Object Passing

Primitive Parameters

- Primitive types: boolean, byte, char, short, int, long, float, double
- In Java, all primitives are passed by value. This means a copy of the value is passed into the method
- Modifying the primitive parameter in the method does NOT change its value outside the method

Object Parameters

- Objects can be passed natively, just like primitives

- Complex {
- Private double real;
- Private double imag;
- Public Complex() // Default Constructor
- { real = 0.0; imag = 0.0; }
- Public Complex (double r, double im)
- { real = r; imag = im; }
- Public Complex Add (Complex b)
- {
- Double r= this.real + b.real;
- Double i = imag + b.imag;
- Complex c_new = new Complex (r,i);
- return c_new;
- }
- Public void Show ()
- {
- System.out,println(real + imag);
- }

```
• Public class ComplexTest {
  void main()
  {
  Complex C1 = new Complex(11,
  2.3);
  Complex C2 = new Complex(9,
  2.3);
  C1.show();
  C2.show();
  Complex C3 =  C1.Add(C2);
  C3.show();

  }
}
```

OBJECTS ARE PASSED BY VALUE

- It is often misstated that Object parameters are passed by Reference.
- While it is true that the parameter is a reference to an Object, the reference itself is passed by Value.

```
public class ObjectPass {
public  int value;

 public void increment(ObjectPass a){
     a.value++;

  }

  public static void main(String[] args) {
    ObjectPass p = new ObjectPass();
    p.value = 5;
     System.out.println("Before calling: " + p.value);
     increment(p);
    System.out.println("After calling: " + p.value);
      }
}
```

Output

-
  ```
  Before calling: 5
  After calling: 6
  ```

- The point here is that what we pass exactly is a handle of an object, and in the called method a new handle created and pointed to the same object.
- Now when more than one handles tied to the same object, it is known as **aliasing**.

- From the example above you can see that both **p** and **a** refer to the same object
- To prove this try to System.out.println(p) and System.out.println(a) you will see the same address.

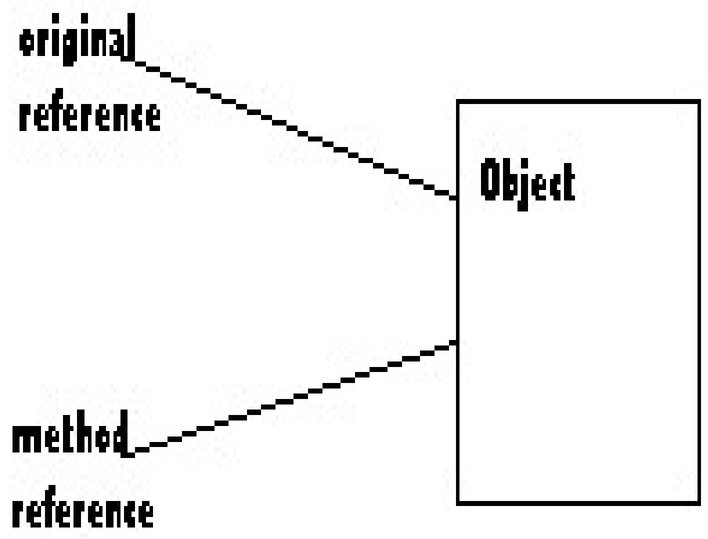


Figure 1. After being passed to a method, an object will have at least two references

