public static int max(int num1, int num2) {
14         int result;
15         if (num1 > num2)
16             result = num1;
17         else
18             result = num2;
19         return result;
20     }
21     // 返回两个浮点数的最大值
22     public static double max(double num1, double num2) {
23         double result;
24         if (num1 > num2)
25             result = num1;
26         else
27             result = num2;
28         return result;
29     }
30     // 返回三个浮点数的最大值
31     public static double max(double num1, double num2, double num3) {
32         return max(max(num1, num2), num3);
33     }
34 }
```

程序的运行结果如图 3.21 所示。

在图 3.21 中，从程序运行结果可以发现，max(3, 8)调用的是 max(int,int)方法，max(3.0,8.0)调用的是 max(double,double)方法，max(3.0,5.0,8.0)调用的是 max

图 3.21　例 3-10 运行结果

(double,double,double)方法。而且 max(3,8.0)也能被执行,实参 3 被自动转换为 double 类型,然后调用 max(double,double)方法。

为什么 max(3,8)不会调用 max(double,double)方法呢? 其实,max(double,double)和 max(int,int)与 max(3,8)都可能匹配。当调用方法时,Java 编译器会根据实参的个数和类型寻找最准确的方法进行调用。因为 max(int,int)比 max(double,double)更精确,所以 max(3,8)会调用 max(int,int)。

调用一个方法时,出现两个或多个可能的匹配时,编译器无法判断哪个是最精确的匹配,则会产生编译错误,称为歧义调用(ambiguous invocation),如例 3-11 所示。

例 3-11　TestAmbiguousInvocation.java

```
1  package test;
2  public class TestAmbiguousInvocation {
3      public static void main(String[] args) {
4          System.out.println(max(3, 8));
5      }
6      // 返回整数和浮点数的最大值
7      public static double max(int num1, double num2) {
8          if (num1 > num2)
9              return num1;
10         else
11             return num2;
12     }
13     // 返回浮点数和整数的最大值
