第四章 数组

1.数组概述

> 数组，表示一块连续的内存空间，可用来存储多个数据(元素)，要求元素类型要一致。

数组初识：

1）如果有五个数据1 2 3 4 5，需要去接收、保存、操作这些数据，需要五个变量

```
int a1 = 1;
int a2 = 2;
int a3 = 3;
int a4 = 4;
int a5 = 5;
// int类型变量，用来标识1块内存，只能用来存放1个数值
```

2）现在有了数组，我们可以使用一个数组来存储这五个数据

```
int[] arr = {1,2,3,4,5};
// 这里使用一个数组来保存这5个元素值
// 数组表示一块连续的内存空间，可以用来存放多个元素值
```

3）我们对数组其实不陌生，之前课程已经接触过，大家可看下面代码

```
//这个参数args的类型是字符串数组
public static void main(String[] args) {
    // 注意：下面代码看不懂没有关系，本章学完能看懂即可

    //控制循环输出的次数
    int num = 1;

    //如果main方法的参数args有接收到参数值的话
```

```
    if(args.length > 0) {
        //把接收到的值转换为int类型，并赋值给变量num
        num = Integer.parseInt(args[0]);
    }

    //循环输出hello，默认输出次数为1，如果用户给main方法传参了，则按照用户的要求的次数进行输出
    for(int i = 0; i < num; i++) {
        System.out.println("hello");
    }
}
```

2.数组定义

数组的定义有2种格式，分别如下：

- **数据类型[] 数组名；** （推荐用法）

  示例：

  ```
  int[] arr;
  double[] arr;
  char[] arr;
  ```

- **数据类型 数组名[];**

  示例：

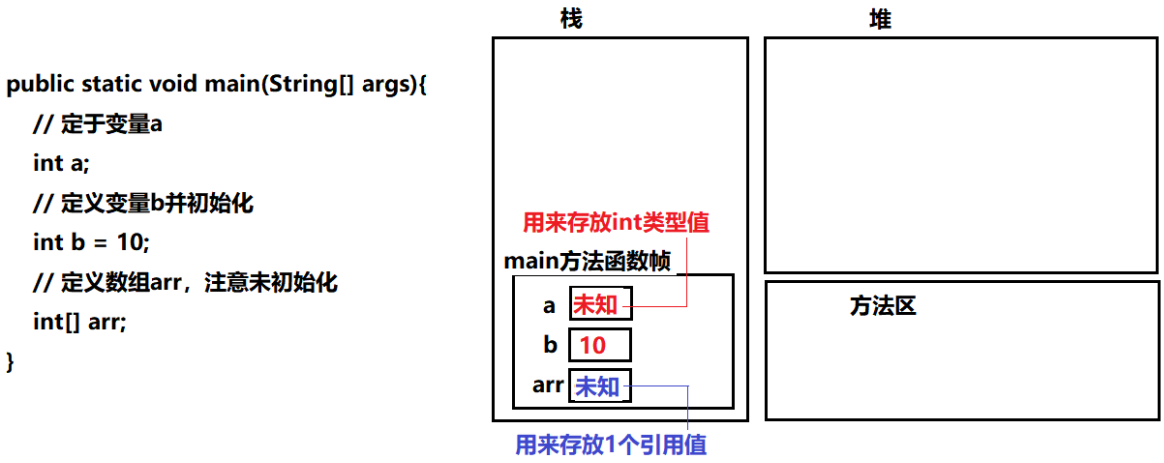
  ```
  int arr[];
  double arr[];
  char arr[];
  ```

两种方式都可以定义数组，我们推荐第一种用法！

> 数组内存理解

大家看下面代码，我们来分析下数组的内存构成。

```
public static void main(String[] args){
    // 定于变量a
    int a;
    // 定义变量b并初始化
    int b = 10;
    // 定义数组arr，注意未初始化
    int[] arr;

    // 编译报错
    //System.out.println(a);
    System.out.println(b);
    // 编译报错
    //System.out.println(arr);
}
```



注意1：数组是引用数据类型，用来存储一个引用值(可理解为地址值)

注意2：数组没有进行初始化，不可以直接使用

3.数组初始化

> 定义数组(开辟栈空间内存)的同时，给其赋上初值，就叫做数组的初始化!

3.1 动态初始化

格式:

```
数据类型[] 数组名 = new 数据类型[数组长度];
```

案例:

```
int[] arr1 = new int[3];
double[] arr2 = new double[2];
String[] arr3 = new String[4];
```

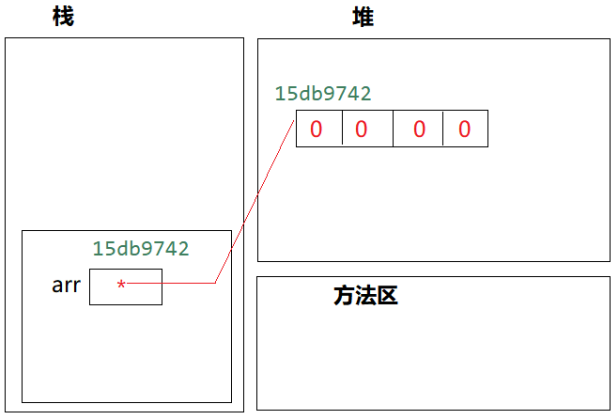
注意:

- `new`是一个关键字，表示为数组开辟内存空间
- 等号两边的数据类型要一致（先记住，后期会补充不一致的情况）
- 数组长度必须要有，可以>=0（一般大于0），但不能为负数

3.2 内存构成

> 数组名标识的那块内存(栈空间)，存放了一个引用值(地址值)，通过该地址值可以找到堆空间相应内存（用来存放数组中所有元素）。
>
> 堆空间内存存在默认初始化：整形数初始化为0，浮点数0.0，引用类型null，字符类型初始化`\u0000`

```java
public static void main(String[] args) {
    // 定义并初始化数组arr
    int[] arr = new int[4];

    // 输出结果为：[I@15db9742
    System.out.println(arr);
}
```



数组名arr 标识的那块内存(栈空间)，存放了一个引用值(地址值)，通过该地址值可以找到堆空间相应内存（用来存放数组中所有元素）。

思考：如何验证上图是否正确呢?

```java
package com.briup.chap04;

public class Test032_Memory {
    // 借助下面案例，可以验证上图
    public static void main(String[] args) {
        // 定义并初始化数组arr
        int[] arr = new int[4];

        /* 验证：上图中堆空间的存在 */
        /* 输出结果为：[I@15db9742
         *  [ :              当前的空间是一个数组类型
         *  I :              当前数组容器中所存储的数据类型
         *  @ :              分隔符
         *  15db9742 :       堆空间十六进制内存地址
        */
        System.out.println(arr);
        System.out.println("--------------------");

        /* 验证：元素初始值为0 */
        // 访问数组元素可以通过数组下标来实现，具体格式：数组名[下标]
        // 注意，下标从0开始，最大值为 数组长度-1
        System.out.println(arr[0]); //第1个元素
        System.out.println(arr[3]); //第4个元素
        System.out.println("--------------------");
```

```
        //数组的遍历
        for(int i = 0; i < 4; i++) {
            System.out.println(arr[i]);
        }
    }
}
```

3.3 数组下标

> 数组的下标的区间为`[0, 数组长度-1]`。
>
> 如果数组长度为`length`，那么数组下标的最小值为0，下标最大值为`length-1`。

通过下标可以访问数组中元素：

```
//数组长度为4，那么其下标就是0~3
int[] arr = new int[4];

//可以通过下标获取数组元素值
System.out.println(arr[0]);
System.out.println(arr[3]);
```