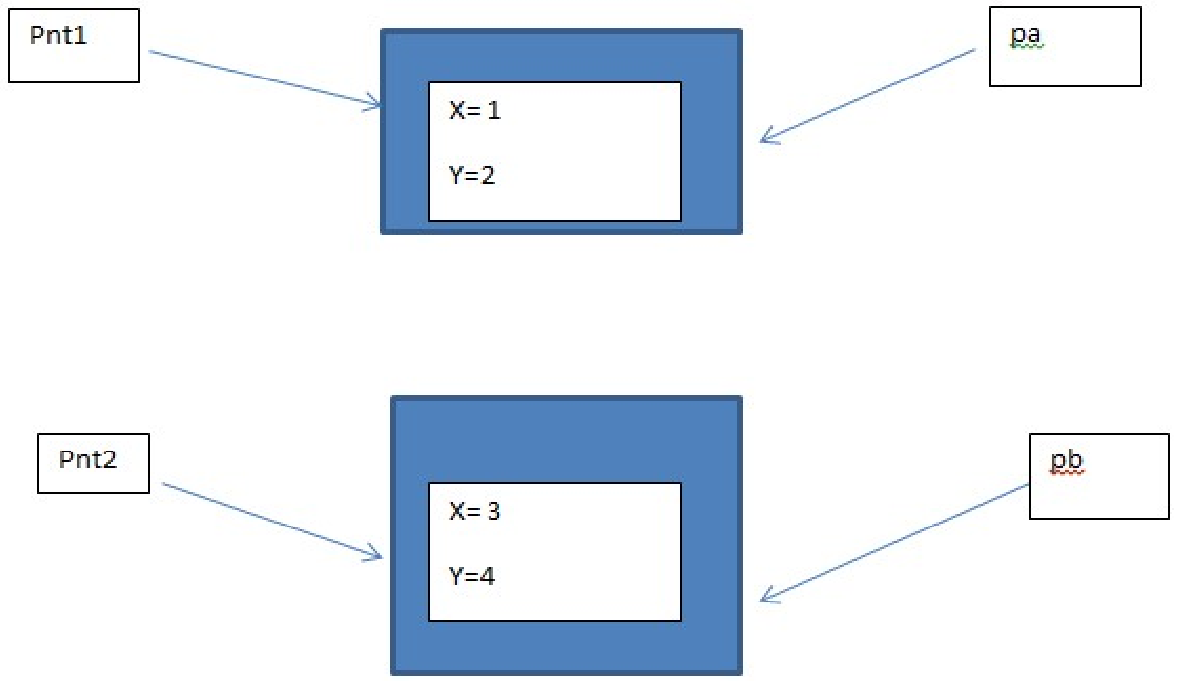
- This is the default way Java does when passing the handle to a called method, create alias.
- When you pass argument just to manipulate its value and not doing any changes to it, then you are safe.

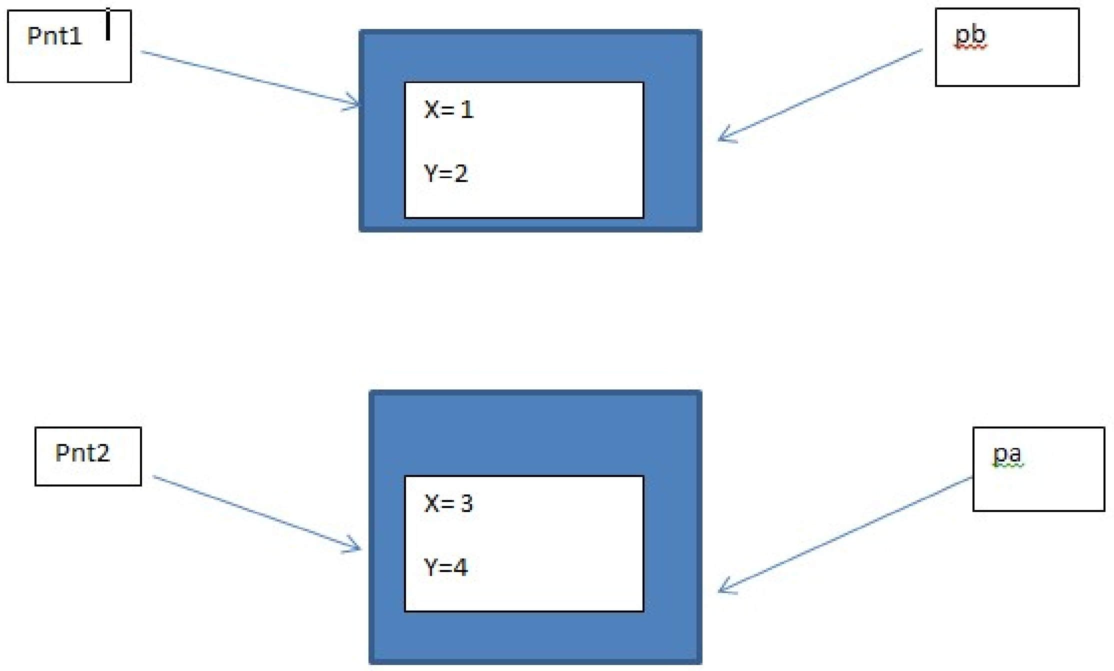
Beware…..

