Pass by Reference vs. Pass by Value

Most methods are passed arguments when they are called. An argument may be a constant or a variable. For example, in the expression

```
Math.sqrt(33)
```

the constant `33` is passed to the `sqrt()` method of the `Math` class. In the expression

```
Math.sqrt(x)
```

the variable `x` is passed. This is simple enough, however there is an important but simple principle at work here. If a variable is passed, the method receives a copy of the variable's value. The value of the original variable cannot be changed within the method. This seems reasonable because the method only has a copy of the value; it does not have access to the original variable. This process is called *pass by value*.

However, if the variable passed is an object, then the effect is different. This idea is explored in detail in this section. But first, let's clarify the terminology. For brevity, we often say things like, "this method returns an object ...", or "this method is passed an object as an argument ...". As noted earlier, this not quite true. More precisely, we should say,

this method returns a reference to an object ...

or

this method is passed a reference to an object as an argument ...

In fact, objects, per se, are never passed to methods or returned by methods. It is always "a reference to an object" that is passed or returned.

The term "variable" also deserves clarification. There are two types of variables in Java: those that hold the value of a primitive data type and those that hold a reference to an object. A variable that holds a reference to an object is an "object variable" — although, again, the prefix "object" is often dropped for brevity.

Returning to the topic of this section, *pass by value* refers to passing a constant or a variable holding a primitive data type to a method, and *pass by reference* refers to passing an object variable to a method. In both cases a copy of the variable is passed to the method. It is a copy of the "value" for a primitive data type variable; it is a copy of the "reference" for an object variable. So, a method receiving an object variable as an argument receives a copy of the reference to the original object. Here's the clincher: If the method uses that reference to make changes to the object, then the original object is changed. This is reasonable because both the original reference and the copy of the reference "refer to" to same thing — the original object.

There is one exception: strings. Since `String` objects are immutable in Java, a method that is passed a reference to a `String` object cannot change the original object. This distinction is also brought out in this section.

Let's explore pass by reference and pass by value through a demo program (see Figure 1). As a self-test, try to determine the output of this program on your own before proceeding. If understood the preceding discussion and if you guessed the correct output, you can confidently skip the rest of this section. The print statement comments include numbers to show the order of printing.

```
 1  public class DemoPassByReference
 2  {
 3     public static void main(String[] args)
 4     {
 5        // Part I - primitive data types
 6        int i = 25;
 7        System.out.println(i);                          // print it (1)
 8        iMethod(i);
 9        System.out.println(i);                          // print it (3)
10        System.out.println("-----------------");
11
12        // Part II - objects and object references
13        StringBuffer sb = new StringBuffer("Hello, world");
14        System.out.println(sb);                         // print it (4)
15        sbMethod(sb);
16        System.out.println(sb);                         // print it (6)
17        System.out.println("-----------------");
18
19        // Part III - strings
20        String s = "Java is fun!";
21        System.out.println(s);                          // print it (7)
22        sMethod(s);
23        System.out.println(s);                          // print it (9)
24     }
25
26     public static void iMethod(int iTest)
27     {
28        iTest = 9;                                      // change it
29        System.out.println(iTest);                      // print it (2)
30     }
31
32     public static void sbMethod(StringBuffer sbTest)
33     {
34        sbTest = sbTest.insert(7, "Java ");             // change it
35        System.out.println(sbTest);                     // print it (5)
36     }
37
38     public static void sMethod(String sTest)
39     {
40        sTest = sTest.substring(8, 11);                 // change it
41        System.out.println(sTest);                      // print it (8)
42     }
43  }
```

Figure 1. `DemoPassByReference.java`

This program generates the following output:

```
25
9
25
-----------------
Hello, world
Hello, Java world
Hello, Java world
-----------------
Java is fun!
fun
Java is fun!
```

`DemoPassByReference` is organized in three parts. The first deals with primitive data types, which are passed by value. The program begins by declaring and `int` variable named `i` and initializing it with the value 25 (line 6). The memory assignment just after line 6 is illustrated in Figure 2a. The value of `i` is printed in line 7.

