Object Oriented Concepts

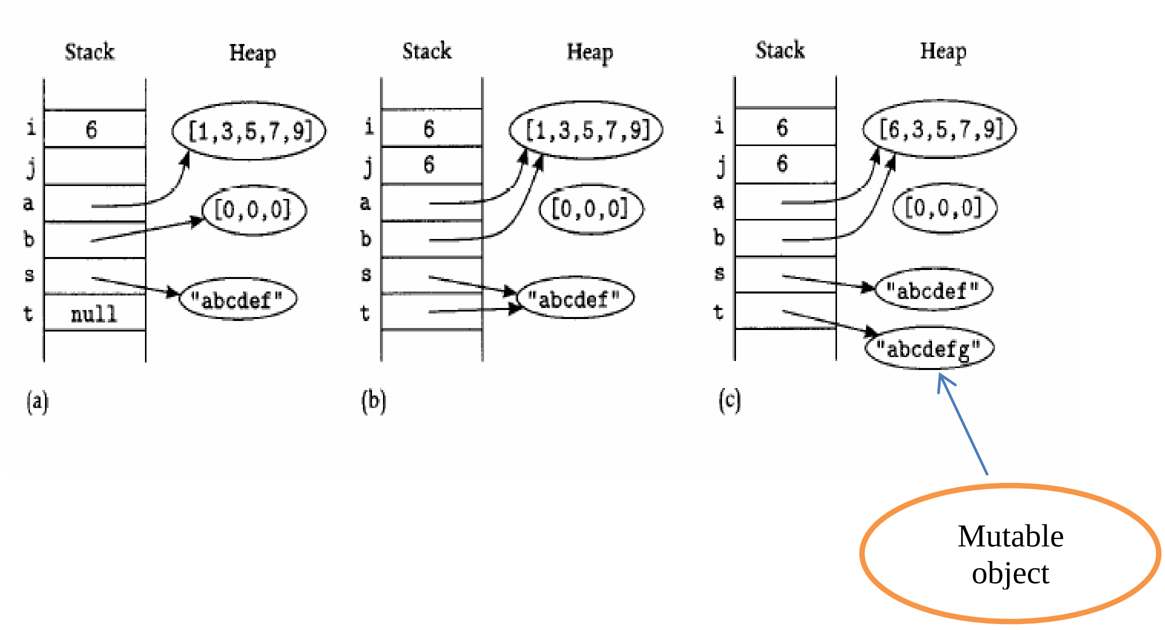
1.1. Mutability

All objects either are immutable or *mutable*. **The state of an immutable object never changes**, while **the state of a mutable object can change.**

String object are mutable, for example String methods have a concatenation operator +, but is does not modify either of its argument, instead, it returns a new string whose state is the concatenation of the states of its arguments. If we did the following assignment to t with the state shown in Figure (b).

t=t + "g";

the results as shown in figure (c) is that t now refers to an a new **String** object whose state is "abcdefg" and the object referred to by s is unaffected.



On the other hand, arrays are mutable. The assignment a[i]=e/// where e element is equal 6

causes the state of array a to change by replacing its ith element with the value obtained by evaluating expression e

