JAVA CHALLENGERS
By Rafael Del Nero, Java Developer, JavaWorld
JAN 2, 2020 11:37 AM PST

About
Tease your mind and test your learning, with these quick introductions to challenging concepts in Java programming.

Does Java pass by reference or pass by value?

Here's what happens when you pass an object reference to a method in Java

Many programming languages allow passing parameters *by reference or by value*. In Java, we can only pass parameters *by value*. This imposes some limits and also raises questions. For instance, if the parameter value is changed in the method, what happens to the value following method execution? You may also wonder how Java manages object values in the memory heap. This *Java Challenger* helps you resolve these and other common questions about object references in Java.

Get the source code

Get the code for this Java Challenger. You can run your own tests while you follow the examples.

Object references are passed by value

All object references in Java are passed by value. This means that a copy of the value will be passed to a method. But the trick is that passing a copy of the value also changes the real value of the object. To understand why, start with this example:

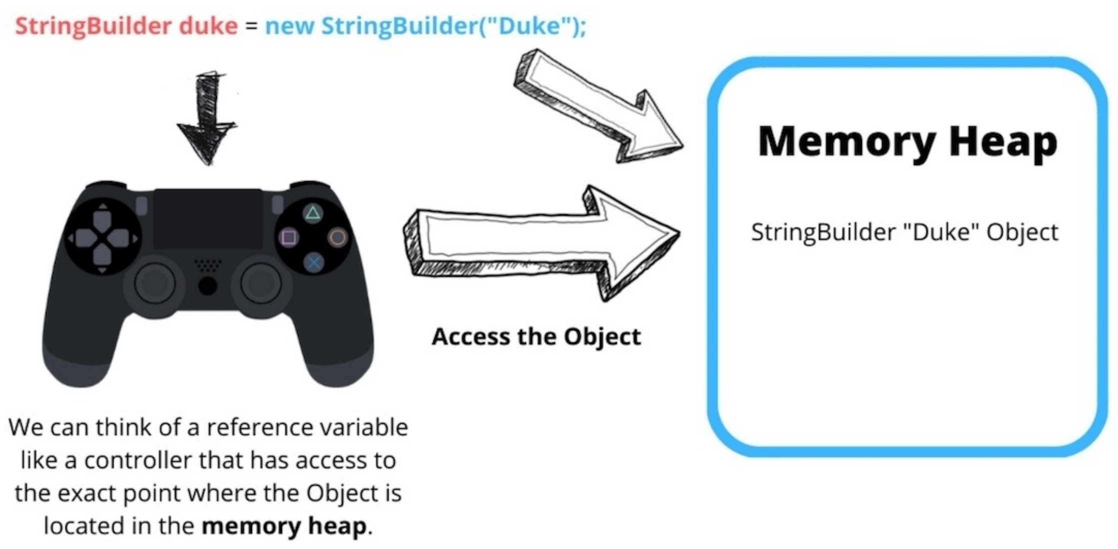
```
public class ObjectReferenceExample {

        public static void main(String... doYourBest) {
            Simpson simpson = new Simpson();
            transformIntoHomer(simpson);
            System.out.println(simpson.name);
        }

        static void transformIntoHomer(Simpson simpson) {
            simpson.name = "Homer";
        }

}

class Simpson {
        String name;
}
```

What do you think the `simpson.name` will be after the `transformIntoHomer` method is executed?

[Also on InfoWorld: JDK 15: The new features in Java 15]

In this case, it will be Homer! The reason is that Java object variables are simply references that point to real objects in the memory heap. Therefore, even though Java passes parameters to methods by value, if the variable points to an object reference, the real object will also be changed.

If you're still not quite clear how this works, take a look at the figure below.


StringBuilder duke = new StringBuilder("Duke");
Memory Heap
StringBuilder "Duke" Object
Access the Object
We can think of a reference variable like a controller that has access to the exact point where the Object is located in the **memory heap**.

Rafael Chinelato Del Nero

Are primitive types passed by value?

Like object types, primitive types are also passed by value. Can you deduce what will happen to the primitive types in the following code example?

```java
public class PrimitiveByValueExample {

        public static void main(String... primitiveByValue) {
            int homerAge = 30;
            changeHomerAge(homerAge);
            System.out.println(homerAge);
        }

        static void changeHomerAge(int homerAge) {
            homerAge = 35;
        }
}
```

If you determined that the value would change to 30, you are correct. It's 30 because (again) Java passes object parameters by value. The number 30 is just a copy of the value, not the real value. Primitive types are allocated in the stack memory, so only the local value will be changed. In this case, there is no object reference.

