day08 面向对象-上

1、方法相关练习

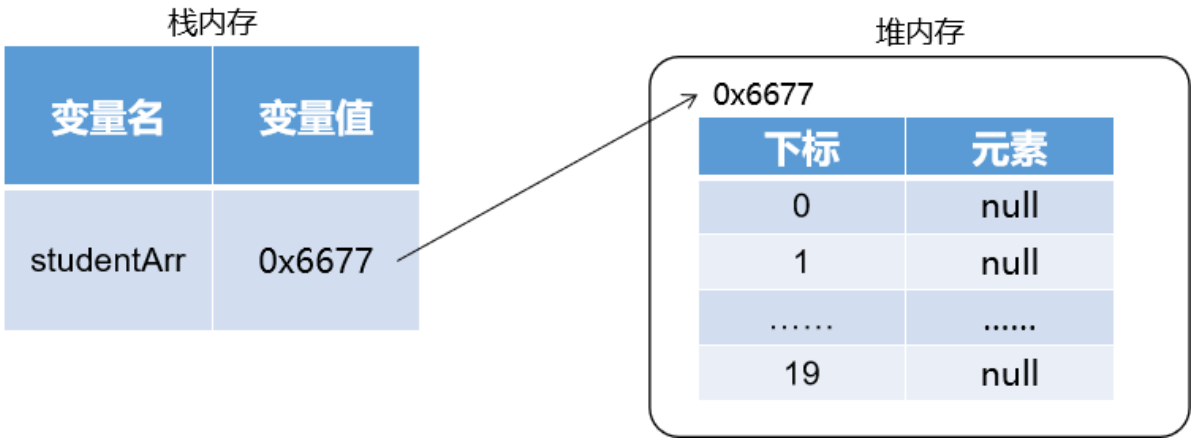
> 定义类Student，包含三个属性：学号number(int)，年级state(int)，成绩score(int)。 创建20个学生对象，学号为1到20，年级和成绩都由随机数确定。
> 问题一：打印出3年级(state值为3）的学生信息。
> 问题二：使用冒泡排序按学生成绩排序，并遍历所有学生信息
>
> 提示：
> 1) 生成随机数：Math.random()，返回值类型double;
> 2) 四舍五入取整：Math.round(double d)，返回值类型long。

①声明Student类

```
public class Student {

    int number;
    int state;
    int score;

    // 用于每次打印Student对象的信息，代码复用，不必每次想打印时都再写一遍
    public void showMyInfo() {
        System.out.println("number=" + number + "\tstate=" + state + "\tscore="
+ score);
    }

}
```

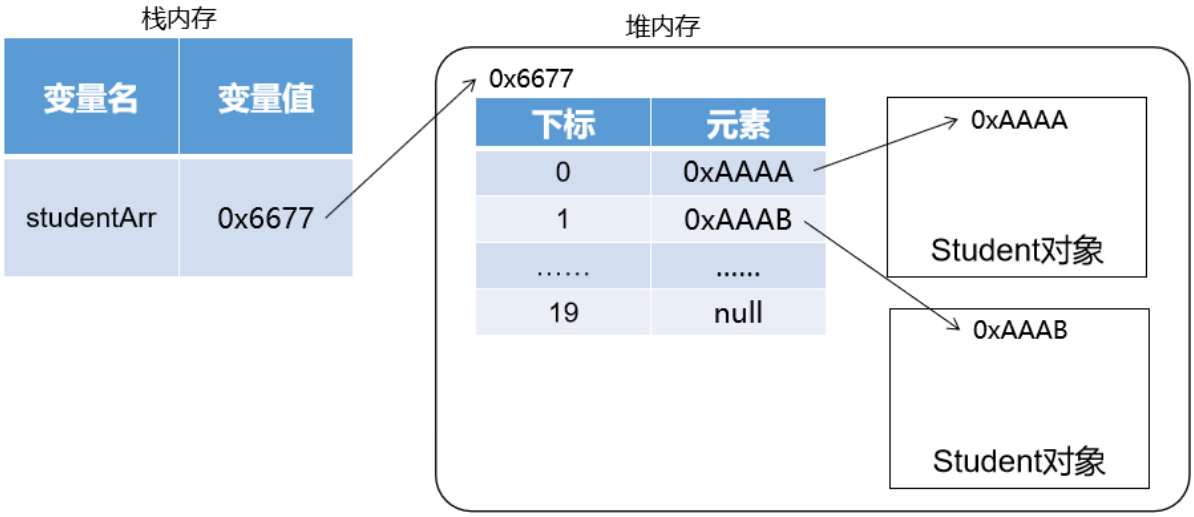
②创建Student数组

```
Student[] studentArr = new Student[20];
```