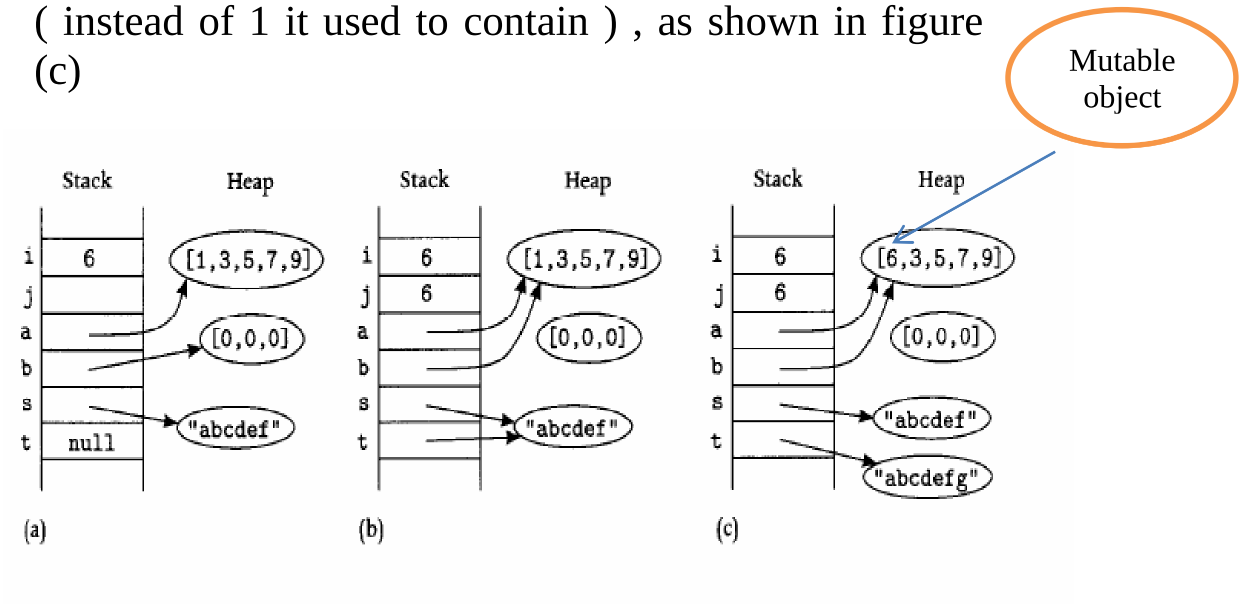
(the modification occurs only if i is in bounds for a, an exception is thrown otherwise).

Discussion Note

If a mutable object is shared by two or more variables, modifications made through one of the variables will be visible when the object is used through the other variable.

Example

Suppose the shared array in figure (b) is modified by
b[0]=i; // i= 6 before
this causes the zeroth element of the array to contain 6 (instead of 1 it used to contain) , as shown in figure (c)



1.2. Method Call Semantic

An attempt to call a method as in this form:
e. m (), // where e is a representative of a class and m is a method
first evaluates **e** to obtain the class of object whose method is being called .
Then the expression for the arguments are evaluated to obtain *actual parameter values المتغيرات الفعلية*. This evaluation happen left to right.

Next an *activation record* is created for the call and pushed onto the stack; the activation record contains room for the *formal parameters* المعاملات الشكلية of the method (the formals are the variables declared in the method header) and any other local storage the method required.
When the actual parameter are *assigned* to the formals , this kind of parameters passing is called *"call by value"* and
Finally control is dispatched to the called method e.m.

Note: if the actual parameters value is a reference to an object, that reference is assigned to the formal . this means that the called procedure shares object with its caller. Furthermore, if these object are mutable, and the called procedure change their state, these changes are visible to the caller when it returns.

Example

```
public static void multiples (int [ ] a, int m) {
   if (a == null) return;
   for (int i = 0; i < a.length; i++) a[i] = a[i]*m;
}
```

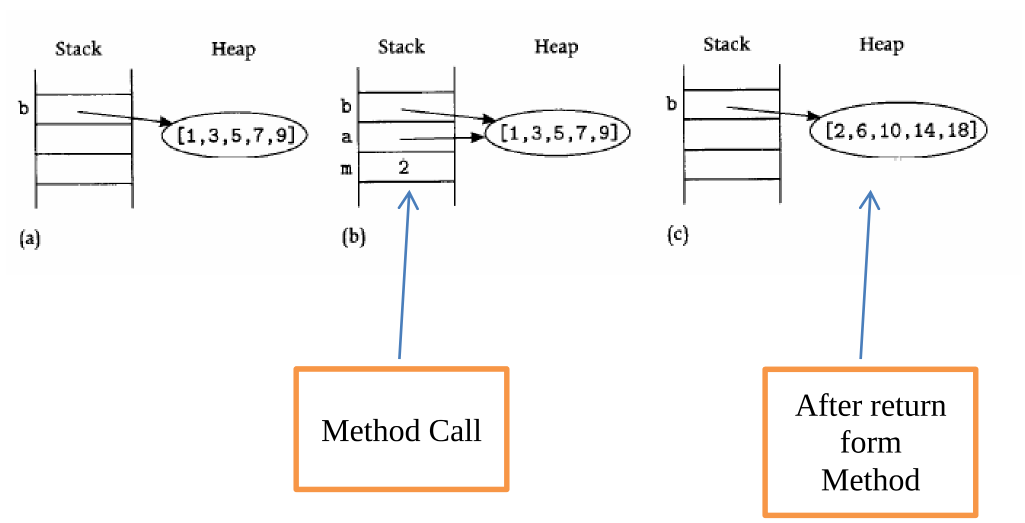
And in the main class, we define

```
int [ ] b = {1,3,5,7,9};
```