Passing immutable object references

What if we did the same test with an immutable `String` object?

The JDK contains many immutable classes. Examples include the wrapper types `Integer`, `Double`, `Float`, `Long`, `Boolean`, `BigDecimal`, and of course the very well known `String` class.

In the next example, notice what happens when we change the value of a `String`.

```
public class StringValueChange {

        public static void main(String... doYourBest) {
            String name = "";
            changeToHomer(name);
            System.out.println(name);
        }

        static void changeToHomer(String name) {
            name = "Homer";
        }
}
```

What do you think the output will be? If you guessed “” then congratulations! That happens because a `String` object is immutable, which means that the fields inside the `String` are final and can’t be changed.

Making the `String` class immutable gives us better control over one of Java's most used objects. If the value of a `String` could be changed, it would create a lot of bugs. Also note that we're not changing an attribute of the `String` class; instead, we're simply assigning a new `String` value to it. In this case, the "Homer" value will be passed to `name` in the `changeToHomer` method. The `String` "Homer" will be eligible to be garbage collected as soon as the `changeToHomer` method completes execution. Even though the object can't be changed, the local variable will be.

Strings and more

Learn more about Java's `String` class and more: See all of Rafael's posts in the Java Challengers series.

Passing mutable object references

Unlike `String`, most objects in the JDK are mutable, like the `StringBuilder` class. The example below is similar to the previous one, but features `StringBuilder` rather than `String`:

```
static class MutableObjectReference {
   public static void main(String... mutableObjectExample) {
     StringBuilder name = new StringBuilder("Homer ");
     addSureName(name);
     System.out.println(name);
   }

   static void addSureName(StringBuilder name) {
     name.append("Simpson");
   }
 }
```

Can you deduce the output for this example? In this case, because we're working with a mutable object, the output will be "Homer Simpson." You could expect the same behaviour from any other mutable object in Java.

You've already learned that Java variables are passed by value, meaning that a copy of the value is passed. Just remember that the copied value points to a *real object* in the Java memory heap. Passing by value still changes the value of the real object.

Take the object references challenge!

In this Java Challenger we'll test what you've learned about object references. In the code example below, you see the immutable `String` and the mutable `StringBuilder` class. Each is being passed as a parameter to a method. Knowing that Java only passes by value, what do you believe will be the output once the main method from this class is executed?

```java
public class DragonWarriorReferenceChallenger {

  public static void main(String... doYourBest) {
    StringBuilder warriorProfession = new StringBuilder("Dragon ");
    String warriorWeapon = "Sword ";
    changeWarriorClass(warriorProfession, warriorWeapon);

    System.out.println("Warrior=" + warriorProfession + " Weapon=" + warriorWeapon);
  }

  static void changeWarriorClass(StringBuilder warriorProfession, String weapon) {
    warriorProfession.append("Knight");
    weapon = "Dragon " + weapon;

    weapon = null;
    warriorProfession = null;
  }

}
```

Here are the options, check the end of this article for the answer key.

A: Warrior=null Weapon=null
B: Warrior=Dragon Weapon=Dragon
C: Warrior=Dragon Knight Weapon=Dragon Sword

D: Warrior=Dragon Knight Weapon=Sword

What's just happened?

The first parameter in the above example is the `warriorProfession` variable, which is a mutable object. The second parameter, weapon, is an immutable `String`:

```
static void changeWarriorClass(StringBuilder warriorProfession, String weapon) {
   ...
 }
```

Now let's analyze what is happening inside this method. At the first line of this method, we append the `Knight` value to the `warriorProfession` variable. Remember that `warriorProfession` is a mutable object; therefore the real object will be changed, and the value from it will be "Dragon Knight."

```
warriorProfession.append("Knight");
```

In the second instruction, the immutable local `String` variable will be changed to "Dragon Sword." The real object will never be changed, however, since `String` is immutable and its attributes are final:

```
weapon = "Dragon " + weapon;
```

Finally, we pass `null` to the variables here, but not to the objects. The objects will remain the same as long as they are still accessible externally--in this case through the main method. And, although the local variables will be null, nothing will happen to the objects:

```
weapon = null;
warriorProfession = null;
```

From all of this we can conclude that the final values from our mutable `StringBuilder` and immutable `String` will be:

```
System.out.println("Warrior=" + warriorProfession + " Weapon=" + warriorWeapon);
```