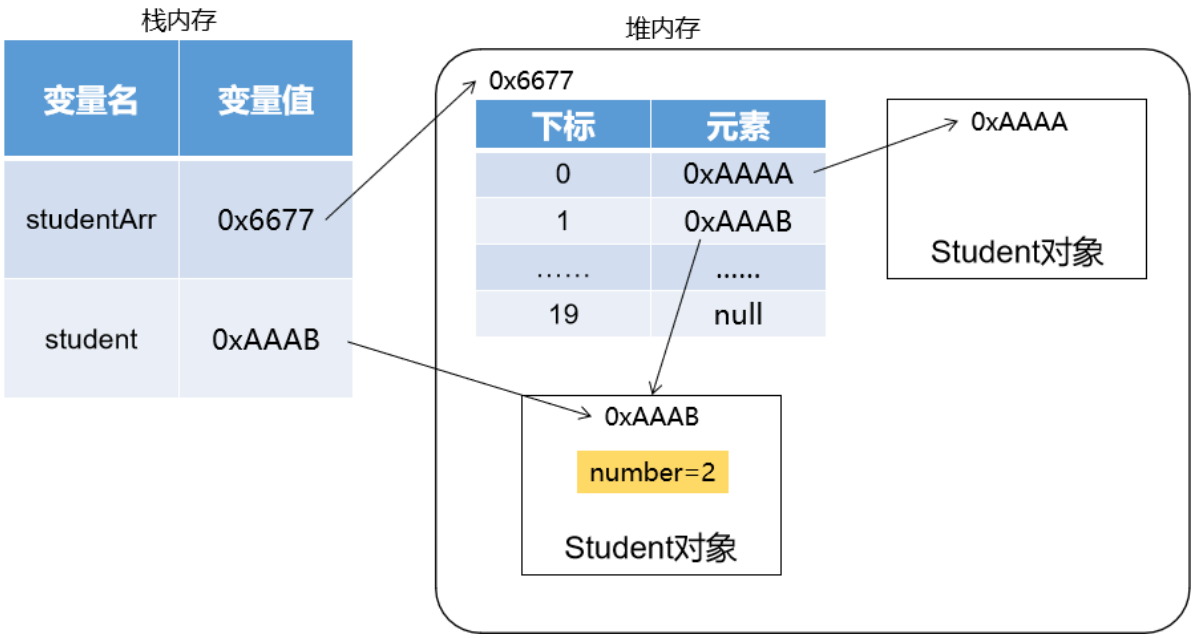
③循环创建Student对象并存入数组

```
// 循环20次创建20个Student对象并存入Student数组
for (int i = 0; i < 20; i++) {
    Student student = new Student();
    studentArr[i] = student;
}
```



④在循环创建Student对象时设置属性

```
// 循环20次创建20个Student对象并存入Student数组
for (int i = 0; i < 20; i++) {

    // 创建Student对象
    Student student = new Student();

    // 给Student对象设置number属性
    student.number = i + 1;

    // 给Student对象设置state属性
    student.state = (int) (Math.random() * 10) + 1;

    // 给Student对象设置score属性
    student.score = (int) (Math.random() * 100);

    // 将Student对象存入数组
    studentArr[i] = student;
}
```

