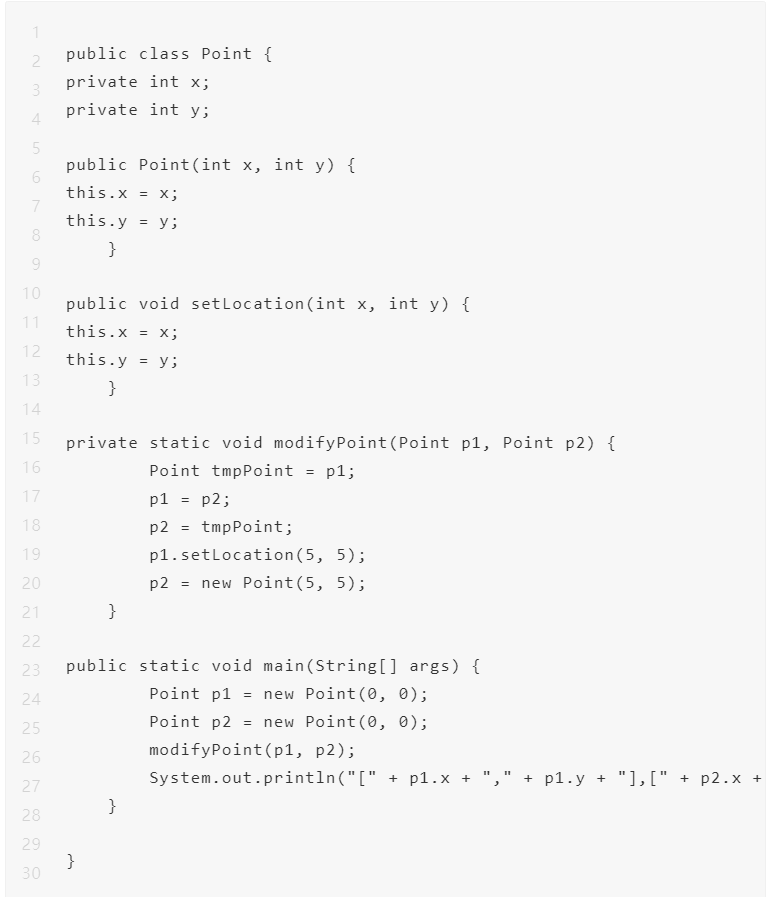
https://mp.weixin.qq.com/s/KkIpLkIMNjnIT0K3A9-0Jg

# 详细解析面试题：Java的值传递 ！

先从一道面试题说起：

以下程序输出结果是什么：



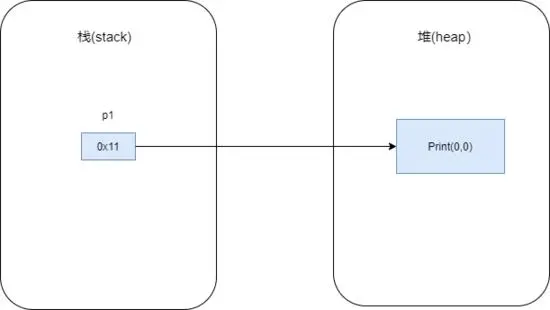
答案

[0,0],[5,5]

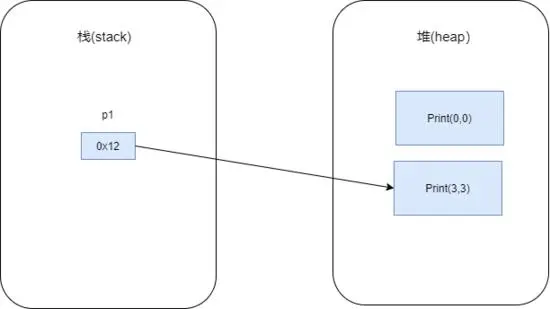
## 1、值与引用

为了纠正值传递和引用传递的一些误解，此处探讨的并不是值类型和引用类型，而是赋值操作时对各部分的名称。

以上面面试题为例，有 Point p1 = new Point(0, 0);



变量p1里存储着实际对象的地址，一般称这种变量为"引用"，引用指向实际对象，我们称实际对象为该引用的值； 赋值操作符=实际上做的就是将引用指向值的地址的工作 ，如果我们有 p1 = new Point(3,3); 的话，情形就是这样：