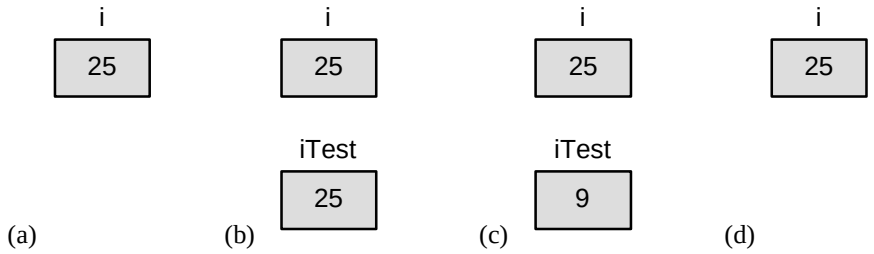

Figure 2. Memory assignments for call-by-value example (`int` variables)

Then, the `int` variable is passed as an argument to a method called `iMethod()` (line 8). The method is defined in lines 26-30. Within the method, a copy of `i` exists as a local variable named `iTest`. The memory assignments just after `iMethod()` begins execution are shown in Figure 2b. Note that `i` and `iTest` are distinct: They are two separate variables that happen to hold the same value. Within the method, the value is changed to 9 (line 28) and then printed again. Back in the `main()` method the original variable is also printed again (line 9). Changing the passed value in `iMethod()` had no effect on the original variable, and, so, the third value printed is the same as the first.

Part II of the program deals with objects and object references. The pass-by-reference concept is illustrated by the object variables `sb` and `sbTest`. In the `main()` method, a `StringBuffer` object is instantiated and initialized with "`Hello, world`" and a reference to it is assigned to the `StringBuffer` object variable `sb` (line 13).

The memory assignments for the object and the object variable are illustrated in Figure 3a. This corresponds to the state of the `sb` just after line 13 in Figure 1. The object is printed in line 14. In line 15, the `sbMethod()` method is called with `sb` as an argument. The method is defined in lines 32-36. Within the method, a copy of `sb` exists as a local variable named `sbTest`. The memory assignments just after the `sbMethod()` begins execution are shown in Figure 3b. Note that `sb` and `sbTest` refer to the same object.

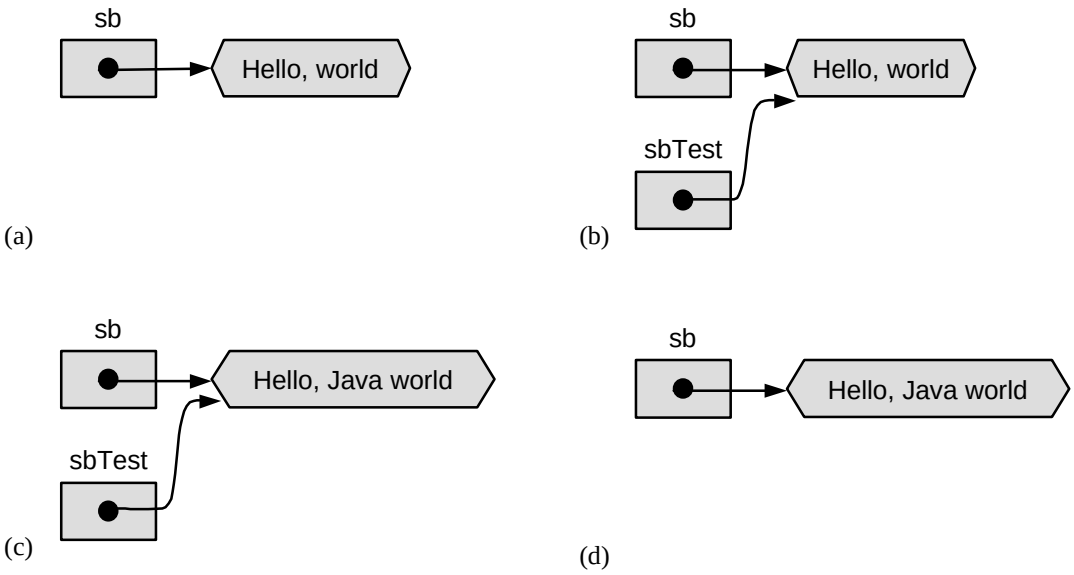

Figure 3. Memory assignments for call-by-reference example (`StringBuffer` objects)

In line 34, the object is modified using the `insert()` method of the `StringBuffer` class. The method is called through the instance variable `sbTest`. The memory assignments just after line 34 executes are shown in Figure 3c. After the `sbMethod()` finishes execution, control passes back to the `main()` method and `sbTest` ceases to exist. The memory assignments just after line 15 in Figure 1 are shown in Figure 3d. Note that the original object has changed, and that the original object variable, `sb`, now refers to the updated object.