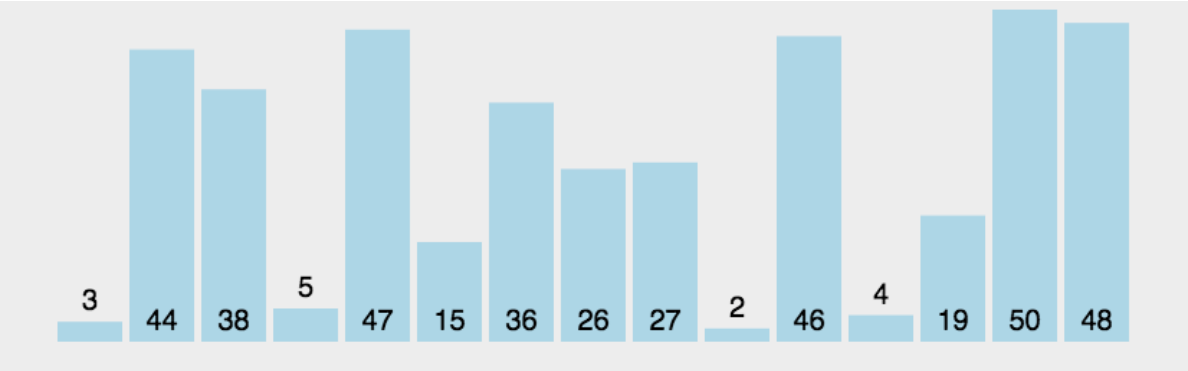


⑤打印年级为3的Student

```
// 循环20次创建20个Student对象并存入Student数组
for (int i = 0; i < 20; i++) {

    // 创建Student对象
    Student student = new Student();

    // 给Student对象设置number属性
    student.number = i + 1;

    // 给Student对象设置state属性
    student.state = (int) (Math.random() * 10) + 1;

    // 给Student对象设置score属性
    student.score = (int) (Math.random() * 100);

    // 将Student对象存入数组
    studentArr[i] = student;

    // 判断当前Student对象的state属性是否为3
    if (student.state == 3) {
        student.showMyInfo();
    }
}
```

⑥根据成绩冒泡排序

[1]冒泡排序算法



- 外层循环：i（0~length-1）
  - 数组的长度是多少，外层就循环多少次
- 内层循环：j（0~j-i-1）
  - 执行范围
    - 当前元素：0~数组长度-i-1（减 i 是因为外层每执行一次，最后一个元素就不用考虑了；减1是为了避免j+1造成数组下标越界）
    - 下一个元素：1~数组长度-i
  - 拿当前元素和下一个元素进行比较
  - 如果当前元素比下一个元素要大，那就两个元素交换位置
  - 内层循环执行完能够把最大的元素移动到数组的末尾

[2]代码

```
// 对数组中的元素进行冒泡排序
// 外层循环：控制『冒泡』次数
for (int i = 0; i < 20; i++) {

    // 内层循环：负责执行『冒泡』
    // 每冒泡一次就是把当前范围最大的元素移动到最后
    // 减 i 是因为已经排序好的元素不用动
    // 减 1 是因为避免j+1找下一个元素时数组下标越界
    for (int j = 0; j < (20 - i - 1) ; j++) {

        // 将『当前学生的分数』和『下一个学生的分数』进行比较
        if (studentArr[j].score > studentArr[j+1].score) {

            // 交换：需要借助中间变量
            Student swap = studentArr[j];

            studentArr[j] = studentArr[j+1];

            studentArr[j+1] = swap;

        }

    }

}
```

⑦再次遍历排序后的数组

```
// 遍历Student数组
for (int i = 0; i < studentArr.length; i++) {

    Student student = studentArr[i];

    student.showMyInfo();

}
```