通过数组下标给数组元素赋值：

```
int[] arr = new int[4];
arr[0] = 337;
arr[1] = 340;
arr[2] = 348;
arr[3] = 352;
```

结合循环来赋值或者取值：

```java
int[] arr = new int[4];

//数组下标的取值范围，从0开始，到数组长度-1
for(int i = 0; i < 4; i++){
    arr[i] = 10 + i;
}

//获取数组每个元素的值，并且输出
for(int i = 0; i < 4; i++){
    System.out.println(arr[i]);
}
```

3.4 数组长度

数组长度，是指在一个数组中，可以存放同一类型元素的最大数量。

获取数组长度固定格式： `数组名.length`

```java
int[] arr = new int[4];
System.out.println(arr.length);
```

数组长度注意事项：

- 数组长度，必须在创建数组对象的时候就明确指定
- 数组长度，一旦确定，就无法再改变
- 数组长度，可以>=0（一般大于0），但不能为负数

借助循环赋值或遍历的最终形式：

```java
package com.briup.chap04;

public class Test035_Length {
    public static void main(String[] args) {
```

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

        //遍历数组中初始元素值，默认为0
        for(int i = 0; i < arr.length; i++){
            System.out.println(arr[i]);
        }

        System.out.println("--------------");

        //逐个元素赋值，借助 数组名.length 完成
        for(int i = 0;i < arr.length; i++){
            arr[i] = 10 + i;
        }

        //遍历数组中所有元素值
        for(int i = 0; i < arr.length; i++){
            System.out.println(arr[i]);
        }
    }
}
```

3.5 数组默认值

> 数组在创建时，会开辟2块内存，数组名对应栈空间那块内存，数组元素会存放在堆空间。
>
> 堆空间数组每一个元素位置上，存在相应的默认值，要么为0，要么为0.0，要么为null。

```
//byte、short、int、long类型数组中的默认值为 0
//例如
int[] a = new int[4];//默认4个数据全是0

//float、double类型数组中的默认值为 0.0
double[] d = new double[4];//默认4个数据全是0.0
```

```
//boolean类型数组中的默认值为 false
boolean[] d = new boolean[4];//默认4个数据全是false

//char类型数组中的默认值为 '\u0000'
char[] d = new char[4];//默认4个数据全是'\u0000'

//引用类型数组中的默认值为 null【不理解引用类型没关系，后续会补充】
String[] d = new String[4];//默认4个数据全是null
```

3.6 静态初始化

在创建数组的同时，直接初始化数组元素的值，称为数组的静态初始化。

静态初始化格式：

- 完整版格式

  `数据类型[] 数组名 = new 数据类型[]{元素1,元素2,...};`

- 简化版格式

  `数据类型[] 数组名 = {元素1,元素2,...};`

注意：数组静态初始化书写方式要严格按照上述两种方式！

示例：下面以创建int类型数组对象进行说明

```
package com.briup.chap04;

//数组静态初始化
public class Test036_Init {
    //定义一个方法，专门输出数组元素
    public static void outArray(int[] arr) {
        for(int i = 0; i < arr.length; i++) {
            System.out.println("arr[" + i + "]: " + arr[i]);
        }
```

```
    }

    public static void main(String[] args) {
        // 完整版本静态初始化
        int[] arr1 = new int[]{1,3,5,7,9};
        outArray(arr1);

        System.out.println("--------------");

        // 简化版本静态初始化
        int[] arr2 = {1,3,5,7,9};
        outArray(arr2);

        System.out.println("--------------");

        // 先定义数组名【在栈空间开辟内存】
        int[] arr3;
        // 再去堆空间开辟内存并静态初始化，然后将堆空间地址值 放入 (数组名标识的)栈空间内存
        arr3 = new int[]{1,3,5,7,9};
        outArray(arr3);
    }
}
```

下面是错误写法：

```
public static void main(String[] args) {
    //省略...

    //错误方式1：不能明确数组长度
    int[] arr4 = new int[3]{1,2,3};

    //错误方式2：简化版格式必须严格按照上述格式书写，不能分两行书写
    int[] arr5;
    arr5 = {1,2,3};
}
```

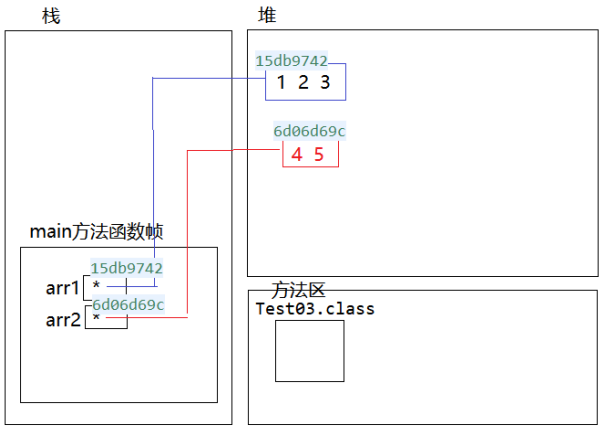
3.7 内存补充

1、两个数组内存结构图

```
public class Test037_Extend {
    public static void main(String[] args) {
        int[] arr1 = new int[] {1,2,3};

        int[] arr2 = {4,5};

        System.out.println(arr1);   //[I@15db9742
        System.out.println(arr2);   //[I@6d06d69c
        System.out.println("---------------");

        for(int i = 0; i < arr1.length; i++) {
            System.out.println(arr1[i]);
        }
        System.out.println("---------------");

        for(int i = 0; i < arr2.length; i++) {
            System.out.println(arr2[i]);
        }
    }
}
```

内存结构图如下：

```
public class Test03 {
      public static void main(String[] args) {
            int[] arr1 = new int[] {1,2,3};

            int[] arr2 = {4,5};

            System.out.println(arr1);     //[I@15db9742
            System.out.println(arr2);     //[I@6d06d69c
            System.out.println("---------------");

            for(int i = 0; i < arr1.length; i++) {
                  System.out.println(arr1[i]);
            }
            System.out.println("---------------");

            for(int i = 0; i < arr2.length; i++) {
                  System.out.println(arr2[i]);
            }
      }
}
```

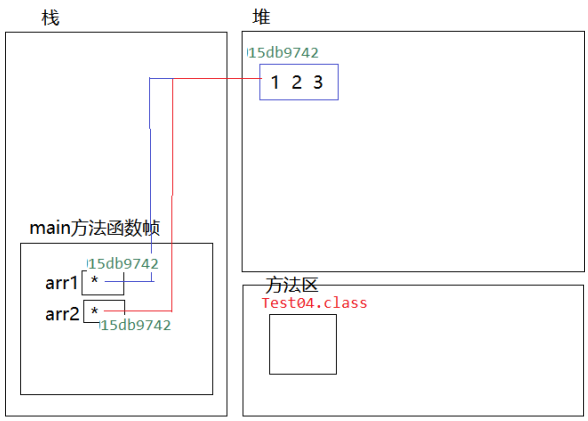


2、使用数组赋值

```java
public class Test037_Extend2 {
    public static void main(String[] args) {
        int[] arr1 = new int[] {1,2,3};
        System.out.println(arr1);   //[I@15db9742

        int[] arr2 = arr1;
        System.out.println(arr2);   //[I@15db9742
        System.out.println("--------------");

        for(int i = 0; i < arr1.length; i++) {
            System.out.println(arr1[i]);
        }
        System.out.println("--------------");

        for(int i = 0; i < arr2.length; i++) {
            System.out.println(arr2[i]);
        }
    }
}
```

内存结构图如下：

