Java With Us

Call by Value and Call by Reference

Before we look into what the terms call by value and call by reference mean, let us look at two simple programs and examine their output.

```
class CallByValue {

   public static void main ( String[] args ) {
      int x =3;
      System.out.println ( "Value of x before calling increment() is "+x);
      increment(x);
      System.out.println ( "Value of x after calling increment() is "+x);
   }

   public static void increment ( int a ) {
      System.out.println ( "Value of a before incrementing is "+a);
      a= a+1;
      System.out.println ( "Value of a after incrementing is "+a);
   }
}
```

The output of this program would be:

```
Value of x before calling increment() is 3
Value of a before incrementing is 3
Value of a after incrementing is 4
Value of x after calling increment() is 3
```

As is evident from the output, the value of x has remain unchanged, even though it was passed as a parameter to the increment() method. (This program contains two static methods. The method increment() was also specified to be static since only static members can be accessed by a static method)

And now, we move on to the second program, where we will make use of class Number that contains a single instance variable x.

```
class Number {
   int x;
}

class CallByReference {

   public static void main ( String[] args ) {
      Number a = new Number();
      a.x=3;
      System.out.println("Value of a.x before calling increment() is "+a.x);
      increment(a);
      System.out.println("Value of a.x after calling increment() is "+a.x);
   }

   public static void increment(Number n) {
      System.out.println("Value of n.x before incrementing x is "+n.x);
      n.x=n.x+1;
      System.out.println("Value of n.x after incrementing x is "+n.x);
   }
}
```

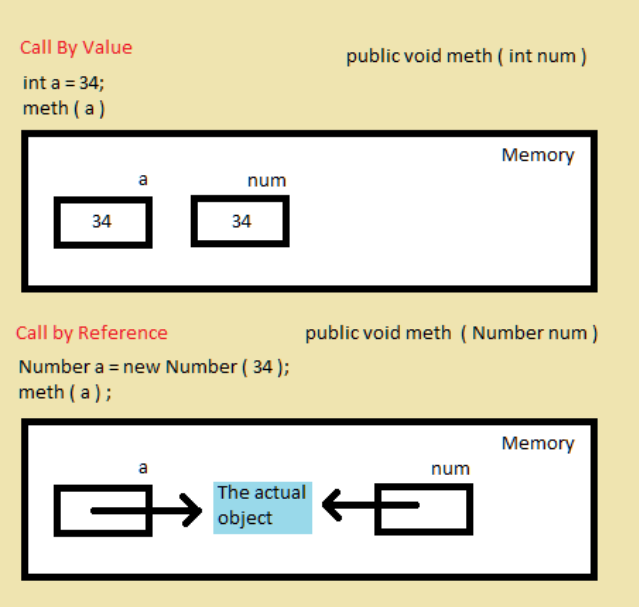
This program would give the following output

```
Value of a.x before calling increment() is 3
Value of n.x before incrementing x is 3
Value of n.x after incrementing x is 4
Value of a.x after calling increment() is 4
```

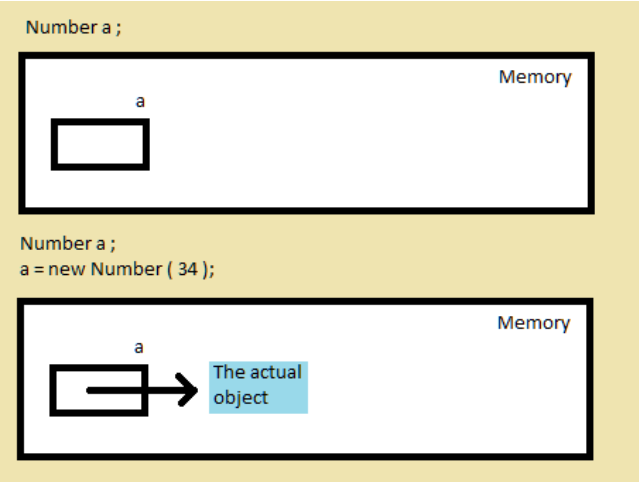
Now, there is a remarkable difference between the outputs obtained in the above two programs. In the first program, the change made to the variable a inside the increment() method had no effect on the original variable x that was passed as an argument. On the other hand, in the second program, changes made to the variable x that was a part of the object in the increment() method had an effect on the original variable (the object which contained that integer variable) that was passed as an argument. The difference lies in the type of the variable that was passed as an argument. int is a primitive data type while Number is a reference data type. Primitive data types in Java are passed by value while reference data types are passed by reference.

What we mean by passing a variable by value is that the value held in the variable that is passed as an argument is copied into the parameters that are defined in the method header. That is why changes made to the variable within the method had no effect on the variable that was passed. On the other hand, when objects are passed, the object itself is passed. No copy is made. Therefore changes made to the object within the method increment() had an effect on the original object.

The following figure illustrate call be reference and call by value.



The concept of call by reference can be better understood if one tries to look into what a reference actually is and how a variable of a class type is represented. When we declare a reference type variable, the compiler allocates only space where the memory address of the object can be stored. The space for the object itself isn't allocated. The space for the object is allocated at the time of object creation using the new keyword. A variable of reference type differs from a variable of a primitive type in the way that a primitive type variable holds the actual data while a reference type variable holds the address of the object which it refers to and not the actual object.



Consider the following program which illustrates theses concepts.

```
class Number {
   int x;
}

public class Reference {

   public static void main ( String[] args ) {
      Number a = new Number();
      a.x=4;
      System.out.println(a.x);
      Number b=a;
      b.x=5;
      System.out.println(b.x);
   }
}
```

The output would be:

```
4 5
```

This program creates just one object and not two. The statement Number b doesn't create a new object. Instead it only allocates some space where the address of the object to which b refers would be stored. The statement b=a; simply copies the value stored in b to a. This value isn't the object itself but is simply the address at which the object is stored. Therefore, both a and b refer to the same object. Any change made to the object through either of the variables gets reflected on the other variable also. That is why when we have changed the value of x through the variable name b to 5, the change was reflected on the value of x accessed through a. this is because, a and b are simply different names for the same object in the computers 'memory.

What the new keyword does is that it simply creates an object and returns reference to that object i.e. the address of that object. we assign this reference (address) to an appropriate variable.
We can compare two reference variables using the == relational operator. The result is true is both the vraibles refer to the same object, otherwise the result is false. Look at the following code for example:

```
Number a =new Number(); // first object
Number b = new Number(); // second object
Number c=b;// c also refers to the second object
boolean result1= a==b; // false
boolean result2= b==c; // true
```

Next : Casting
Prev : Scope and lifetime of Variables