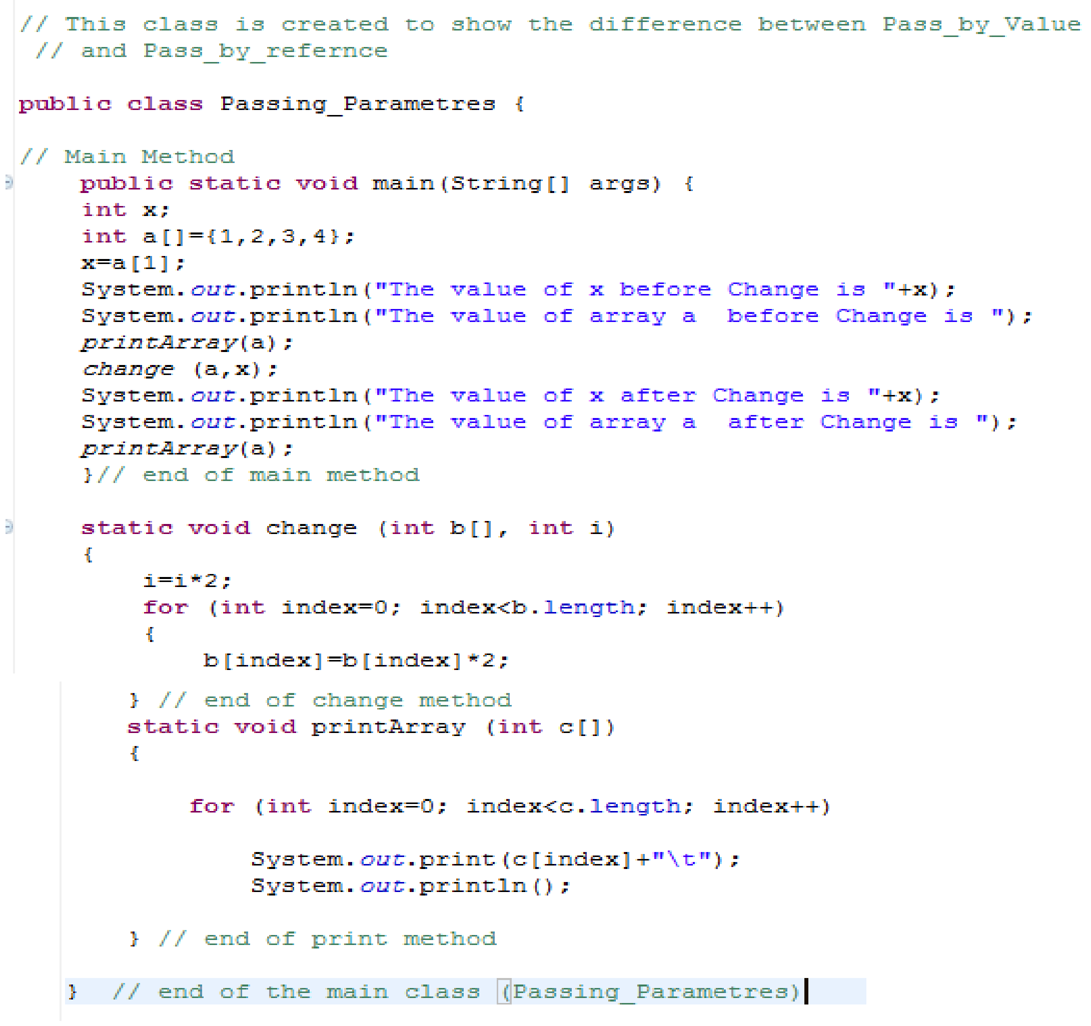
And we call the method

```
Arrays.multiples(b, 2);
```