```java
public class Test04 {
    public static void main(String[] args) {
        int[] arr1 = new int[] {1,2,3};
        System.out.println(arr1);     //[I@15db9742

        int[] arr2 = arr1;
        System.out.println(arr2);     //[I@15db9742
        System.out.println("--------------");

        for(int i = 0; i < arr1.length; i++) {
            System.out.println(arr1[i]);
        }
        System.out.println("--------------");

        for(int i = 0; i < arr2.length; i++) {
            System.out.println(arr2[i]);
        }
    }
}
```



4.数组异常

使用数组的过程中，经常会遇到以下2种异常，具体如下：

4.1 索引越界

- 出现原因

```
package com.briup.chap04;

public class Test041_IndexOut {
    public static void main(String[] args) {
        int[] arr = new int[4];

        //数组下标最大取值为3，现在取4，超出了范围，会产生索引越界异
常
        System.out.println(arr[4]);
    }
}
```

程序运行后，将会抛出`ArrayIndexOutOfBoundsException`数组越界异常。

在开发中，数组越界异常是不能出现的，一旦出现了，就必须要修改代码。

- 解决方案

将错误的索引修改为正确的索引范围即可！

4.2 空指针

- 出现原因

```
package com.briup.chap04;

public class Test042_NullPointer {
    public static void main(String[] args) {
        int[] arr = new int[3];

        //把null赋值给数组
        arr = null;
        System.out.println(arr[0]);
    }
}
```

`arr = null` 这行代码，意味着变量arr将不再保存数组的内存地址，我们通过arr这个标识符再也找不到堆空间数组元素，因此运行的时候会抛出 `NullPointerException` 空指针异常。

在开发中，空指针异常是不能出现的，一旦出现了，就必须要修改代码。

- 解决方案

  给数组一个真正的堆内存空间引用即可！

5.工具类

`java.util.Arrays`类，是JavaSE API中提供给我们使用的一个工具类，这个类中包含了操作数组的常用方法，比如排序、查询、复制、填充数据等，借助它我们在代码中操作数组会更方便。

Arrays中的常用方法：

- **toString方法**
  可以把一个数组变为对应的String形式
- **copyOf方法**

  可以把一个数组进行复制
  该方法中也是采用了arraycopy方法来实现的功能

- **sort方法**
  可以对数组进行排序
- **binarySearch方法**
  在数组中，查找指定的值，返回这个指定的值在数组中的下标，但是查找之前需要在数组中先进行排序，可以使用sort方法先进行排序
- copyOfRange方法（了解）

  也是复制数组的方法，但是可以指定从哪一个下标位置开始复制
  该方法中也是采用了arraycopy方法来实现的功能
- fill（了解）
  可以使用一个特定的值，把数组中的空间全都赋成这个值

案例展示：

```
package com.briup.chap04;

import java.util.Arrays;

public class Test05_Arrays {
    public static void main(String[] args) {
        int[] a = {1,3,5,2,6,8};

        //获取数组的字符串形式并输出
        System.out.println(Arrays.toString(a));

        //借助工具类完成数组拷贝
        a = Arrays.copyOf(a,10);
        //思考：数组的长度不能修改，此时a长度变成了10，如何解释？
        System.out.println(a.length);
        System.out.println(Arrays.toString(a));

        //对数组排序
        Arrays.sort(a);
        System.out.println(Arrays.toString(a));

        //二分查找
        int index = Arrays.binarySearch(a,5);
```

```
        System.out.println(index);

        //数组元素填充
        Arrays.fill(a,100);
        System.out.println(Arrays.toString(a));
    }
}
```

6.综合案例

案例1：求数组平均值

实现一个方法，参数是int[]，该方法可以计算出数组中所有数据的平均值并返回

```
package com.briup.chap04;

public class Test06_Case {
    //求数组的平均值
    public static double getAvg(int[] arr){
        int length = arr.length;

        double sum = 0;
        for(int i=0; i < length; i++){
            sum = sum + arr[i];
        }

        return sum/length;
    }

    public static void main(String[] args) {
        int[] array = new int[] {12,3,5,7,6};
        double avg = getAvg(array);
        System.out.println("平均值： " + avg);
    }
```

```
}
```

案例2：求数组最大值

实现一个方法，参数是int[]，该方法可以计算出数组中所有数据的最大值并返回

```
package com.briup.chap04;

public class Test06_Case {
    //省略...

    //求最大值
    public static int getMax(int[] arr){
        //max变量中的值，表示当前找到的最大值
        //假设数组中下标为0的位置是最大值
        //这步假设赋值的目的就是为了给局部变量max一个初始值
        int max = arr[0];

        for(int i=1; i < arr.length; i++){
            if(arr[i] > max){
                max = arr[i];
            }
        }
        return max;
    }

    public static void main(String[] args) {
        int[] array = new int[] {12,3,5,7,6};
        int max = getMax(array);
        System.out.println("最大值： " + max);
    }
}
```

案例3：数组反转

实现一个方法，参数是int[]，该方法可以将数组中所有元素进行反转，第一个和最后一个元素互换、第二个和倒数第二个互换...

```
package com.briup.chap04;

import java.util.Arrays;

public class Test06_Case {
    //省略...

    //反转方法
    //      1 2 3 4 5
    //      5 4 3 2 1
    //      1 2 3 4 5 6
    //      6 5 4 3 2 1
    public static void reverseArray(int[] arr) {
        int len = arr.length;
        for(int i = 0; i < len/2; i++) {
            //交换 arr[i] 和 arr[len-1-i]的值
            arr[i] = arr[i] ^ arr[len-1-i];
            arr[len-1-i] = arr[i] ^ arr[len-1-i];
            arr[i] = arr[i] ^ arr[len-1-i];
        }
    }

    public static void main(String[] args) {
        //1.准备数组
        int[] array = {8,6,9,1,4,12,7};

        //简单遍历写法：借助Arrays工具类实现
        System.out.println(Arrays.toString(array));

        //2.反转
        reverseArray(array);

        //3.遍历
        System.out.println(Arrays.toString(array));
    }
```

```
}
```

案例4：冒泡排序

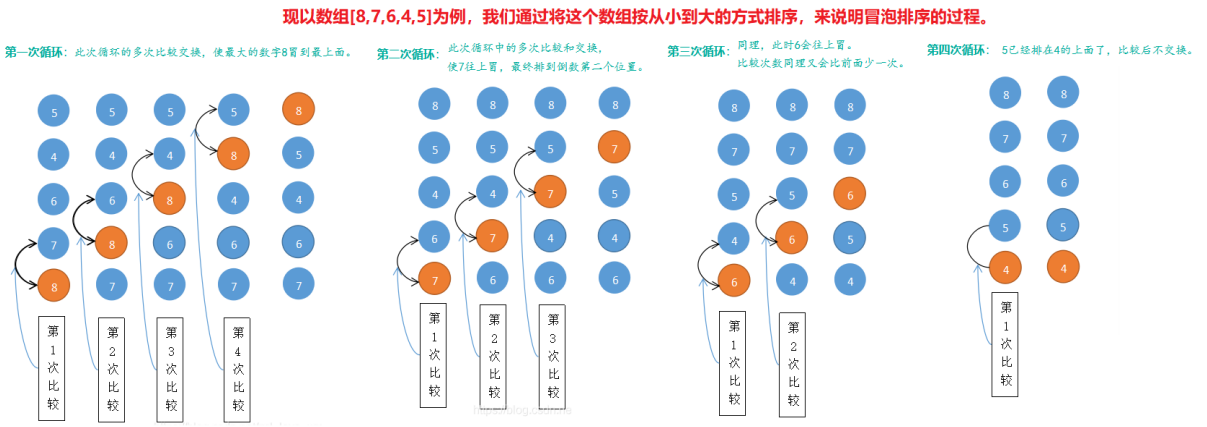
> 冒泡(Bubble Sort)排序是一种简单排序算法，它通过依次比较交换两个相邻元素实现功能。每一次冒泡会让至少一个元素移动到它应该在的位置上，这样 n 次冒泡就完成了 n 个数据的排序工作。

> 这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端（升序或降序排列），就如同碳酸饮料中二氧化碳的气泡最终会上浮到顶端一样，故名“冒泡排序”。

冒泡排序算法实现步骤：

1. 比较相邻的元素，如果第一个比第二个大，就交换他们两个。
2. 对每一对相邻元素重复上述工作，从第一对到最后一对。完成后，最大的数会放到最后位置。
3. 针对所有的元素重复以上的步骤，除了最后一个。
4. 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

冒泡排序过程具体见下图：