- **public void** tricky **(Point pa, Point pb)**
- **{**
- **Point ptemp = new Point ();**
- **ptemp = pb;**
- **pb= pa;**
- **pa= ptemp;**
- **System.out**.println("X: " + pa.x + " Y: " +pa.y);
  System.out.println("X: " + + pb.x + " Y: " + pb.y);
  }

- **public static void** main(**String** [] args)
- {
- **Point** pnt1 = **new Point**(1,2);
- **Point** pnt2 = **new Point**(3,4);
- **System.out**.println("X: " + pnt1.x + " Y: " +pnt1.y);
  System.out.println("X: " + pnt2.x + " Y: " +pnt2.y);
  System.out.println(" "); tricky(pnt1,pnt2);
- **System.out**.println("X: " + pnt1.x + " Y:" + pnt1.y);
  System.out.println("X: " + pnt2.x + " Y: " +pnt2.y);
- }

- The method “tricky” is not performing swapping of object passed by main(), it swaps the objects in the function “tricky”


Pnt1
pa
X= 1
Y=2
Pnt2
pb
X= 3
Y=4


Pnt1
pb
X= 1
Y=2
Pnt2
pa
X= 3
Y=4

The `equals` Method

- When the == operator is used with reference variables, the memory address of the objects are compared.
- The contents of the objects are not compared.
- All objects have an `equals` method.
- The default operation of the `equals` method is to compare memory addresses of the objects (just like the == operator).

The equals Method

- The `Stock` class has an `equals` method.
- If we try the following:

```
Stock stock1 = new Stock("GMX", 55.3);
Stock stock2 = new Stock("GMX", 55.3);
if (stock1 == stock2) // This is a mistake.
  System.out.println("The objects are the same.");
else
  System.out.println("The objects are not the same.");
```

only the addresses of the objects are compared.

The equals Method

- Instead of using the == operator to compare two `Stock` objects, we should use the `equals` method.

```
public boolean equals(Stock object2)
{
   boolean status;

   if(symbol.equals(Object2.symbol) && sharePrice == Object2.sharePrice)
      status = true;
   else
      status = false;
   return status;
}
```

- Now, objects can be compared by their contents rather than by their memory addresses.

Methods That Copy Objects

- There are two ways to copy an object.
  - You cannot use the assignment operator to copy reference types
  - Reference only copy (shallow Copy)
    - This is simply copying the address of an object into another reference variable.

```
Stock stock1 = new Stock("GMX", 55.3);
Stock stock2 = stock1;
```

Deep copy (correct)

- This involves creating a new instance of the class and copying the values from one object into the new object.

Copy Constructors

- A copy constructor accepts an existing object of the same class and clones it

```
public Stock(Stock object2)
{
 if (object2 == null) //Not a real stock.
   {
     System.out.println("Fatal Error.");
     System.exit(0);
   }
      this.symbol = object2.symbol;
      this.sharePrice = object2.sharePrice;
}

// Create a Stock object
Stock company1 = new Stock("XYZ", 9.62);

//Create company2, a copy of company1
Stock company2 = new Stock(company1);
```

Important Points

- Calling object is always present in the function
- If another object is required for the operation of a method , we need to pass it through the argument.

- Class name is a user defined type.
- Class references can be used as function argument
- Class references can be returned from Functions

- Object is a composite entity
- Do not apply any arithmetic and logical operation on object name directly.

END