

## 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.

```

1 public class DemoPassByReference
2 {
3     public static void main(String[] args)
4     {
5         // declare and initialize variables and objects
6         int i = 25;
7         String s = "Java is fun!";
8         StringBuffer sb = new StringBuffer("Hello, world");
9
10        // print variable i and objects s and sb
11        System.out.println(i); // print it (1)
12        System.out.println(s); // print it (2)
13        System.out.println(sb); // print it (3)
14        System.out.println("-----");
15
16        // attempt to change i, s, and sb using methods
17        iMethod(i);
18        sMethod(s);
19        sbMethod(sb);
20        System.out.println("-----");
21
22        // print variable i and objects s and sb (again)
23        System.out.println(i); // print it (7)
24        System.out.println(s); // print it (8)
25        System.out.println(sb); // print it (9)
26    }
27
28    public static void iMethod(int iTest)
29    {
30        iTest = 9; // change it
31        System.out.println(iTest); // print it (4)
32        return;
33    }
34
35    public static void sMethod(String sTest)
36    {
37        sTest = sTest.substring(8, 11); // change it
38        System.out.println(sTest); // print it (5)
39        return;
40    }
41
42    public static void sbMethod(StringBuffer sbTest)
43    {
44        sbTest = sbTest.insert(7, "Java "); // change it
45        System.out.println(sbTest); // print it (6)
46        return;
47    }
48 }

```

Figure 1. DemoPassByReference.java

This program generates the following output:

```

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

```

`DemoPassByReference` begins by declaring and initializing three variables: an `int` variable named `i`, a `String` object variable named `s`, and a `StringBuffer` object variable named `sb` (see lines 6-8). The three are printed in lines 11-13. Then, each variable is passed as an argument to a method. Within each method, the copy of the variable exists as a local variable. The value of the variable — or the value of the object referred to by the variable, in the case of the `String` and `StringBuffer` object variables — is changed and printed within each method (lines 31, 38, and 45). The print statements are numbered to show the order of printing. Back in the `main()` method, the three values are printed again (lines 23-25). Have a look at the output and see if it is consistent with our previous discussion. It is. Let's examine the treatment of each of the three types of variables used in `DemoPassByReference`. We'll begin with the `StringBuffer` objects.

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 8). The memory assignments for the object and the object variable are illustrated in Figure 2a. This corresponds to the state of the `sb` and `sbTest` variables just after line 8 in Figure 1. In line 19, the `sbMethod()` method is called with `sb` as an argument. The method is defined in lines 42-47. 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 2b. Note that `sb` and `sbTest` refer to the same object.

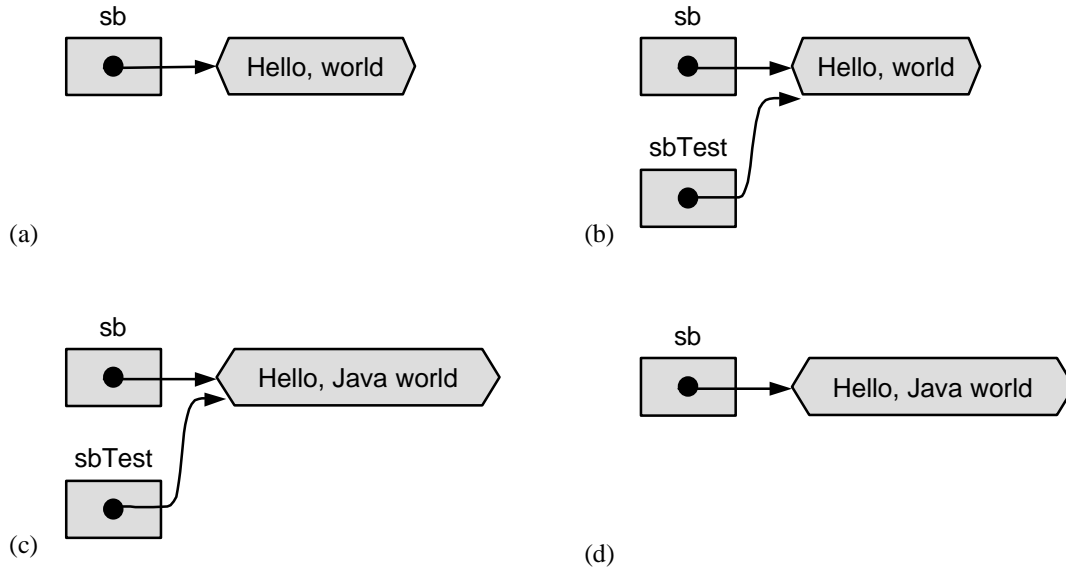


Figure 2. Memory assignments for call-by-reference example (StringBuffer objects)

In line 44, 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 44 executes are shown in Figure 2c. After the `sbMethod()` finishes execution, control passes back to the `main()` method and `sbTest` ceases to exist. The memory assignments just after line 19 in Figure 1 are shown in Figure 2d. 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 pass 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.