我们要注意到，在堆中的对象 Print(0,0) 并没有发生改变，改变的只有引用 p1 指向的地址。

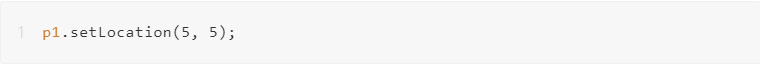
值与引用就是以上这些， **而值传递和引用传递，和这些一点关系也没有** 。

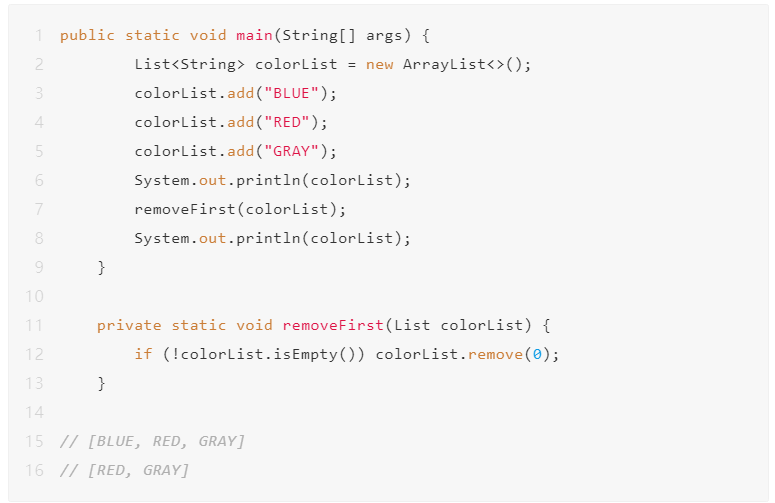
## 2、值传递和引用传递

我们可以找到网上资料对值传递的定义，大多是这样的:

在方法调用时，传入方法内部的是实参引用的拷贝，因此对形参的任何操作都不会影响到实参。

这句话本身对值传递的定义是比较准确的，但由于前后的矛盾和概念的盲区，让我们理解起来产生了歧义，概况起来歧义有三：



很多时候我们对值传递有疑惑，**本质原因是**没有弄明白值传递和引用传递到底是在描述什么，我们错以为它们的名字是自解释的：

传递的是值就是值传递，传递的是引用就是引用传递，Java是值传递意味着java在方法调用时传递的是值。

这就是我们上面说的 **概念盲区** ，对值传递和引用转递的正确解释其实是这样的：

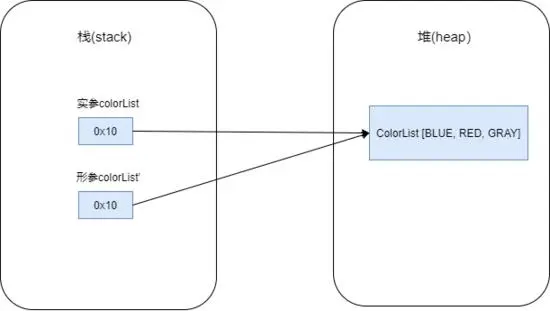
值传递（Call by value）和引用传递（Call by reference），描述的是函数调用时参数的求值策略（ **Evaluation strategy** ），是对调用函数时，求值和取值方式的描述，而非传递的内容。

值与引用描述了两种内存分配方式，值在堆上分配，引用在栈上分配（在这里要区分值类型和引用类型）。而值传递和引用传递描述的则是参数求值策略， **两者之间不存在任何依赖关系** 。

综上，我们可以得出Java是值传递这种说法也是不准确的，完整的说法应该是 **java在函数调用时采用的求值策略为值传递。** 严格意义上讲，求值策略也不仅仅有值传递和引用传递，还有 **共享对象传递（Call by sharing）** 和 **值-返回传递（Call by copy-restore）** ，但这些概念对于我们理解Java值传递并无益处，如果感兴趣可以点击链接参见详细。

理清了值传递和引用传递所描述的具体内容，我们现在来看关于Java值传递概念的第三点歧义：形参的任何操作都不会影响到实参。

以上述 colorList 代码为例，我们已经知道值传递这种求值策略在函数调用时会把实参引用的拷贝传入函数内，所以在调用 removeFirst() 方法时有：



形参colorList'作为实参colorList的拷贝，它们寄存着一样的地址，都指向堆中的同一个实例 ColorList [BLUE, RED, GRAY] 。

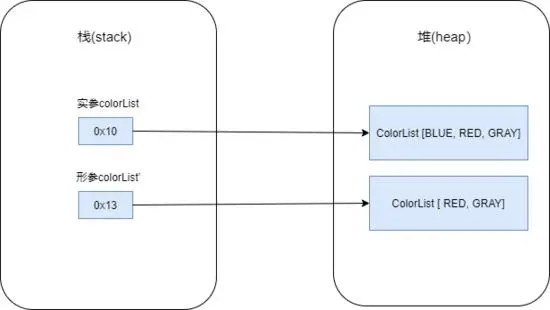
我们在 removeFirst() 方法内操作的是形参colorList'。但是刚刚说到，无论是形参还是实参，它们都只会存储实例对象的地址，真正的对象仍是堆中它们共同指向的 ColorList 。而我们在 removeFirst() 方法内调用的 remove() ， **是实例对象 ColorList 本身提供的可以改变自身的函数**。

