JAVA CORE

Java – pass by reference or pass by value

BY HUSSEINTEREK · JULY 15, 2017

Before describing how arguments are passed in java, it is worth to define how java variables are allocated inside the memory. Basically we talk about 2 types of variables: **primitives** and **objects.**

Primitive variables are always stored inside the stack memory (*the memory space which holds method specific variables that are short-lived, in addition to references to other objects in the heap*) , however in case of objects, they are stored at 2 stages, the actual object data

FOLLOW:

NEXT STORY
Working with hashcode() and equals() in java

PREVIOUS STORY
Java – Read files from classpath

RECOMMENDED

J2EE
Build REST web service using Spring
3 NOV, 2017

JAVA CORE
Java – pass by reference or pass by value
15 JUL, 2017

JAVA CORE
How to create an immutable class in java
21 JUL, 2017

SPRING BOOT

is stored inside the heap memory (*the memory space which holds objects and JRE classes*) and a reference for the object is kept inside stack memory which just points to the actual object.

1. By value VS By reference

What is meant by ***"By value"*** and ***"By reference"***:

- **By value:** when arguments are passed by value to a method, it means that a copy of the original variable is being sent to the method and not the original one, so any changes applied inside the method

Deploy Spring Boot application on external Tomcat
24 NOV, 2017

J2EE

Pass data from servlet to jsp
6 OCT, 2017

LIKE US ON FACEBOOK

are actually affecting the copy version.

- **By reference:** When arguments are passed by reference, it means that a reference or a pointer to the original variable is being passed to the method and not the original variable data.

2. How arguments are passed in java?

In java, arguments are always passed by **value** regardless of the original variable type. Each time a method is invoked, the following happens:

- A copy for each argument is created in the stack memory and the copy version is passed to the method.
  - If the original variable type is primitive, then simply, a copy of the variable is created inside the stack memory and then passed to the method.
  - If the original type is not primitive, then a new reference or pointer is created inside the stack memory which points to the actual object data and the new reference is then passed to the method, (at this

stage, 2 references are pointing to the same object data).

3. Fixing some concerns !!

In the following example, we try to validate that *"java is always pass by value"* through passing several argument types *(primitive, wrappers, collections, business objects)* and checking whether they are modified after the method call.

- **Passing primitive arguments:**

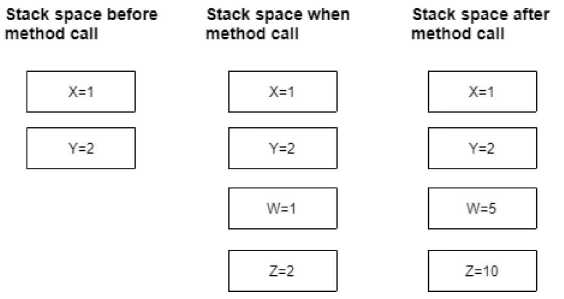
```
public static void main(String[] args) {

        int x = 1;
        int y = 2;
        System.out.print("Values of x & y be
fore primitive modification: ");
        System.out.println(" x = " + x + " ;
 y = " + y );
        modifyPrimitiveTypes(x,y);
        System.out.print("Values of x & y af
ter primitive modification: ");
        System.out.println(" x = " + x + " ;
 y = " + y );
    }


private static void modifyPrimitiveTypes(int
 x, int y)
    {
        x = 5;
        y = 10;
    }
```

Output:

```
Values of x & y before primitive modificatio
n:  x = 1 ; y = 2
Values of x & y after primitive modification
:  x = 1 ; y = 2
```

Output Description:

The 2 variables *x & y* are of primitive types and they are stored inside the stack memory. When calling ***modifyPrimitiveTypes(),*** 2 copies are created inside the stack memory (let's say w & z) and then passed to the method. **Hence original variables are not being sent to the method and any modification inside the method flow is affecting only the copies.**



- **Passing wrapper arguments:**

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

        Integer obj1 = new Integer(1);
        Integer obj2 = new Integer(2);
        System.out.print("Values of obj1 & o
bj2 before wrapper modification: ");
        System.out.println("obj1 = " + obj1.
intValue() + " ; obj2 = " + obj2.intValue())
;

        modifyWrappers(obj1, obj2);

        System.out.print("Values of obj1 & o
bj2 after wrapper modification: ");
        System.out.println("obj1 = " + obj1.
intValue() + " ; obj2 = " + obj2.intValue())
;