A major exception is, of course, the `String` class, because `String` objects, once instantiated, cannot change. For completeness, let's walk through the memory assignments for the `String` objects in the `DemoPassByReference` program. A `String` object is instantiated in line 7 and a reference to it is assigned to the `String` object variable `s`. The memory assignments at this point are shown in Figure 3a.

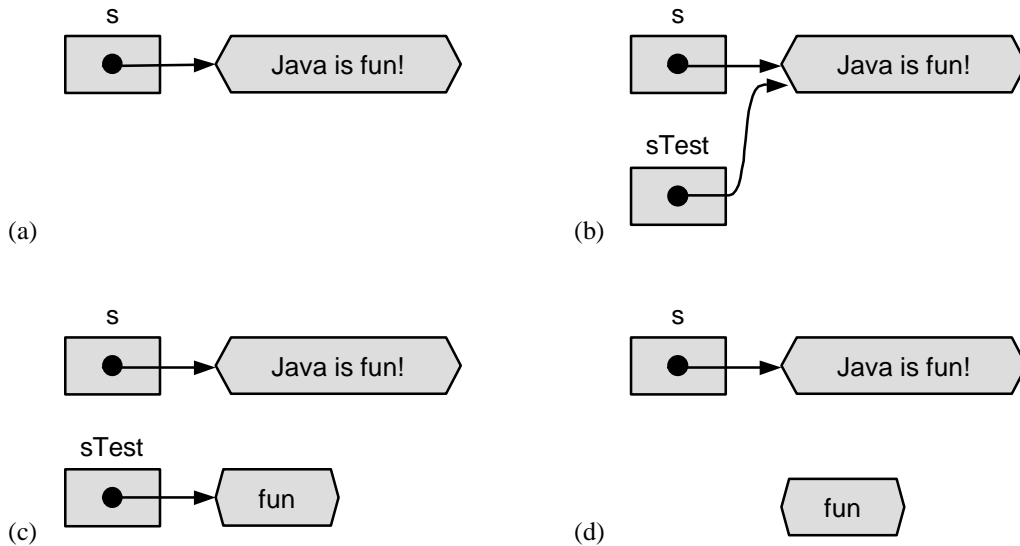


Figure 3. Memory assignments for call-by-reference example (String objects)

In line 18 the `sMethod()` is called with `s` as an argument. The method is defined in lines 35-40. 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 3b. 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 37, 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 3c.

After the `sMethod()` finishes execution, control passes back to the `main()` method and `sTest` ceases to exist. The memory assignments just after line 18 in Figure 1 are shown in Figure 3d. Note that the original object did *not* change. The `String` object containing "fun" is now redundant and its space is eventually reclaimed.

Finally, let's examine the memory assignments for the integer variables in `DemoPassByReference`. When an integer variable is passed as an argument to a method, it is passed by value. In the demo program, an integer variable named `i` is declared and initialized with 25 in line 6. The memory assignment is illustrated in Figure 4a.

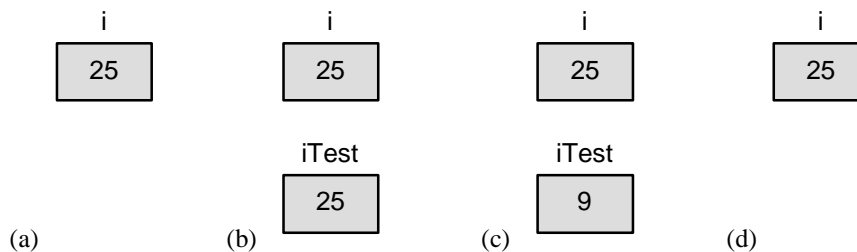


Figure 4. Memory assignments for call-by-value example (int variables)

In line 17, `iMethod()` is called with `i` as an argument. The method is defined in lines 28-33. 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 3b. Note that `i` and `iTest` are distinct: They are two separate variables that happen to hold the same value.

Within the `iMethod()` method, `iTest` is re-assigned the value 9. The memory assignments just after line 30 executes are shown in Figure 4c. After the `iMethod()` finishes execution, control passes back to the `main()` method and `iTest` ceases to exist. The memory assignments just after line 17 in Figure 1 are shown in Figure 4d. Note that the value of the original variable did *not* change.