⑧有可能出现的问题

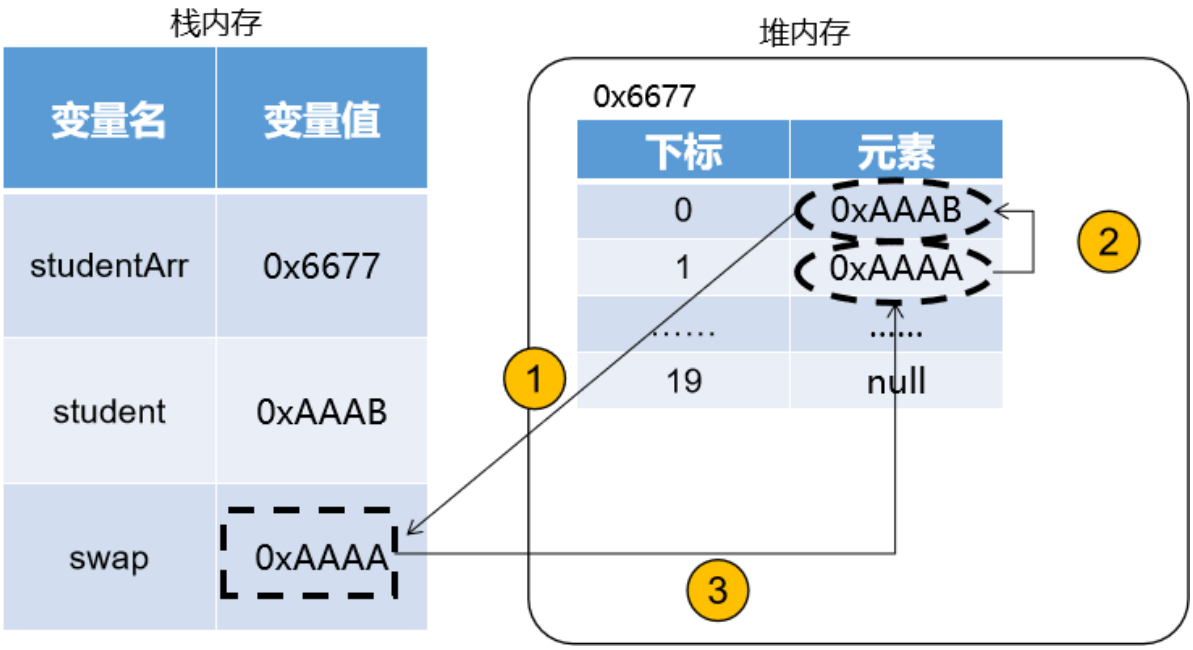
[1]仅使用几个变量来执行交换

代码按下面这样写不会真正实现交换。因为数组中对应位置的元素没有被触及。

```
Student stuCurrent = studentArr[j];
Student stuNext = studentArr[j+1];

// 将『当前学生的分数』和『下一个学生的分数』进行比较
if (studentArr[j].score > studentArr[j+1].score) {

    // 交换：需要借助中间变量
    Student swap = stuCurrent;

    stuCurrent = stuNext;

    stuNext = swap;

}
```

下面是正确代码：

```
// 交换：需要借助中间变量
Student swap = studentArr[j];

studentArr[j] = studentArr[j+1];

studentArr[j+1] = swap;
```



[2]内层循环的终点没有-1

错误代码：

```
for (int j = 0; j < (20 - i) ; j++) {

    // 将『当前学生的分数』和『下一个学生的分数』进行比较
    if (studentArr[j].score > studentArr[j+1].score) {

        // 交换：需要借助中间变量
        Student swap = studentArr[j];

        studentArr[j] = studentArr[j+1];

        studentArr[j+1] = swap;

    }

}
```

抛出的异常：

> Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 20
> at com.atguigu.object.exer.StudentTest.main(StudentTest.java:48)

