# Day05 Java基础

## 二维数组

### 二维数组的定义：

相比起普通数组（一维数组），普通数组中每个位置存 储的都是一个元素，二维数组每个位置存储的都是一个普通数组。所以二维数 组是用来存储数组的数组

### 二维数组的定义格式：

数据类型[][] 数组名 = new 数据类型[二维数组的长度][包含的一维数组 的长度];

int[][] arr = new int[3][5]; --- 定义包含3个一维的整型数组，每 个一维数组能存储5个整数。

数据类型[][] 数组名 = new 数据类型[二维数组的长度][];

int[][] arr = new int[3][]; --- 定义了包含3个一维的整型数组，没 有规定每一个一维数组的大小

arr[0] = new int[3];

arr[1] = new int[7];

arr[2] = new int[4];

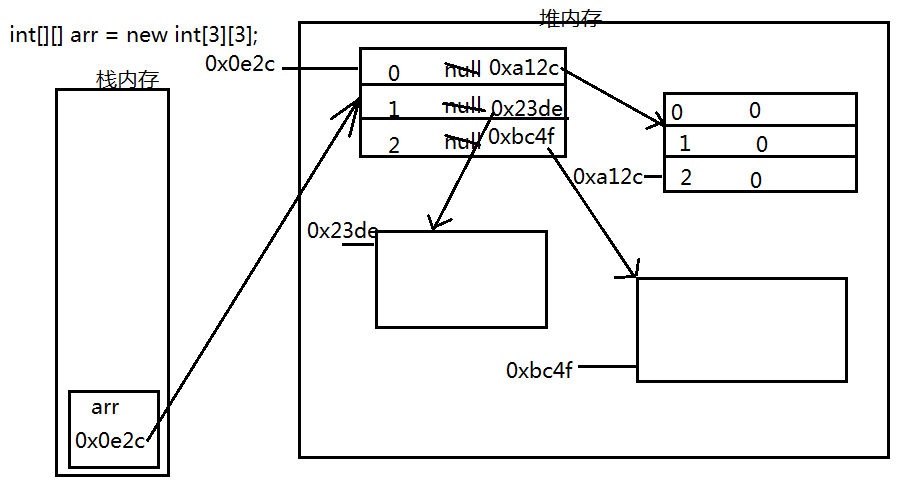
数据类型[][] 数组名 = {{一维数组1},{一维数组2},{一维数组 3}, ... ,{一维数组n}};

int[][] arr = {{2,4}, {1,2,3,5,7}, {8}, {8,4,3,0}}; --包含了4 个一维数组

System.out.println(arr[1][2]); --- 3

System.out.println(arr[1]);---打印对应的一维数组的地址

### 二维数组在内存中的存储



如果没有给二维数组中对应的一维数组定义大小，那么针对这个一维数组操作 会出现空指针异常---NullPointerException

注意：int[] a,b[]; 根据昨天笔记中提到的，可以判断出a是一个一维数 组，b是一个二维数组。

## 方法定义

### 方法的定义：

将一段代码/逻辑提取出来进行包装，这种包装形式就称之 为方法/函数。

在我们实际开发中，有些代码是需要重复的利用到的，但是如果我们每次都重 写一遍的话，是非常没有效率了，所以我们将这些常用的代码写成一个方法， 这样每次需要使用之前曾经使用过的方法的时候，只需调用方法就可以实现， 大大提升了代码的复用性。

### 定义格式：

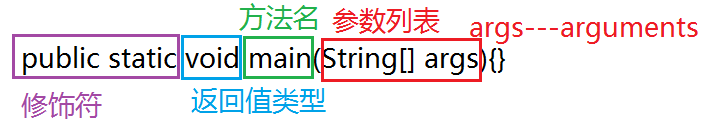
修饰符 返回值类型 方法名(参数列表){

代码块;

return 返回值;

}

以main方法为例：



在我们定义一个方法的时候，首先要确定他的可用范围即权限修饰符，然后要 判断是这个方法是否要返回一个值，如果不需要的话，则在返回值类型出用 void。如果需要返回值，则写相应的返回值的类型，然后，我们需要判断是 否需要给方法传入参数，以及传入几个参数。

### 代码详解

求2个整数的和

明确方法的返回值类型---求2个整数的和---结果一定是一个int类型--- 返回值类型是int

明确方法是否需---这2个整数对于当前方法而言是2个未知量，需 要以参 数的形式体现

public static int add(int i, int j){

int sum = i + j;

// 将结果返回给用方法的人

return sum;

}

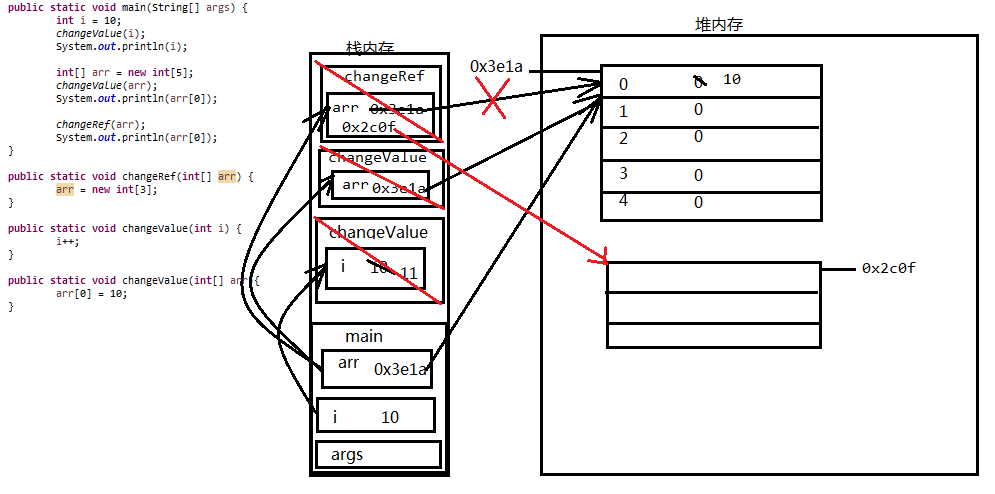
## 方法重载

### 重载的概念：

在同一个类中，包含两个方法名一致，而参数列表不一样的 方法，这就叫做方法的重载

### 方法的重载，重点关注的是方法的参数列表

### 关于方法的传值：



在方法传参的时候，基本类型传递的是实际的数据，所以新方法中数据发生改 变不影响原方法中的数据；数组传递的是地址，所以在新方法中如果不改变地 址，则会影响原来的方法中的数组；如果在新方法中改变了地址，则不会影响 原来方法中的数组

## 方法递归

### 递归的定义：

在我们一些代码逻辑中，会存在一中需要调用方法本身的情 况，这种方法自己调用自己的形式，就叫方法的递归。

### 代码实现：

求一个数的前n项和。

public static int sum(int n){

if(n == 1)

return 1;

// 在方法中调用了自己本身 --- 方法的递归

return n + sum(n - 1);

}

注意：因为方法是在栈内存中执行的，所以不断的调用方法本身，很容易产生 栈溢出异常，循环的效率要高于递归---所以能使用循环的而时候减少使用递 归