```
package com.briup.chap04;
```

```java
import java.util.Arrays;

public class Test06_Sort {
    public static void main(String[] args) {
        int[] array = {4,5,3,2,1};

        //排序
        bubbleSort(array);

        //遍历
        System.out.println(Arrays.toString(array));
    }

    //冒泡排序
    public static void bubbleSort(int[] arr) {
        //每次排序，可以将 未排序序列中 最大值 放到最后位置
        //len个成员，一共排序len-1次即可
        for(int i = 0; i < arr.length-1; i++) {
            //每次排序，借助交换 相邻两个数，实现  最大值移动到最后位置
            for(int j = 0; j < arr.length-1-i; j++) {
                if(arr[j] > arr[j+1]) {
                    arr[j] = arr[j] ^ arr[j+1];
                    arr[j+1] = arr[j] ^ arr[j+1];
                    arr[j] = arr[j] ^ arr[j+1];
                }
            }

            System.out.print("第"+(i+1)+"次排序后： ");
            System.out.println(Arrays.toString(arr));
        }
    }
}
```

案例5： 二分查找

在一个有序序列中查找其中某个元素，我们可以采用二分查找（折半查找），它的基本思想是：将n个元素分成个数大致相同的两半，取a[n/2]与欲查找的x作比较，如果x=a[n/2]则找到x，算法终止；如 果x<a[n/2]，则我们只要在数组a的左半部继续搜索x；如果x>a[n/2]，则我们只要在数组a的右半部继续搜索x。


```
package com.briup.chap04;

public class Test06_BinarySearch {
    //二分查找：针对有序序列进行 查找
    public static void main(String[] args) {
        int[] arr = {1,  3,  4,  5,  7,  9,  10};

        int index = binarySearch(arr, 1);
        System.out.println("1: " + index);

        index = binarySearch(arr, 2);
        System.out.println("2: " + index);

        index = binarySearch(arr, 10);
        System.out.println("10: " + index);
    }

    //二分查找算法,如果value存在arr中，则返回元素位置，如果找不到返回-1
    public static int binarySearch(int[] arr,int value) {
```

```
        int start = 0;
        int end = arr.length - 1;
        int mid;
        while(true) {
            mid = (start+end) / 2;

            if(value == arr[mid])
                return mid;
            else if(value > arr[mid]) {
                start = mid + 1;
            }else {
                end = mid - 1;
            }

            //当start > end 循环结束
            if(start > end)
                break;
        }

        return -1;
    }
}
```

7.扩展案例

已经完全掌握了冒泡排序和二分查找的同学，可以自己尝试学习选择、插入排序，掌握后最后学习希尔排序。不要求今天全部掌握，最近2-3天掌握即可！

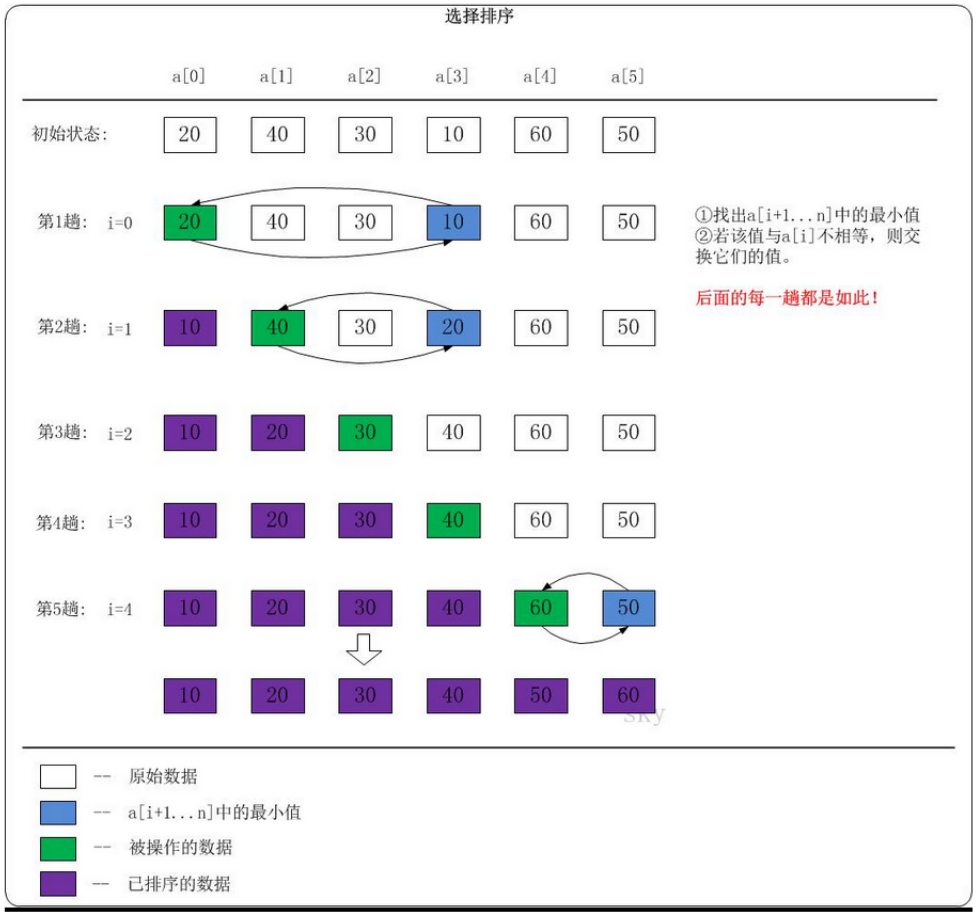
7.1 选择排序

> 选择排序(Selection Sort)的原理有点类似插入排序，也分已排序区间和未排序区间。但是选择排序每次会从未排序区间中找到最小的元素，将其放到已排序区间的末尾，最终完成排序。

选择排序算法描述：

1. 初始状态：无序区间为 Arr[0.1..n]，有序区间为空;
2. 第i==1趟排序开始，从无序区中选出最小的元素Arr[k]，将它与无序区的第1个元素交换，从而得到有序区间Arr[0..i-1],无序区间Arr[i..n];
3. 继续后面第i趟排序(i=2,3…n-1)，重复上面第二步过程;
4. 第n-1趟排序结束，数组排序完成。

选择排序过程如下图：



源码实现：

```
import java.util.Arrays;

public class Test07_SelectSort {
    public static void main(String[] args) {
        //准备一个int数组
        int[] array = {5, 2, 6, 5, 9, 0, 3};
```

```
        System.out.println("排序前： "+ Arrays.toString(array));

        //插入排序
        selectionSort(array);