异常的原因：

studentArr[j+1].score中 j 是19，j+1是20，导致数组下标越界。

⑨StudentTest全部代码

```
package com.atguigu.object.exer;

public class StudentTest {

    public static void main(String[] args) {

        // 声明一个Student类型的数组，来存放循环创建的20个Student对象
```

```
        Student[] studentArr = new Student[20];

        // 循环20次创建20个Student对象并存入Student数组
        for (int i = 0; i < 20; i++) {

            // 创建Student对象
            Student student = new Student();

            // 给Student对象设置number属性
            student.number = i + 1;

            // 给Student对象设置state属性
            student.state = (int) (Math.random() * 10) + 1;

            // 给Student对象设置score属性
            student.score = (int) (Math.random() * 100);

            // 将Student对象存入数组
            studentArr[i] = student;

            // 判断当前Student对象的state属性是否为3
//              if (student.state == 3) {
//                  student.showMyInfo();
//              }

            // 开发过程中，为了看到全部数据的情况，逐个打印
            student.showMyInfo();
        }

        // 对数组中的元素进行冒泡排序
        // 外层循环：控制『冒泡』次数
        for (int i = 0; i < 20; i++) {

            // 内层循环：负责执行『冒泡』
            // 每冒泡一次就是把当前范围最大的元素移动到最后
            // 减 i 是因为已经排序好的元素不用动
            // 减 1 是因为避免j+1找下一个元素时数组下标越界
            for (int j = 0; j < (20 - i - 1) ; j++) {

                // 将『当前学生的分数』和『下一个学生的分数』进行比较
                if (studentArr[j].score > studentArr[j+1].score) {

                    // 交换：需要借助中间变量
                    Student swap = studentArr[j];

                    studentArr[j] = studentArr[j+1];

                    studentArr[j+1] = swap;

                }

            }

        }

        // 打印分割线
        System.out.println("==================================");
```

```
        // 遍历Student数组
        for (int i = 0; i < studentArr.length; i++) {

            Student student = studentArr[i];

            student.showMyInfo();

        }

    }

}
```

2、方法重载

①需求

计数器类中已有方法：做两个int类型的加法

```
public int add(int a, int b)
```

想要增加新的方法：做两个double类型的加法

```
public double add(double a, double b)
```

为了满足更多使用情况，还想有更多方法：

```
public int add(int a, int b, int c)
```

小结：在一个类中，很可能会有很多类似的需求，为了满足这些需求，我们会声明很多相似的方法。同时为了让方法的调用者体验更好、更容易找到所需方法，这些功能相近的方法最好使用『同一个方法名』。

②前提

- 同一个类中
- 同名的方法

③方法重载的好处

- 没有重载不方便：让方法调用者，在调用方法的时候，不必为了相似的功能而查阅文档，查找各种不同的方法名，降低学习成本，提高开发效率。
- 有了重置很方便：在调用一系列重载的方法时，感觉上就像是在调用同一个方法。对使用者来说，只需要知道一个方法名就能够应对各种不同情况。

④规则限制

限制的来源：本质上只要让系统能够区分清楚我们具体要调用哪一个方法。

- 在同一个类中，如果两个方法的方法名一致，那么参数列表必须不同。
- 参数列表区分
    - 要么是参数个数不同
    - 要么是参数类型不同

⑤重载方法举例

[1]参数个数不同

```
public int add(int a, int b)
public int add(int a, int b, int c)
```

[2]参数类型不同

```
public int add(int a, int b)
public double add(double a, double b)
```

或:

```
public double add(int a, double b)
public double add(double a, int b)
```