14     public static double max(double num1, int num2) {
15         if (num1 > num2)
16             return num1;
17         else
18             return num2;
19     }
20 }
```

程序的运行结果如图 3.22 所示。

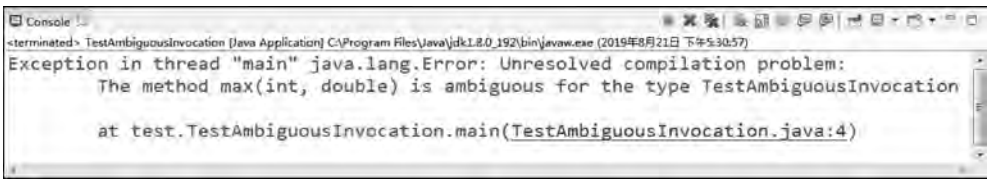

图 3.22　例 3-11 运行结果

在图 3.22 中，程序编译错误并提示“对 max 的引用不明确”，原因在于 max(int, double)和 max(double,int)与 max(3,8)都匹配，从而产生歧义，导致编译错误。

方法只能根据参数列表(参数类型、参数顺序和参数个数)进行重载，而不能通过修饰符或返回值来重载。

注意：在同一个类中，方法重载指的是可以定义多个名称相同，形参列表不同的方法(即形参的排列顺序、类型、个数，满足任意一个不同即可)，它对方法的返回值不做要求。在方法的调用上系统会自动根据传过来的参数数量和类型来决定使用哪一个方法。

3.2.4 方法的递归

方法的递归是指一个方法直接或间接调用自身的行为，递归必须要有结束条件，否则会无限地递归。递归用于解决使用简单循环难以实现的问题，如例 3-12 所示。

例 3-12 TestRecursion.java

```
1  public class TestRecursion {
2      public static void main(String[] args) {
3          System.out.println("4 的阶乘:" + fact(4));
4      }
5      /*
6          计算阶乘
7          阶乘计算公式:
8          0!= 1
9          n! = n * (n-1)!; n>0
10     */
11     public static long fact(int n) {
12         // 结束条件
13         if (n == 0)
14             return 1;
15         return n * fact(n - 1);
16     }
17 }
```

程序的运行结果如图 3.23 所示。

图 3.23 例 3-12 运行结果

在例 3-12 中，定义 fact()方法用于计算阶乘，方法是将数学上的阶乘公式转换为代码。当用 n=0 调用该方法时，程序立即返回结果，这种简单情况称为结束条件，如果没有终止条件，就会出现无限递归。当用 n>0 调用该方法时，就将这个原始问题分解成计算 n-1 的阶乘的子问题，持续分解，直到问题达到最终条件为止，就将结果返回给调用者。然后调用者进行计算并将结果返回给它自己的调用者，过程持续进行，直到结果返回原始调用者为止。原始问题就可以通过将 fact(n-1)的结果乘以 n 得到。调用过程称为递归调用，如图 3.24 所示。

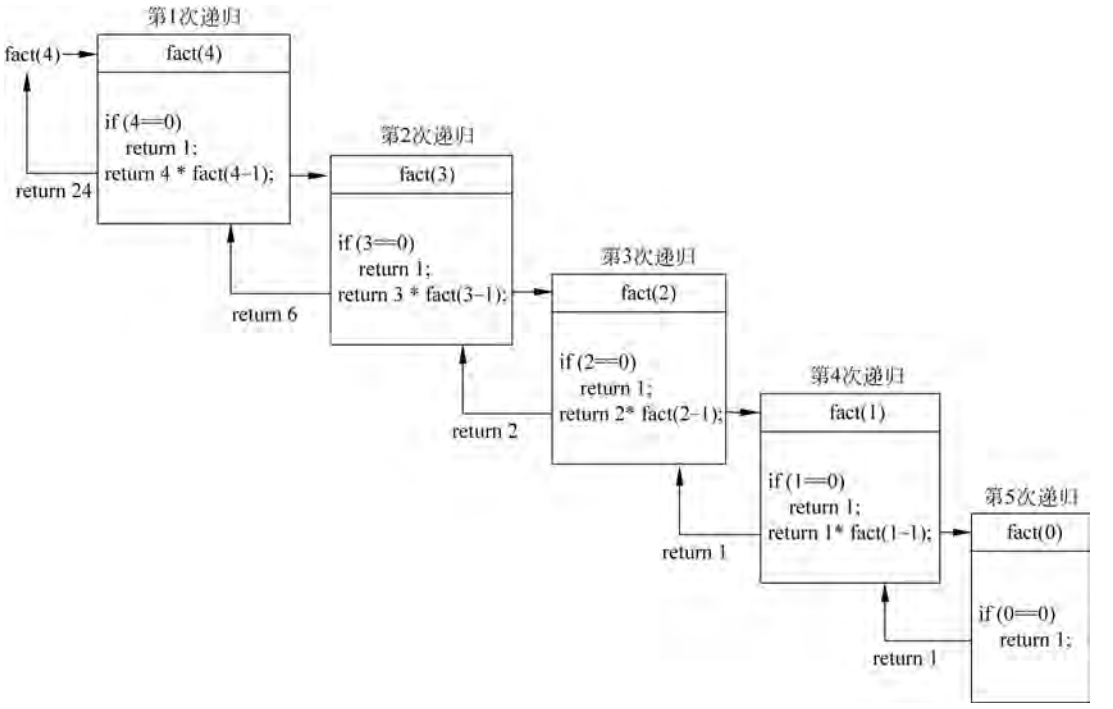

图 3.24　递归原理

在图 3.24 中,描述了例 3-12 中的递归调用过程,整个递归过程中 fact()方法被调用了 5 次,每次调用 n 的值都会递减,当 n 的值为 0 时,所有递归调用的方法都会以相反的顺序相继结束,所有的返回值会进行累乘,最终得到结果 24。

3.3　数组的引用传递