The scenario played out above, is a typical example of pass by reference. It demonstrates that methods can change the objects instantiated in other methods when they are passed a reference to the object as an argument. A similar scenario could be crafted for objects of any of Java's numerous classes. The `StringBuffer` class is a good choice for the example, because `StringBuffer` objects are tangible, printable entities. Other objects are less tangible (e.g., objects of the `Random` class); however, the same rules apply.

Part III of the program deals with strings. The `String` class is unlike other classes in Java because `String` objects, once instantiated, cannot change. For completeness, let's walk through the memory assignments for the `String` objects in the `DemoPassByRefererнce` program. A `String` object is instantiated in line 20 and a reference to it is assigned to the `String` object variable `s`. The memory assignments at this point are shown in Figure 4a.

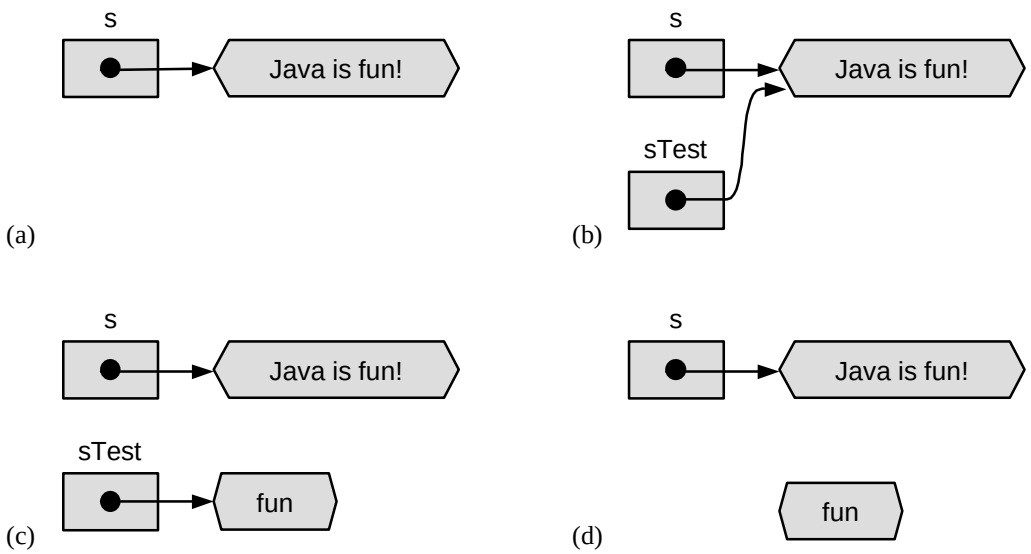

Figure 4. Memory assignments for call-by-reference example (`String` objects)

In line 22 the `sMethod()` is called with `s` as an argument. The method is defined in lines 38-42. Within the method, a copy of `s` exists as a local variable named `sTest`. The memory assignments just after the `sMethod()` begins execution are shown in Figure 4b. Note that `s` and `sTest` refer to the same object. At this point, there is no difference in how memory was assigned for the `String` or `StringBuffer` objects. However, in line 40, Java's unique treatment of `String` objects surfaces. The `substring()` method of the `String` class is called to extract the string "fun" from the original string. The result is the instantiation of a new `String` object. A reference to the new object is assigned to `sTest`. The memory assignments just after line 37 executes are shown in Figure 4c.

After the `sMethod()` finishes execution, control passes back to the `main()` method and `sTest` ceases to exist. The memory assignments just after line 22 in Figure 1 are shown in Figure 4d. Note that the original object did *not* change. The `String` object containing "fun" is now redundant and its space is eventually reclaimed.