3、方法可变参数

①需求

在实际开发过程中，确实有一些情况不确定在调用方法时传入几个参数。所以为了让调用方法时能够弹性传参，JavaSE5.0标准增加了可变参数功能。

②声明和调用

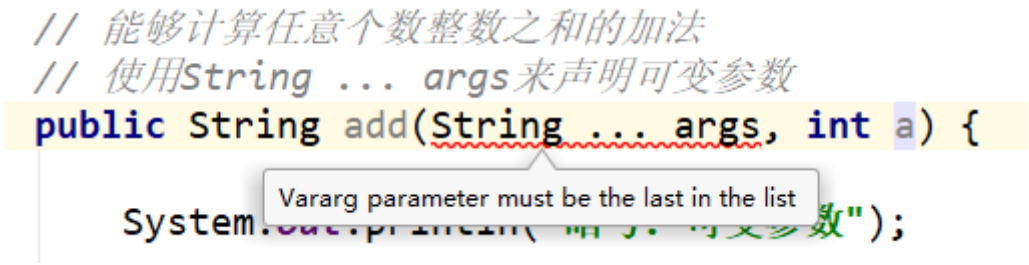
```
// 能够计算任意个数整数之和的加法
// 使用String ... args来声明可变参数
public String add(String ... args) {

    System.out.println("暗号：可变参数");

    String sum = "";

    // 在方法体内，可变参数按照数组形式来处理
    for (int i = 0; i < args.length; i++) {
        sum = sum + args[i];
    }

    return sum;
}
```

测试代码:

```
String addResultStr = calculator.add("a");
System.out.println("addResultStr = " + addResultStr);

addResultStr = calculator.add("a", "b");
System.out.println("addResultStr = " + addResultStr);

addResultStr = calculator.add("a", "b", "c");
System.out.println("addResultStr = " + addResultStr);

addResultStr = calculator.add();
System.out.println("addResultStr = " + addResultStr);
```

③语法规则

- 可变参数必须在整个参数列表的最后



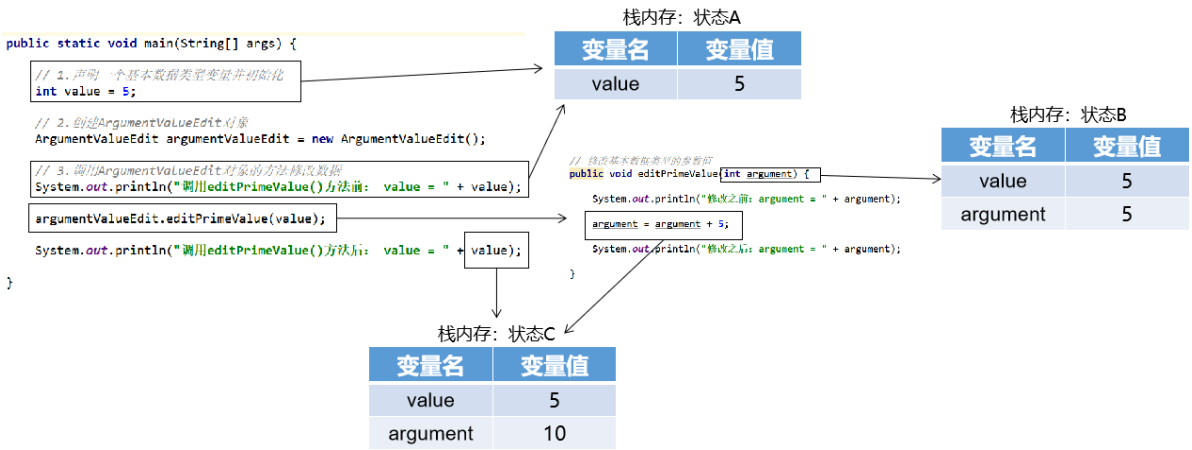
- 当调用方法时实际传入的参数既匹配可变参数列表方法，又匹配一个具体的参数列表方法，那么系统会优先调用具体的参数列表方法

> 举例：调用方法add(5, 3)
>
> 可变参数方法：add(int ... args)
>
> 具体参数方法：add(int i, int j)【系统会调用这个方法】