The results in the Stack storage are:



Example 2



```
// This class is created to show the difference between Pass_by_Value
 // and Pass_by_refernce

public class Passing_Parametres {

// Main Method
    public static void main(String[] args)  {
    int x;
    int a[]={1,2,3,4};
    x=a[1];
    System.out.println("The value of x before Change is "+x);
    System.out.println("The value of array a  before Change is ");
    printArray(a);
    change (a,x);
    System.out.println("The value of x after Change is "+x);
    System.out.println("The value of array a  after Change is ");
    printArray(a);
    }// end of main method

    static void change (int b[], int i)
    {
        i=i*2;
        for (int index=0; index<b.length; index++)
        {
            b[index]=b[index]*2;

       } // end of change method
       static void printArray (int c[])
       {

           for (int index=0; index<c.length; index++)

               System.out.print(c[index]+"\t");
               System.out.println();

       } // end of print method

   }  // end of the main class (Passing_Parametres)
```

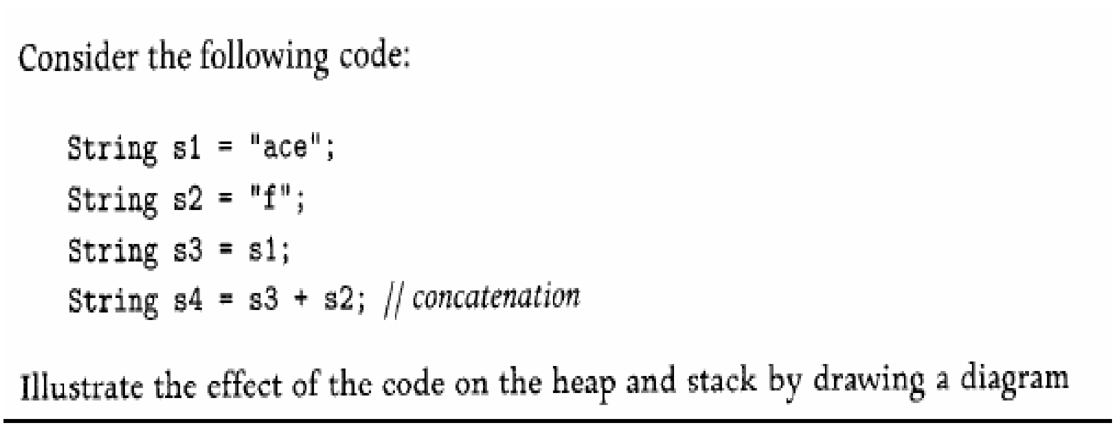
Example 2

```
public class CallsByRefers {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		CallsByRefers cc=new CallsByRefers();
		int [] b= {0,2,3,4};
		int kl=9;
		cc.change(b, kl);
		for (int o=0; o<b.length;o++)
		{
			System.out.println(b[o]);
			}
		System.out.println("The Value of kl after return from 
		method "+kl);
		}


	public int change (int [] a, int j){
		j = j*2;
		System.out.println("The value of i inside the method 
		"+j);
		for (int k=0; k<a.length;k++)
		{
		 a[k]=a[k]*2;

		}
			return j;
				}// End Main Method


					}// End CallsByRefers Class
```

The Output for Example 2 is :-

```
The value of i inside the method 18
0
4
6
8
The Value of kl after return from method 9
```

Quiz #1



Consider the following code:

```
    String s1 = "ace";
    String s2 = "f";
    String s3 = s1;
    String s4 = s3 + s2; // concatenation
```

Illustrate the effect of the code on the heap and stack by drawing a diagram