        //输出排序结果
        System.out.println("排序后： "+ Arrays.toString(array));
    }

    public static void selectionSort(int[] arr) {
        int len = arr.length;
        if(len <= 1)
            return;

        //外层循环控制总体排序次数
        for(int i = 0; i < len-1; i++) {
            int minIndex = i;
            //内层循环找到当前无序列表中最小下标
            for(int j = i + 1; j < len; j++) {
                if(arr[minIndex] > arr[j]) {
                    minIndex = j;
                }
            }

            //将无需列表中最小值添加到 有序列表最后位置
            if(minIndex != i) {
                arr[minIndex] = arr[minIndex] ^ arr[i];
                arr[i] = arr[minIndex] ^ arr[i];
                arr[minIndex] = arr[minIndex] ^ arr[i];
            }

            //System.out.println(Arrays.toString(arr));
        }
    }
}
```

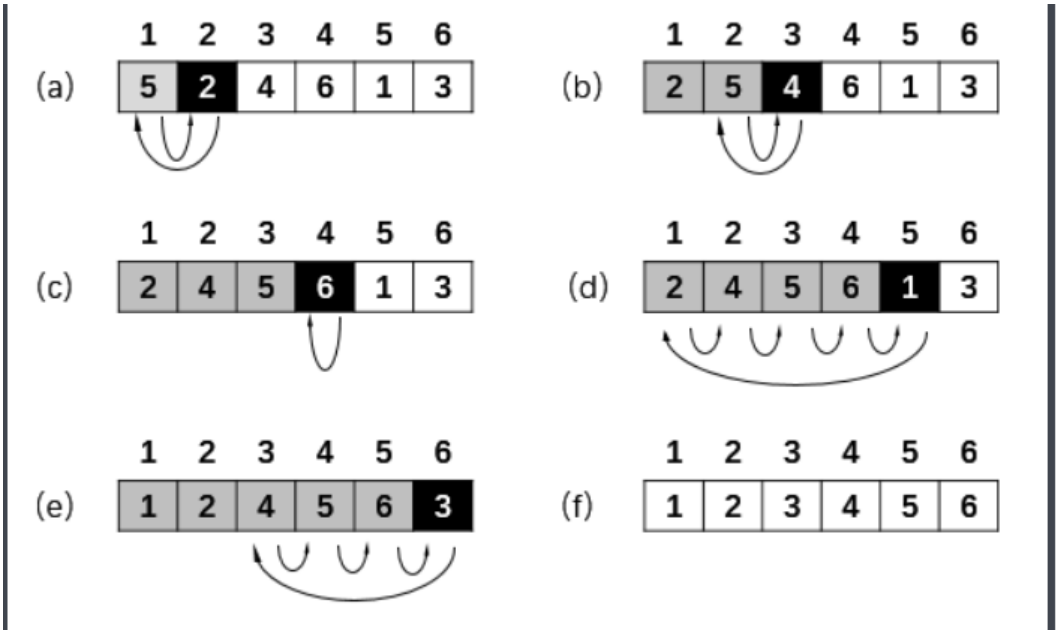
7.2 插入排序

> 插入排序(Insertion Sort)，一般也被称为直接插入排序。对于少量元素的排序，它是一个有效的算法。

插入排序算法描述：

1. 将数组分成两部分，已排序、未排序区间，初始情况下，已排序区间只有一个元素，即数组第一个元素;
2. 取未排序区间中第一个元素，插入到已排序区间中合适的位置，这样子就得到了一个更大的已排序区间;
3. 重复这个过程，直到未排序区间中元素为空，算法结束。

插入排序过程见下图：



源码实现：

```
import java.util.Arrays;

public class Test07_InsertSort {
    public static void main(String[] args) {
        //准备一个int数组
        int[] array = {5, 2, 6, 5, 9, 0, 3};
```

```
        System.out.println("排序前： "+ Arrays.toString(array));

        //插入排序
        insertionSort(array);

        //输出排序结果
        System.out.println("排序后： "+ Arrays.toString(array));
    }

    public static void insertionSort(int[] arr) {
        int len = arr.length;
        if(len <= 1) {
            return;
        }

        //外层循环控制 总体循环次数
        for(int i = 1; i < len; i++) {
            //内层循环做的事情：将无序列表中第一个元素插入到有序列表中合
适位置
            int value = arr[i];
            //获取有序列表中最后一个元素下标
            int j = i - 1;
            for(; j >= 0; j--) {
                if(value < arr[j]) {
                    arr[j+1] = arr[j];
                }else {
                    break;
                }
            }

            //将需要插入的元素 放置到合适位置
            arr[j+1] = value;

            //一次排序完成后，输出 方便 观察
            System.out.println(Arrays.toString(arr));
        }
    }
```

```
}
```

7.3 希尔排序

> 希尔(shell)排序是Donald Shell于1959年提出的一种排序算法。希尔排序也是一种插入排序，它是简单插入排序经过改进之后的一个更高效的版本，也称为缩小增量排序，同时该算法是冲破O(n2）的第一批算法之一。

希尔排序对直接插入排序改进的着眼点：

- 若待排序序列中**元素基本有序**时，直接插入排序的效率可以大大提高
- 如果待排序序列中**元素数量较小**时，直接插入排序效率很高

希尔排序算法思路：

将整个待排序序列分割成**若干个子序列**，在子序列内部分别进行**直接插入排序**，等到整个序列**基本有序**时，再对全体成员进行直接插入排序！

待解决问题：

- 如何分割子序列，才能保证最终能得到基本有序?
- 子序列内部如何进行直接插入排序?

有序?

基本有序：接近正序，例如{1, 2, 8, 4, 5, 6, 7, 3, 9}；
局部有序：部分有序，例如{6, 7, 8, 9, 1, 2, 3, 4, 5}。
局部有序不能提高直接插入排序算法的时间性能。

启示?

子序列的构成不能是简单地“逐段分割”，而是将相距某个“增量”的记录组成一个子序列。

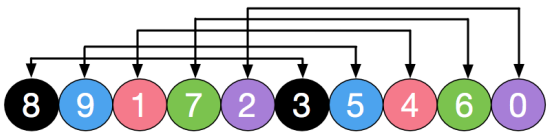
分割方案

1. 将有n个元素的数组分成n/2个数字序列，第i个元素和第i+n/2，i+n/2*m...个元素为一组；
2. 对一组数列进行简单插入排序；
3. 然后，调整增量为n/4,从而得到新的几组数列，再次排序；
4. 不断重复上述过程，直到增量为1，shell排序完全转化成简单插入排序，完成该趟排序，则数组排序成功。

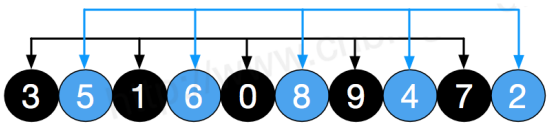
希尔排序流程：

原始数组 以下数据元素颜色相同为一组

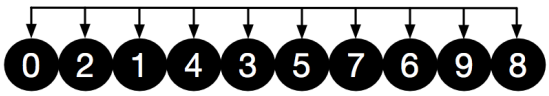
初始增量 gap=length/2=5，意味着整个数组被分为5组，[8,3] [9,5] [1,4] [7,6] [2,0]



对这5组分别进行直接插入排序，结果如下，可以看到，像3，5，6这些小元素都被调到前面了，
然后缩小增量 gap=5/2=2,数组被分为2组 [3,1,0,9,7] [5,6,8,4,2]



对以上2组再分别进行直接插入排序，结果如下，可以看到，此时整个数组的有序程度更进一步啦。
再缩小增量gap=2/2=1，此时，整个数组为1组[0,2,1,4,3,5,7,6,9,8]，如下



经过上面的“宏观调控”，整个数组的有序化程度成果喜人。
此时，仅仅需要对以上数列简单微调，无需大量移动操作即可完成整个数组的排序。

具体源码实现：