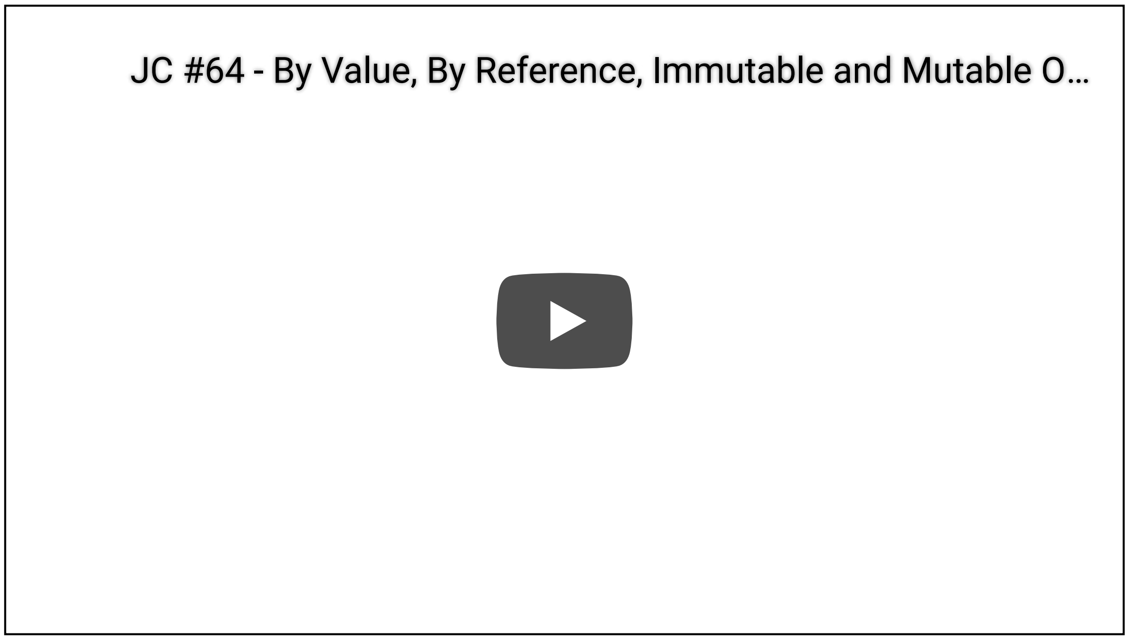
The only value that changed in the `changeWarriorClass` method was `warriorProfession`, because it's a mutable `StringBuilder` object. Note that `warriorWeapon` did not change because it's an immutable `String` object.

The correct output from our Challenger code would be:

D: Warrior=Dragon Knight Weapon=Sword.

Video challenge! Debugging object references in Java

Debugging is one of the easiest ways to fully absorb programming concepts while also improving your code. In this video you can follow along while I debug and explain object references in Java.



Common mistakes with object references

- Trying to change an immutable value by reference.
- Trying to change a primitive variable by reference.

- Expecting the real object won't change when you change a mutable object parameter in a method.

What to remember about object references

- Java always passes parameter variables by value.
- Object variables in Java always point to the real object in the memory heap.
- A mutable object’s value can be changed when it is passed to a method.
- An immutable object’s value cannot be changed, even if it is passed a new value.
- “Passing by value” refers to passing a copy of the value.
- “Passing by reference” refers to passing the real reference of the variable in memory.

Learn more about Java

- Get more quick code tips: Read all of Rafael's posts in the JavaWorld Java Challengers series.
- Learn more about mutable and immutable Java objects (such as `String` and `StringBuffer`) and how to use them in your code.
- You might be surprised to learn that Java's primitive types are controversial. In this feature, John I. Moore makes the case for keeping them, and learning to use them well.
- Keep building your Java programming skills in the Java Dev Gym.
- If you liked debugging Java inheritance, check out more videos in Rafael's Java Challenges video playlist (videos in this series are not affiliated with JavaWorld).
- Want to work on stress free projects and write bug-free code? Head over to the NoBugsProject for your copy of *No Bugs, No Stress - Create Life-Changing Software Without Destroying Your Life*.

This story, "Does Java pass by reference or pass by value?" was originally published by JavaWorld.

YOU MAY ALSO LIKE

Tim O'Reilly: the golden age of the programmer is over

Why developers should use graph databases

Most cloud security problems breathe

How to work with Quartz.Net in C#

Machine teaching with Microsoft’s Project Bonsai

The decline of Heroku

Angular 12 arrives with pile of improvements

ProxyJump is safer than SSH agent forwarding

Don’t migrate your problems to the cloud

The 24 highest paying developer roles in 2020

7 tools transforming JavaScript development