Lesson 19: Passing by Value and by Reference

Consider the following class:

```
public class Tester
{
     public static void main(String[] args)
     {
          double b[]= new double[10];
          b[3] = 19;

          BankAccount myAccount = new BankAccount(79);
          //sets balance to
          int y =39; //79

          method1(y, b, myAccount);
          System.out.println(y + " " + b[3] + " " +
          myAccount.balance);
          //prints …. 39 -54.0 702.0
     }
     public static void method1(int x, double a[ ],
     BankAccount theAccount)
     {
          x =332;
          a[3] = -54;
          theAccount.balance = 702;
     }
}
```

Passing by value:

This demonstrates that primitive data types like *int*, *double*, etc. are **passed by value**; i.e. a new copy of the variable is temporarily created in the method and any changes to that variable in the method are **not** made to the original. Notice above that we pass an *int* type *y* to the method where the temporary copy is called *x*. The *x* is changed to 332; however, back in the calling code, *y* stays at its original value of 39.

Passing by reference:

Arrays (the *b[ ]* array) and **objects** (*myAccount*) are **passed by reference**. Notice both of these are modified in *method1*, and sure enough, back in the calling code these changes are reflected there.

> *Actually, arrays are objects, so our rule is simply stated,* ***"Objects are passed by reference."***

There is an exception to the above rule. A *String* is an object; however, it acts like a primitive data type and is passed by value.

Passing an array to a method:

Now we are going to look a little deeper into passing arrays to methods. We must remember that the array may be named something different in the method; however, that new name is just a **reference** back to the original array. Passing an array **does not create a new array** in the method.

Consider the following code:

```
public class Tester
{
     public static void main(String args[])
     {
          int s[] = {1,2,3,4,5,6};
          for(int g = 0; g < s.length; g++) //prints first
          System.out.print(s[g] + " ");
          System.out.print("\n");
          testMethod(s);
          for(int g = 0; g < s.length; g++) //prints last
          System.out.print(s[g] + " ");
     }
     public static void testMethod(int pp[])
     {
          //pp references the s array in main
          int len = pp.length;
          int t2[] = new int[len];

          for(int j=0; j<len; j++)
               t2[j] = pp[len -j -1];

          for(int k=0; k<t2.length; k++) //prints t2 array
               System.out.print(t2[k] + " ");

          System.out.print("\n");
          pp = t2; //pp now references the local t2 array
     }
}
```

The output looks like this:

```
1 2 3 4 5 6
6 5 4 3 2 1
1 2 3 4 5 6
```

Everything about this printout looks normal except this last line. How do we explain that the *s* array is completely unchanged in the *main* method even though *t2* is clearly reversed in *testMethod* and is then assigned to *pp* (which seems to be a reference back to *s*)?

The explanation is in realizing that *testMethod* initially **does** assign a reference for *pp* back to the *s* array. However, *pp* is later assigned as a reference to the local array *t2* and no longer references the *s* array in *main*. From this point, *pp* cannot affect the *s* array back in *main*. The assignment *pp = t2* **does not assign numbers** in one array to another; rather, it **reassigns a reference** to an array.