```
import java.util.Arrays;

public class Test07_ShellSort {
```

```
    public static void main(String[] args) {
        //准备一个int数组
        int[] array = {5, 2, 6, 5, 9, 0, 3};

        System.out.println("排序前: "+ Arrays.toString(array));

        //shell排序
        shellSort(array);

        //输出排序结果
        System.out.println("排序后: "+ Arrays.toString(array));
    }

    //由简单插入排序 改造得到 shell排序
    public static void shellSort(int[] array) {

        int len = array.length;
        if(len <= 1)
            return;

        //设置初始增量
        int gap = len / 2;

        //由增量控制整体排序次数
        while(gap > 0) {

            //插入排序改造
            for(int i = gap; i < len; i++) {
                //记录要插入的值
                int value = array[i];
                //有序序列的最后一个元素下标
                int j = i - gap;
                for(; j >= 0; j -= gap) {
                    if(value < array[j]) {
                        array[j + gap] = array[j];
                    }else {
                        break;
                    }
```

```
                }

                array[j+gap] = value;
            }

            System.out.println(Arrays.toString(array));

            gap = gap / 2;
        }
    }
}
```

8.数组拷贝

> 数组的长度确定后便不能修改，如果需要数组存放更多元素，可以通过创建长度更长的新数组，然后先复制老数组内容到新数组中，再往新数组中放入额外的元素。

在`java.lang.System`类中提供一个名为`arraycopy`的方法可以实现复制数组中元素的功能

```
//该方法的声明
public static void arraycopy(Object src,
                             int srcPos,
                             Object dest,
                             int destPos,
                             int length)
//参数1,需要被复制的目标数组
//参数2,从目标数组的哪一个位置开始复制
//参数3,需要把数据复制到另外一个新的数组中
//参数4,把数据复制到新数组的时候，需要把数据从什么位置开始复制进去
//参数5,复制的目标数组的长度
```

案例展示:

定义一个方法，传递一个数组对象给它，其将数组长度扩大到原来的2倍并返回。

```
package com.briup.chap04;

import java.util.Arrays;

public class Test08_ArrayCopy {
    public static int[] dilatation(int[] arr) {
        //额外准备一个数组，容量为arr的两倍，用来存储arr拷贝过来的元素
        int[] b = new int[arr.length*2];

        //将arr数组拷贝到b数组中，从0开始放
        System.arraycopy(arr, 0, b, 0, arr.length);

        return b;
    }

    public static void main(String[] args) {
        //准备一个数组，并进行初始化
        int[] arr = {1,3,5,7,9};

        int[] newArr = dilatation(arr);

        //输出newArr数组内容
        System.out.println(Arrays.toString(newArr));
        //结果为：[1, 3, 5, 7, 9, 0, 0, 0, 0, 0]
    }
}
```

9.二维数组

> 如果把普通的数组（一维数组），看作一个小盒子的话，盒子里面可以存放很多数据，那么二维数组就是像一个大点的盒子，里面可以存放很多小盒子（一维数组）。

9.1 定义格式

二维数组固定定义格式有2种，具体如下：

格式1：

```
数据类型[][] 数组名 = new 数据类型[一维长度m][二维长度n];
```

m：表示二维数组的元素数量，即可以存放多少个一维数组

n：表示每一个一维数组，可以存放多少个元素

格式2：

```
数据类型[][] 数组名 = new 数据类型[一维长度][];
```

案例展示：

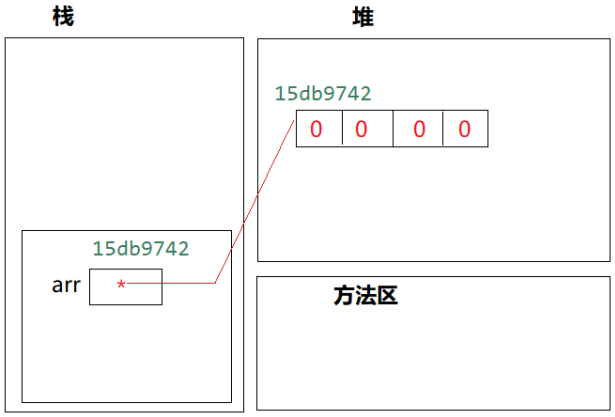
```
package com.briup.chap04;

import java.util.Arrays;

public class Test091_Basic {
    public static void main(String[] args) {
        //一维长度2，代表这个二维数组里面包含2个元素，每个元素都是一个一维数组
        //二维长度3，代表这个二维数组中元素，类型都是int[3]的一维数组，存放3个int数据
        int[][] arr = new int[2][3];
        /*
            [[I@15db9742

            [[:          2个中括号就代表的是2维数组
```

```
            I:              数组中存储的数据类型为int
            15db9742:       十六进制内存地址
        */
        System.out.println(arr);

        //二维数组的每个元素值(第一维), 对应的是一维数组的内存地址值
        System.out.println(arr[0]); //[I@15db9742
        System.out.println(arr[1]); //[I@6d06d69c

        System.out.println("--------------");

        //第二种定义格式
        int[][] arr2 = new int[2][];

        //输出arr2中2个元素值，默认为null、null
        System.out.println(Arrays.toString(arr2));

        //给二维数组的每个元素赋值
        //arr[0] = new int[2];
        //arr[1] = new int[3];
    }
}
```

9.1 内存结构

一维数组内存结构:

```java
public static void main(String[] args) {
    // 定义并初始化数组arr
    int[] arr = new int[4];

    // 输出结果为：[I@15db9742
    System.out.println(arr);
}
```



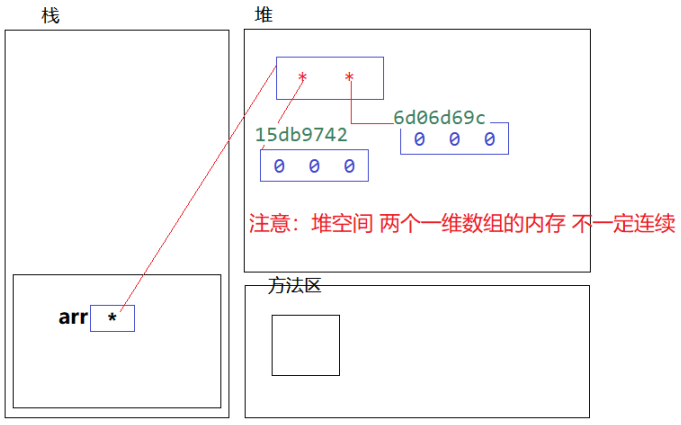
数组名arr 标识的那块内存(栈空间)，存放了一个引用值(地址值)，通过该地址值可以找到堆空间相应内存（用来存放数组中所有元素）。

二维数组内存结构：

可以把二维数组看成一个一维数组，数组的每个元素对应的内存区域中，存放的是一维数组引用值，具体可参考下面2个图：

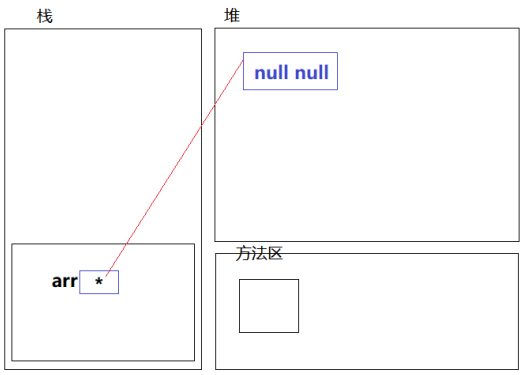
二维数组第 1 种定义格式，内存构成