在方法调用时,参数按值传递,即用实参的值去初始化形参。对于基本数据类型,形参和实参是两个不同的存储单元,因此方法执行中形参的改变不影响实参的值;对于引用数据类型,形参和实参存储的是引用(内存地址),都指向同一内存单元,在方法执行中,对形参的操作实际上就是对实参的操作,即对执行内存单元的操作,因此,方法执行中形参的改变会影响实参。

向方法传递数组时,方法的接收参数必须是符合其类型的数组;从方法返回数组时,返回值类型必须明确地声明其返回的数组类型。数组属于引用类型,所以在执行方法中对数组的任何操作,结果都将保存下来,如例 3-13 所示。

例 3-13　TestRefArray.java

```
1  public class TestRefArray {
2          public static void main(String[] args) {
3          int[] array = {1, 3, 5};
4          rev(array);                              // 将数组元素反序
5          System.out.print("数组的反序:");
```

```
6           printArray(array);                          // 打印反序后的数组
7           int[] copy = copy(array);                   // 复制数组
8           array[0] = 9;                               // 修改源数组
9           System.out.print("修改源数组:");
10          printArray(array);                          // 打印源数组
11          System.out.print("复制的数组:");
12          printArray(copy);                           // 打印复制数组
13      }
14      // 将数组元素反序
15      public static void rev(int[] pa) {
16          for (int i = 0, j = pa.length - 1; i < j; i++, j--) {
17              int tmp = pa[i];
18              pa[i] = pa[j];
19              pa[j] = tmp;
20          }
21      }
22      // 复制数组元素
23      public static int[] copy(int[] pa) {
24          int[] newarray = new int[pa.length];
25          for (int i = 0; i < pa.length; i++) {
26              newarray[i] = pa[i];
27          }
28          return newarray;
29      }
30      // 打印数组元素
31      public static void printArray(int[] pa) {
32          for (int i = 0; i < pa.length; i++) {
33              System.out.print(pa[i] + "\t");