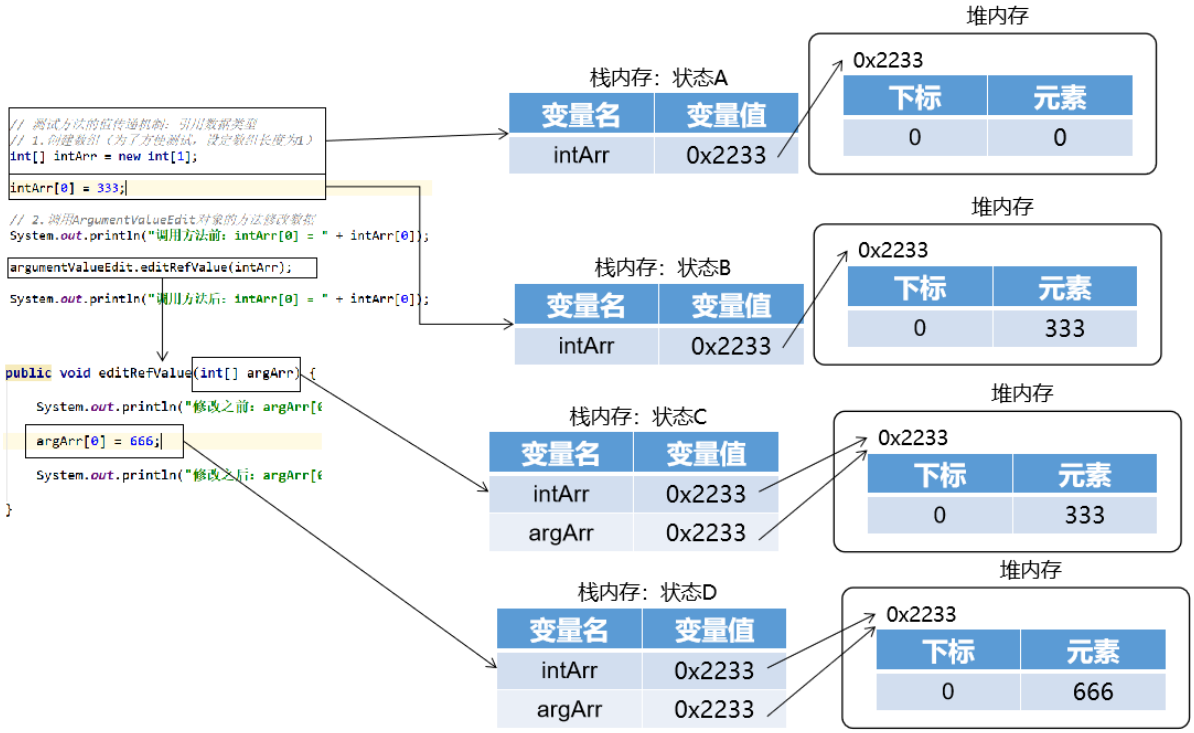
- 一个方法只能声明一个可变参数

4、方法参数值传递

①基本数据类型



②引用数据类型


// 测试方法的值传递机制：引用数据类型
// 1.创建数组（为了方便测试，设定数组长度为1）
int[] intArr = new int[1];
intArr[0] = 333;
// 2.调用ArgumentValueEdit对象的方法修改数组
System.out.println("调用方法前：intArr[0] = " + intArr[0]);
argumentValueEdit.editRefValue(intArr);
System.out.println("调用方法后：intArr[0] = " + intArr[0]);
public void editRefValue(int[] argArr) {
System.out.println("修改之前：argArr[
argArr[0] = 666;
System.out.println("修改之后：argArr[
}
栈内存：状态A
变量名
变量值
intArr
0x2233
堆内存
0x2233
下标
元素
0
0
栈内存：状态B
变量名
变量值
intArr
0x2233
堆内存
0x2233
下标
元素
0
333
栈内存：状态C
变量名
变量值
intArr
0x2233
argArr
0x2233
堆内存
0x2233
下标
元素
0
333
栈内存：状态D
变量名
变量值
intArr
0x2233
argArr
0x2233
堆内存
0x2233
下标
元素
0
666