```java
int[][] arr = new int[2][3];
System.out.println(arr[0]); //[I@15db9742
System.out.println(arr[1]); //[I@6d06d69c
```

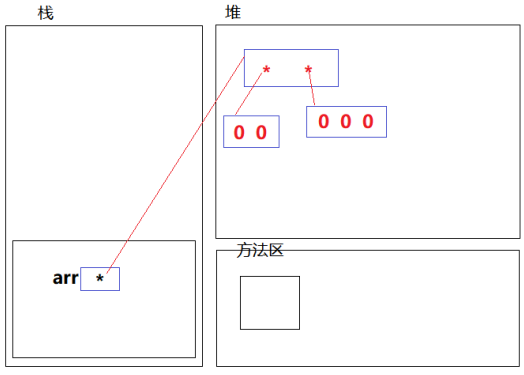


二维数组第2种定义格式，内存构成

```java
int[][] arr = new int[2][];
```




```java
int[][] arr = new int[2][];
arr[0] = new int[2];
arr[1] = new int[3];
```



9.3 元素访问

二维数组中元素的访问和赋值，也是通过数组下标实现的。

书写格式：

`二维数组名[一维下标m][二维下标n];`

例如：`int arr[2][3];`

注意：m、n的取值都是`[0，length-1]`，注意不要越界，否则会出现异常`ArrayIndexOutOfBoundsException`。

案例展示：

```
package com.briup.chap04;

public class Test093_Access {
    public static void main(String[] args) {
        // 数据类型[][] 变量名 = new 数据类型[m][n];
        int[][] arr = new int[2][3];

        //获取二维数组元素值并输出
        System.out.println(arr[0][0]);
        System.out.println(arr[1][1]);

        System.out.println("-----------");

        // 向二维数组中存储元素
        arr[0][0] = 11;
        arr[0][1] = 22;
        arr[0][2] = 33;

        arr[1][0] = 11;
```

```
        arr[1][1] = 22;
        arr[1][2] = 33;

        //arr[2][0] = 3;  数组越界异常

        // 3. 遍历二维数组，获取所有元素，累加求和
        for (int i = 0; i < arr.length; i++) {
            for(int j = 0; j < arr[i].length; j++){
                System.out.print(arr[i][j] + " ");
            }

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

9.4 初始化

二维数组的静态初始化，有点类似一维数组的初始化，具体格式如下：

完整格式：

`数据类型[][] 数组名 = new 数据类型[][]{ {元素1, 元素2...} , {元素1, 元素2...};`

简化格式：

`数据类型[][] 数组名 = { {元素1, 元素2...} , {元素1, 元素2...} ...};`

案例：

```
package com.briup.chap04;

public class Test094_Init {
    //封装二维数组遍历方法
    public static void outArray(int[][] arr) {
```

```
        // 遍历二维数组，获取所有元素，累加求和
        for (int i = 0; i < arr.length; i++) {
            for(int j = 0; j < arr[i].length; j++){
                System.out.print(arr[i][j] + " ");
            }

            System.out.println();
        }
    }

    public static void main(String[] args) {
        //第一种：完整格式
        int[][] arr1 = new int[][]{{1,2,3},{4,5}};
        outArray(arr1);

        System.out.println("----------");

        //第二种：简化格式
        int[][] arr2 = {{11, 22, 33}, {44, 55}};
        outArray(arr2);

        System.out.println("----------");

        //第三种：建议从内存角度理解
        int[] arr3 = {11, 33};
        int[] arr4 = {44, 55, 66};
        int[][] array = {arr3, arr4};
        outArray(array);
    }
}
```

9.5 综合案例

案例1：二维数组元素遍历

```java
//上面案例已经使用
public static void outArray(int[][] arr) {
    // 遍历二维数组，获取所有元素，累加求和
    for (int i = 0; i < arr.length; i++) {
        for(int j = 0; j < arr[i].length; j++){
            System.out.print(arr[i][j] + " ");
        }

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

案例2：二维数组元素求和

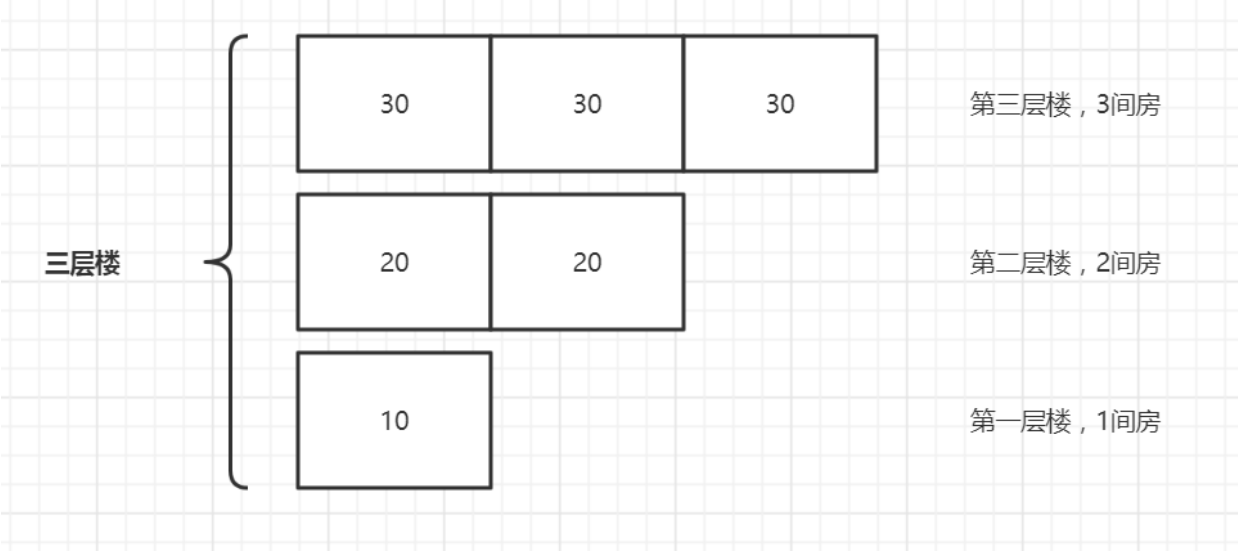
```java
package com.briup.chap04;

public class Test095_Sum {
    public static void main(String[] args) {
        // 1. 定义求和变量，准备记录最终累加结果
        int sum = 0;

        // 2. 使用二维数组来存储数据
        int[][] arr = new int[3][];
        arr[0] = new int[]{10};
        arr[1] = new int[]{20,20};
        arr[2] = new int[]{30,30,30};
        //思考，下面一行代码是否正确
        //arr[2] = {30,30,30};

        // 3. 遍历二维数组，获取所有元素，累加求和
        for (int i = 0; i < arr.length; i++) {
            for(int j = 0; j < arr[i].length; j++){
                sum += arr[i][j];
            }
        }
```