所以，当 removeFirst() 方法内执行 colorList'.remove(0) 后，形参colorList' 所指向的实例对象 ColorList 就变成了 ColorList[RED, GRAY] ，而实参colorList仍然指向了这个实例对象，所以当方法调用完成后，实例对象改变了。

这样看来，关于“形参的任何操作都不会影响到实参”确实是不严谨的，那么这句话在什么情况下生效呢，我们将 removeFirst() 方法改成如下代码：



在这个新的 removeFirst() 方法中有：



按照值传递的求值策略规定，传入实参引用的拷贝这一步没变。

在这个 removeFirst() 方法中我们将形参 colorList' 的引用做了重赋值操作，所以现在它指向了一个新的实例 ColorList [ RED, GRAY] ，但是随着 removeFirst() 方法执行完毕后退出，形参colorList'也会被回收，而实参colorList指向的 ColorList [BLUE, RED, GRAY] 并没有发生改变（变的是形参指向的实例，但是在方法退出后，方法参数 colorList' 就会被回收，而它指向的实例也就会变成垃圾对象）。

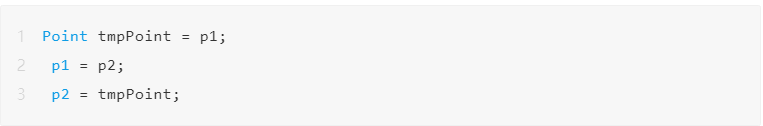
关于值传递概念的三点歧义就解释完毕了，我们在这里可以给出java值传递一个正确完整的概念了：

java 使用的是一种名为值传递的求值策略，这种策略在传值过程中会复制实参的引用，并将这份拷贝传入到方法形参中， **所以对形参的任何重赋值操作都不会对实参产生影响** 。

在下一小节中我们会解析开篇的面试题，如果你对Java值传递仍然理解不好，我会在下一节中分享一个技巧帮助你更彻底的弄明白Java值传递。

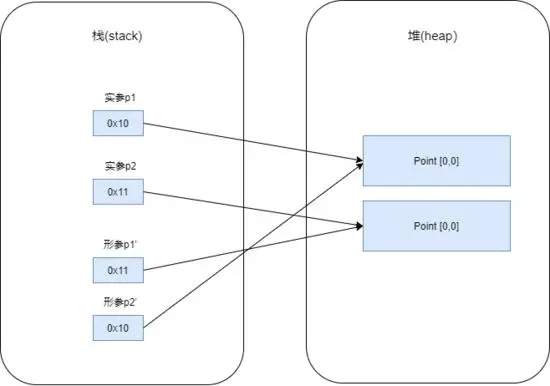
## 3、面试题解析

回到开篇的面试题，在 modifyPoint() 方法中，头三行代码使用了一个临时变量 tmpPoint 互换了形参 p1 和 p2 的值，所以在方法内部，形参 p1 实际上指向了实参 p2 ，形参 p2 指向了实参 p1 ：



关于Java的值传递策略，有一种比较取巧的理解，就是完全可以把函数的传参过程等同于赋值操作符=来理解，我们在调用modifyPoint(p1,p2)时，可以理解在方法内部有形参p1'=实参p1，形参p2'=实参p2，这样我们在方法里操作的p1'和p2'实际只是个临时变量，它的生命周期仅限于方法里。

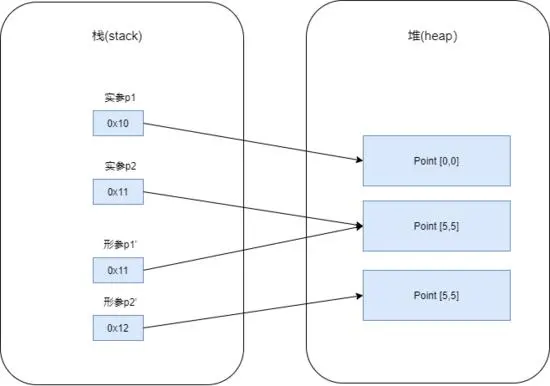
在形参 p1' 与形参 p2' 互换后，堆栈信息简易表示为：



然后调用了 p1'.setLocation(5, 5); 我们在上节说过，如果实例对象本身提供了改变自身的方法，那么在形参调用该方法后也会改变实参的，因为它们都指向了同一个实例，所以这时实参 p2 也变为 Point[5.5] 。

代码走到下一行 p2' = new Point(5, 5); 基础整篇文章的论述，这一行我们可以直接跳过不管了，因为 **形参的重赋值操作不会影响到实参** 。

最后的堆栈信息如下：



所以最终答案显而易见：[0,0],[5,5]