# 第四章 数组

## 4.1 内存中的数组

当数组的引用变量指向有效的内存后，才可以通过该数组变量来访问数组元素。实际的数组元素被存储在堆（Heap）内存中，引用变量则被存储在栈（Stack）内存中。

一个方法执行时，每个方法都会建立自己的内存栈，在这个方法内定义的变量将会被逐个存入这块内存栈里面，当方法执行结束时，这个方法的内存栈也就自然被销毁了。当我们在内存中创建一个对象时，这个对象将被保存在堆内存，以便于反复利用，这是因为对象创建成本通常较大，这样对象不会随着方法的结束而销毁，只有当一个对象没有任何引用变量引用它的时候，系统的垃圾回收机制才会在合适的时候回收它。

当执行 int[] iArr; 后，只是在栈内存中定义了一个空引用，这个引用并未指向任何有效的内存，因此也就无法指定数组的长度，当执行  “iArr = new int[5];”  实现动态初始化后，系统将负责为该数组分配内存空间，并分配默认的初始值，所有数组元素都被默认赋为0。

## 4.2 数组常用排序

冒泡排序

*/\* 从小到大排序 \*/  
/\* 0 > 1 ? 互换 ： 不变 \*/  
/\* 1 > 2 ? 互换 ： 不变 \*/  
/\* 每次内循环结束都会在末尾产生一个判断后最大的数 \*/  
/\* 所以每次内循环都要减去i \*/*

**public static void** bubbleSort(**int**[] arr){  
 *// 9个数据元素的话需要比较8次才能出结果（外循环），所以需要arr.length - 1* **for**(**int** i = 0; i < arr.**length** - 1; i++){  
 *// 每次内循环需要减去 i ， 因为每次内循环之后都会生成最大的数在最后面，这个最大的数不需要再比较* **for**(**int** j = 0; j < arr.**length** - 1 - i; j ++){  
 **if**(arr[j] > arr[j + 1]){  
 **int** tempV = arr[j];  
 arr[j] = arr[j + 1];  
 arr[j + 1] = tempV;  
 }  
 }  
 }  
}

*// 增强型冒泡排序 有问题***public static void** bubbleSortPlus(Integer[] arr) {  
 **int** tempV = 0;  
 *// 用于记录位置是否改变*  
 **for**(**int** i = 0; i < arr.**length** - 1; i ++){

**boolean** isChange = **false**;  
 **for**(**int** j = 0; j < arr.**length** - i - 1; j ++){  
 **if**(arr[j+1] < arr[j]){  
 tempV = arr[j+1];  
 arr[j+1] = arr[j];  
 arr[j] = tempV;  
 *// 位置发生改变* isChange = **true**;  
 }  
 }  
 *// 当位置不发生改变，说明已排好序* **if**(!isChange) **break**;  
 }  
}

选择排序

*// 增强版选择排序***public static** Integer[] selectionSort(Integer[] arr){  
 **for**(**int** i = 0; i < arr.**length** - 1; i ++){  
 **int** minIndex = i;  
 **for**(**int** k = i + 1; k < arr.**length**; k ++){  
 **if**(arr[k] < arr[minIndex]){  
 minIndex = k; *//记录最小的数组下标 ， 内循环完再交换位置。* }  
 }  
 **int** tempV = arr[i];  
 arr[i] = arr[minIndex];  
 arr[minIndex] = tempV;  
 }  
 **return** arr;  
}

插入排序

**public static void** insertionSort(Integer[] array) {  
 **int** current;  
 **for** (**int** i = 0; i < array.**length** - 1; i++) {  
 current = array[i + 1];  
 **int** preIndex = i;  
 **while** (preIndex >= 0 && current < array[preIndex]) {  
 array[preIndex + 1] = array[preIndex];  
 preIndex--;  
 }  
 array[preIndex + 1] = current;  
 }  
}

## 4.3数组的优缺点

优点：随机访问的效率很高。根据下标访问元素效率高。

缺点：数组大小一经确定不能改变

数据进行增删操作的时候，效率低，因为增删是要改变数组大小的，而数组的特性是不能改变大小，所以增删的时候只能创建新的数组并且把原有的数组内容移动到新的数组，然后销毁原有的数组，这样效率低下。

数组元素的类型只能是一种

数组通过内容查找元素的效率比较低，假如要查找的元素在最后一个，这样只能一个一个元素去找，因为数组是连续的。

数组的元素是连续分配的，所以在堆内存中必须找到连续的内存空间才能容纳数组的所有数据，对内存有比较高的要求。

## 4.4 数组的特殊性

多维数组可以使用Arrays.deepToSting()将多维数组转换为String。

不能使用泛型创建数组，如果允许创建泛型数组，将能在数组p里存放任何类的对象，并且能够通过编译，因为在编译阶段p被认为是一个Object[ ]，也就是p里面可以放一个int，也可以放一个Pair，当我们取出里面的int，并强制转换为Pair，调用它的info（）时会怎样？java.lang.ClassCastException！这就违反了泛型引入的原则。所以，Java不允许创建泛型数组。解决方法是使用Object转型。

Object[] objs = new Object[size];

T[] Objs = (T[])Objs;

## 4.5 Arrays实用功能

### 4.5.1常用函数

* 1. Arrays.equals()用于比较两个数组是否相等（deepEquals用于多维数组）。
  2. Arrays.fill()用于填充数组。
  3. Arrays.sort()用于对数组排序。
  4. Arrays.binarySearch()用于在已经排序的数据中查找元素（二分法）。
  5. Arrays.toString()产生数组的String表示。
  6. Arrays.hashCode()产生数组的散列码。
  7. Arrays.asList()转变成list容器。

### 4.5.2复制数组

System.arraycopy()使用此函数复制数组比for循环快很多。

格式System.arraycopy(src, beginIndex1, desc, beginIndex2, length);

src:原数组

beginIndex1:原数组开始复制的位置

desc:目标数组

beginIndex2:目标数组开始被赋值的位置

length：复制的长度