}

private static void modifyWrappers(Integer x
, Integer y)
    {
        x = new Integer(5);
        y = new Integer(10);
    }
```

Output:

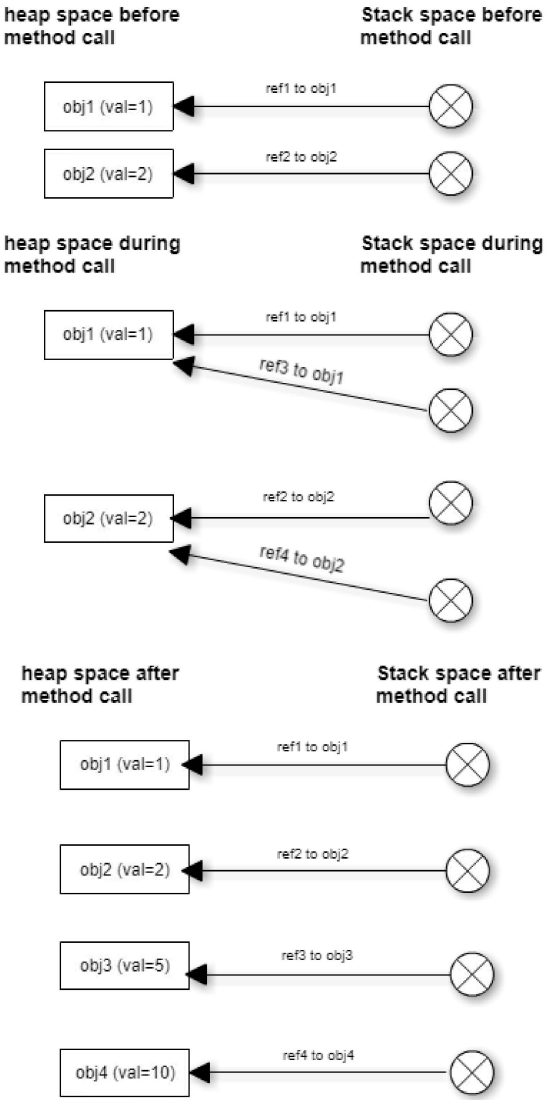
```
Values of obj1 & obj2 before wrapper modific
ation: obj1 = 1 ; obj2 = 2
Values of obj1 & obj2 after wrapper modifica
tion: obj1 = 1 ; obj2 = 2
```

Output Description:

Wrappers are stored inside the heap memory with a correspondent reference inside the stack memory.

When calling ***modifyWrappers(),*** a copy for each reference is created inside the stack memory and the

copies are passed to the method. **Any change on the reference inside the method is actually changing the reference of the copies and not the original references.**



P.S: if you change the value of wrapper objects inside the method like this: ***x* += *2*,** the change is not reflected outside the method since wrapper objects are immutable which means that they create a new instance each time their state is modified. For more information about immutable classes check "How to

create an immutable class in java".
String objects work similar to wrappers, so the above rules apply also on strings.

- **Passing Collection argument:**

```
public static void main(String[] args) {
        List<Integer> lstNums = new ArrayLis
t<Integer>();
        lstNums.add(1);
        System.out.println("Size of list bef
ore List modification = " + lstNums.size());
        modifyList(lstNums);
        System.out.println("Size of list aft
er List modification = " + lstNums.size());
    }

private static void modifyList(List<Integer>
 lstParam)
    {
        lstParam.add(2);
    }
```

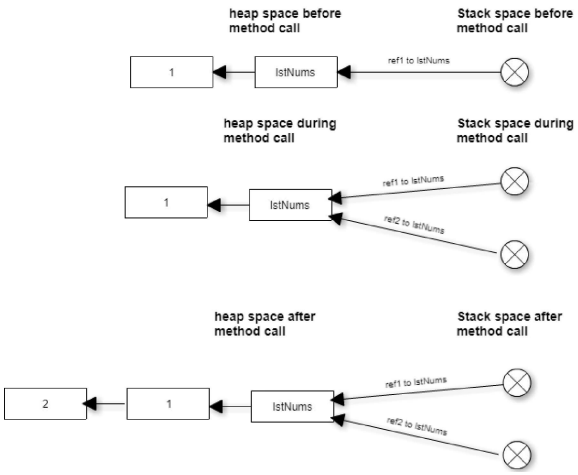
Output:

```
Size of list before List modification = 1
Size of list after List modification = 2
```

Output Description:

When defining an **Arraylist or any collection** in java, a reference is created inside the stack which points to multiple objects inside the heap memory, when calling ***modifyList(),*** a copy of the reference is created and passed to the method, so that the actual object data is referenced by 2 references and any change done by one reference is reflected on the other.

Inside the method, we called *lstParam.add(2)* , which actually tries to create a new Integer object in the heap memory and link it to the existing list of objects . **Hence the original list reference can see the modification since both references are pointing to the same object in memory.**



- **Passing business object as argument:**

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

        Student student = new Student();
        System.out.println("Value of name be
fore Student modification = " + student.getN
ame());
        modifyStudent(student);
        System.out.println("Value of name af
ter Student modification = " + student.getNa
me());
    }

private static void modifyStudent(Student st
udent)
    {
        student.setName("Alex");
    }
```

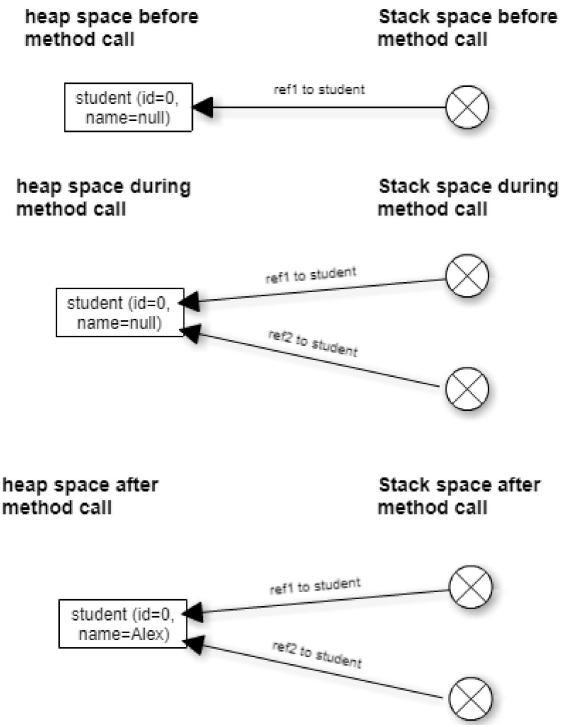
Output:

```
Value of name before Student modification =
```

```
null
Value of name after Student modification = A
lex
```

Output Description:

The *student* object is created inside the heap space and a reference for it is defined inside the stack, when calling *modifyStudent(),* a copy of the reference is created inside the stack and passed to the method. **Any modifications on the object attributes inside the method is reflected on the original reference.**



4. Conclusion

In java, arguments are always passed by value , the copy would be either a reference or a variable depending on the original variable type. From now on, you can use the following tips in order to understand how modifying arguments inside the method affect the original variable:

1. Modifying the value of a primitive argument would never affect the original variable.
2. Changing the reference of an object argument inside the method would never affect the original reference, however it creates a completely new object in the heap space.
3. Modifying the attributes of the object argument inside the method is reflected outside it.
4. Modifying collections & maps inside the method is reflected outside it.

Tags: #java

husseinterek

Founder of programmergate.com, I have a passion in software engineering and everything related to java environment.

YOU MAY ALSO LIKE...

How to use Enums in java

AUGUST 2, 2017

Java IO: difference between absolute,relative and canonical path

JULY 28, 2017

How to create an immutable class in java

JULY 21, 2017

Leave a Reply

8 Comments on "Java – pass by reference or pass by value"

Join the discussion

▲ newest ▲ oldest ▲ most voted

vikas

great explanation

Guest

+ 0 − Reply

7 months ago

husseinterek Thank you Vikas. Glad you found it helpful, you can subscribe to our mailing list to get notified of similar interesting articles.

Author

+ 0 − Reply

6 months ago

Ritika

Nice explanation

Guest

+ 0 − Reply

7 months ago

husseinterek

Thank you Ritika. You can subscribe to our mailing list to get notified of similar interesting articles.

Author

+ 0 − Reply

6 months ago

Sudheer

Great work, clean and nice explanation..

Guest

+ 0 − Reply

6 months ago

husseinterek

Thank you Sudheer. You can subscribe to our mailing list to get notified of any new posts.

Author

+ 0 − Reply

6 months ago

Guest

Turkmen

One of the greatest explanations of this topic I have ever came across ! thanks

+ 0 − Reply

6 months ago

Author

husseinterek

Thanks Turkmen for checking out the post! Glad you found it helpful.
You can subscribe to our mailing list to get notified of any new posts.

+ 0 − Reply

6 months ago