34          }
35          System.out.println();
36      }
37  }
```

程序的运行结果如图 3.25 所示。

图 3.25　例 3-13 运行结果

在例 3-13 中，定义了 3 个方法 rev()、printArray()和 copy()，其中，rev()反序一个数组，printArray()打印数组元素，而 copy()复制一个数组。因为数组是引用类型，所以，在方法中修改数组，其结果也会保存下来。接下来演示 rev()反序数组的过程，如图 3.26 所示；copy()复制数组的过程如图 3.27 所示。

在图 3.26 中，声明的 array 数组元素是 1、3、5。当将此数组传递到 rev()方法中时，使用形参 pa 接收，也就是说，此时的 array 实际上是将使用权传递给了 rev()方法，为数组起

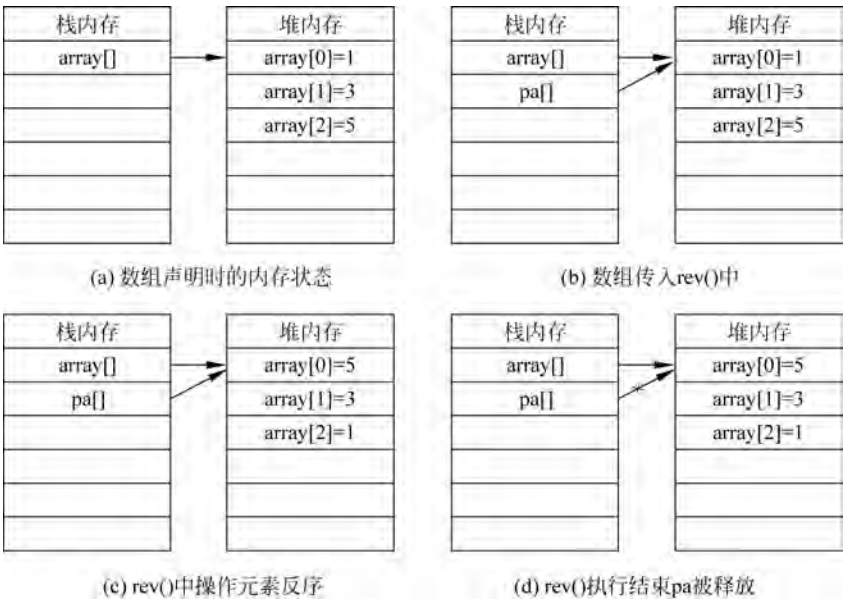

图 3.26　传递数组

了一个别名 pa,然后在 rev()方法中通过 pa 进行元素反序操作。rev()方法执行完毕,局部变量 pa 被释放,但是对于数组元素的改变却保留了下来,这就是参数在数组的引用传递的过程。

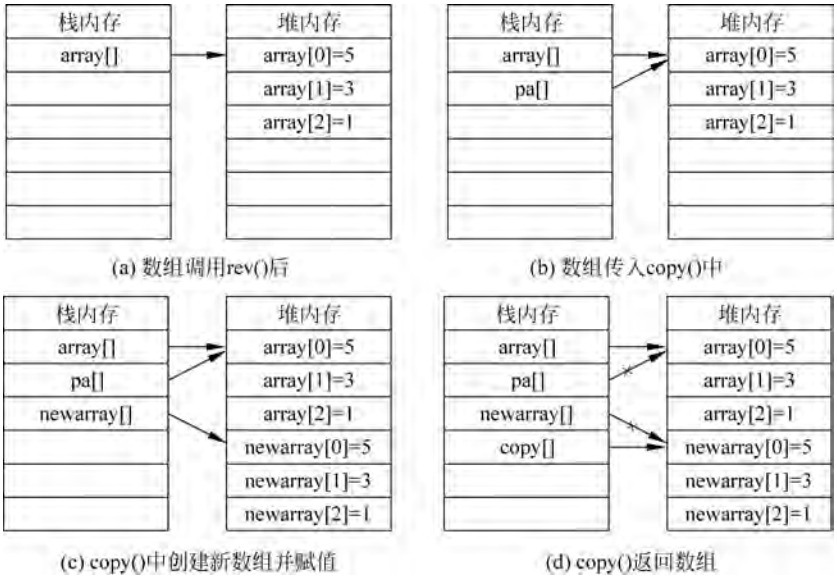

图 3.27　返回数组

在图 3.27 中,声明的 array 数组元素是 1、3、5。当将此数组传递到 copy()方法时,使用形参 pa 接收,同时在该方法中创建新数组 newarray,然后通过 pa 将数组元素值复制到 newarray 数组。copy()方法执行完毕之后,局部变量 pa 和 newarray 被释放,但是 newarray 被传递到 copy 数组,这就是返回值是数组的引用传递的过程。

小　　结

通过本章的学习，读者能够掌握 Java 数组与方法的使用。重点要熟悉的是在实际开发中，要完成一个复杂的程序，可将一个复杂的程序简化成若干个方法来实现，如遇到需要定义多个相同类型的变量时，可以使用数组。

习　　题

1. 填空题

(1) 当调用方法时，给参数传入的值称为实际参数，简称_____。

(2) 在 Java 中，当一个方法不需要返回数据时返回类型必须是_____。

(3) 一个数组中只能存储同一种_____的数据。

(4) 数组的元素可通过_____来访问。

(5) _____是指方法名称相同，但形参列表不同的方法。

2. 选择题

(1) 定义了一维 int 型数组 a[10]后，下面错误的引用是(　　)。

A. a[0]=1;　　B. a[10]=2;

C. a[0]=5 * 2;　　D. a[1]=a[2] * a[0];

(2) 数组对象在 Java 中存储在(　　)中。

A. 栈　　B. 队列　　C. 堆　　D. 链表

(3) 方法的(　　)是指一个方法直接或间接调用自身的行为。

A. 传递　　B. 递归　　C. 访问　　D. 方法

(4) 数组 a 的第三个元素表示为(　　)。

A. a[2]　　B. a(3)　　C. a(2)　　D. a[3]

(5) 关于数组作为方法的参数时，向方法传递的是(　　)。

A. 数组的元素　　B. 数组的引用

C. 数组的栈地址　　D. 数组自身

3. 思考题

(1) 什么时候为数组分配内存？

(2) 数组一旦被创建，大小能不能改变？

(3) 实参是如何传递给方法的？

(4) 什么是方法的重载？

4. 编程题

(1) 编写方法返回两个整数的最大公约数和最小公倍数。

(2) 编写程序计算字符数组中每个字符出现的次数。