```
        // 4. 输出最终结果
        System.out.println(sum);
    }
}
```

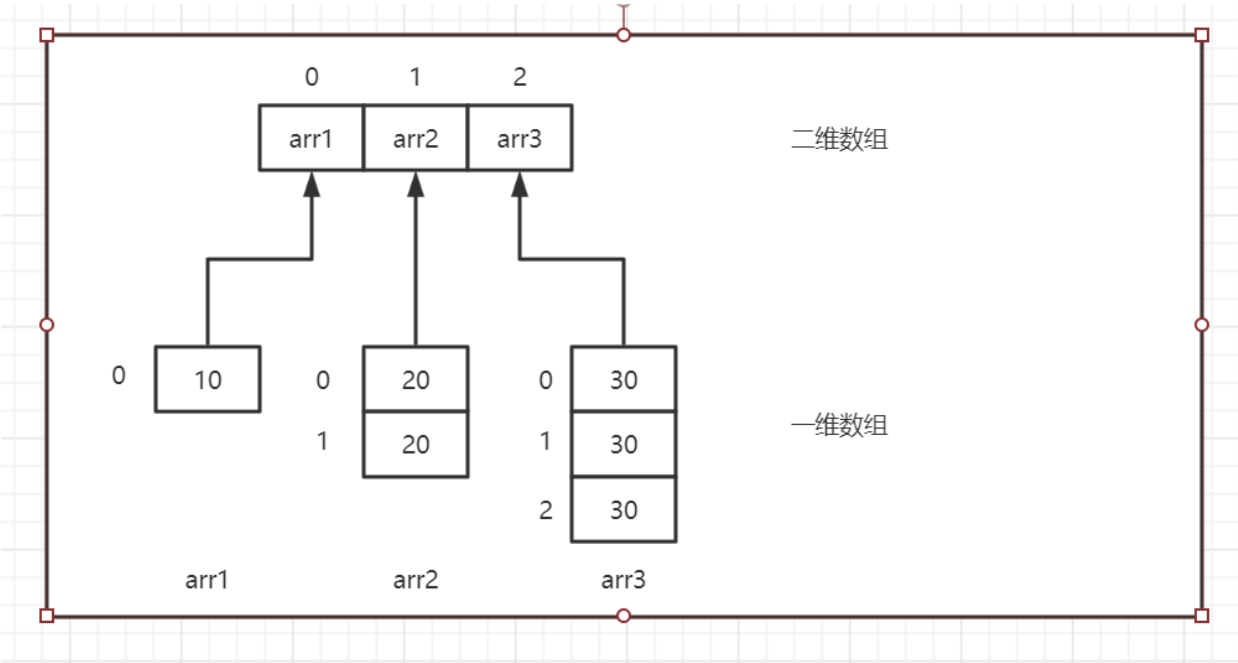
注意事项：

可以把上述案例中二维数组理解为一栋大厦，共有3层楼，每层楼有多个房间，每个房间可以存放一个int数据，现在每个房间的默认值都是0。

可以想象为以下情况：



在JVM内存中的情况如下：



案例3：创建杨辉三角并输出

使用二维数组，打印输出杨辉三角，效果如下：

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

具体实现：

```
package com.briup.chap04;

public class Test095_Triangle {
    public static void main(String[] args) {
        //1.定义2维数组
        int[][] arr = new int[5][5];

        //2.给二维数组赋值，得到杨辉三角
        for(int i = 0; i < arr.length; i++) {
            for(int j = 0; j < arr[i].length; j++) {
                //a.j==0  第一列都为 1
```

```
                if(j == 0)
                    arr[i][j] = 1;
                else if(i == j) {
                //b. 行和列相同  设置1
                    arr[i][j] = 1;
                }else if(i > j) {
                //c. 当前位置值 == 上一行相同列 + 上一行前一列
                    arr[i][j] = arr[i-1][j] + arr[i-1][j-1];
                }
            }
        }

        //3.遍历二维数组，输出杨辉三角
        for(int i = 0; i < arr.length; i++) {
            for(int j = 0; j < arr[i].length; j++) {
                //只输出 左下部分，右上部分0值不输出
                if(i >= j)
                    System.out.print(arr[i][j] + "\t");
            }
            System.out.println();
        }
    }
}
```

扩展功能：按下图形式输出杨辉三角

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

代码实现：

```
package com.briup.chap04;
```

```java
public class Test095_Triangle2 {
    public static void main(String[] args) {
        //line，表示需要输出的行数
        int line = 5;
        int[][] arr = new int[line][];

        //根据规律，构造出二维数组并且赋值
        for(int i = 0; i < arr.length; i++){

            arr[i] = new int[i+1];
            //循环给二维数组中的每个位置赋值
            for(int j = 0; j < arr[i].length; j++){
                if(j == 0 || j == arr[i].length - 1){
                    arr[i][j] = 1;
                }
                else{
                    //除了下标中的0和最后一个，其他的元素都具备相同的规
律
                    //这个位置的值=上一层和它相同下标的值+前一个元素的
值
                    arr[i][j] = arr[i-1][j] + arr[i-1][j-1];
                }
            }
        }

        //把赋值完成的二维数组按要求进行输出
        for(int i = 0; i < arr.length; i++){
            //控制每行开始输出的空格
            for(int j = 0;j < (arr.length - i -1); j++){
                System.out.print(" ");
            }
            //控制输出二维数组中的值，记得值和值之间有空格隔开
            for(int k = 0; k < arr[i].length; k++){
                System.out.print(arr[i][k]+" ");
            }
            //当前行输出完，再单独输出一个换行
            System.out.println();
        }
```

```
    }
}
```

10.可变参数列表

> JDK1.5或者以上版本中，可以使用可变参数列表

格式:

```
修饰符 返回值类型 方法名(数据类型... 参数名) {
    方法体语句;
}
```

使用:

```
//普通方法定义
public static void fun(int[] a){
    //...
}

//可变参数列表 方法定义
public static void test(int... a){
    //...
}

public static void main(String[] args) {
    int[] arr = {1,2,3};

    //普通方法的调用，只有下面一种形式
    fun(arr);

    //可变参数列表方法的调用，下面形式都可以
    test();          //不传参
    test(1);         //传递1个元素
    test(1,2,3,4);   //传递多个元素
```

```
    test(arr);      //传递数组
}
```

> **结论:**
>
> 可变参数列表本质上是一个数组，方法中使用可变参数列表，比用数组作参数功能更强大。

案例展示:

```
package com.briup.chap04;

public class Test10_Variable {
    public static void main(String[] args) {
        int[] array = {1,3,2,4,6};

        //1.调用普通方法实现功能
        double avg = getAvg(array);
        System.out.println(avg);
        System.out.println("------------");

        //2.使用可变参数列表
        int sum = getSum(array);
        System.out.println(sum);

        System.out.println("------------");

        //3.可变参数列表特殊形式
        sum = getSum(1,2,3,4);  //依旧能够成功
        System.out.println(sum);
    }

    //普通方法
    public static double getAvg(int[] arr) {
        double sum = 0;
```

```
        for(int i = 0; i < arr.length; i++) {
            sum += arr[i];
        }

        return sum/arr.length;
    }

    //使用可变参数列表
    public static int getSum(int... arr) {
        //【可变参数列表本质上是数组】
        // [I@15db9742
        System.out.println(arr);
        // 5
        System.out.println(arr.length);
        int sum = 0;

        for(int i = 0; i < arr.length; i++) {
            sum += arr[i];
        }

        return sum;
    }
}
```

补充：方法中有一个可变参数同时，还可以存在其他参数

```
//普通参数len和可变参数arr共存
public static int getSum2(int len, int... arr){
    int sum = 0;
    for(int i = 0; i < len; i++) {
        sum += arr[i];
    }
    return sum;
}
```

注意事项：可变参数和普通参数共存的时候，可变参数必须放到最后一个参数的位置

总结：

> 1. 可变参数列表 可以接受 0-n个参数
> 2. 可变参数列表还可以接受数组
> 3. 可变参数列表必须放在函数参数列表的最右端，且只能出现1次