The Object Class

The Object Class is the root class of all classes in Java. Every class that you create inherits from the Object class implicitly (this means that you do not have to write “extends Object” while defining your class).
The Object class is a part of the java.lang package, and has some of the following methods:

1) clone()
2) getClass()
3) toString()
4) hashCode()
5) equals()

The functionality of these methods has been explained in the following code:

```
import java.lang.*;
import java.util.*;
public class Employee{

    private int id;
    private String name;

    public int getID(){
        return id;
    }
    public void setID(int id){
        this.id = id;
    }
    // getter and setter methods
    public String getName(){
        return name;
    }
    public void setName(String name){
        this.name = name;
    }
    public Employee(int id, String name){
        super();
        this.id = id;
        this.name = name;
    }
```

```
    public static void main(String[] args){
        Employee emp = new Employee(14, "Raman");
        Employee emp2 = new Employee(29, "Shreya");
        Employee emp3 = new Employee(14, "Raman");
        Employee emp4 = emp;
        System.out.println(emp.toString()); // Statement-1
        System.out.println(emp.equals(emp2)); // Statement-2
        System.out.println(emp.equals(emp3)); // Statement-3
        System.out.println(emp.equals(emp4)); //Statement-4
    }
}
```

a) What is the output returned by the toString() method in Statement-1? Now override the toString() method so that Statement-1 returns the ID and name of the employee.

b) Observe the outputs for statements 2-4, especially statement-3. The output for this would be false. Despite having the same values of attributes, why is the equals() method returning false?

c) Now override the equals() method, so that the print statement in statement-3 returns true as answer (please ask the students to ignore the try-catch block used in the solution of this part).

d) Now check the overridden equals() method that you have written. You would probably have used the equals() method itself to compare the names of the Employees (Strings). Obviously, since the answer in part c is now true, this equals method MUST be returning true. Why is it that this comparison of names using equals() returns true, but using equals() originally in part-b for statement-3 returned false?

e) Note to TAs: Kindly briefly discuss the difference between equals() method and == operator, by giving examples of the String class.

Solution:

a) Overridden toString() method:

```
@Override
    public String toString(){
        return "Employee ID: "+id+"\nEmployee Name: "+name;
    }
```

b) false
false
true, Statement-3 returns false because in this case, equals() method is doing comparison by reference, NOT by values.

c) Overridden equals() method:

```
@Override
    public boolean equals(Object obj){
        try{
            Employee emp = (Employee)obj;
            if(name.equals(emp.getName()) && id==emp.getID()){
                return true;
            }
            else{
                return false;
            }
        }
        catch(Exception ex){
            return false;
        }
    }
```

d) In the String class defined by Java, the equals() method has already been overwritten to compare 2 strings by values. If you want to compare 2 strings by reference, you can use the == operator.

Object cloning:

Now we will modify the above program to implement the Cloneable interface, so that we can clne class objects. Note that the Cloneable interface is a marker interface, that is, it does not define any methods that you have to override.

```
import java.lang.*;
import java.util.*;
public class Employee implements Cloneable{

    private int id;
```

```
    private String name;

    public int getID(){
        return id;
    }
    public void setID(int id){
        this.id = id;
    }
    // getter and setter methods
    public String getName(){
        return name;
    }
    public void setName(String name){
        this.name = name;
    }
    public Employee(int id, String name){
        //super();
        this.id = id;
        this.name = name;
    }
    public static void main(String[] args) throws
CloneNotSupportedException{
        Employee emp = new Employee(29, "Singh");
        Employee emp2 = (Employee) emp.clone();
        System.out.println(emp.getID()+" "+emp.getName());
        System.out.println(emp2.getID()+" "+emp2.getName());
    }


}
```

Question 2:

a) Edit the name and ID of employee 1 using the setter methods of the class. Now compare this employee with employee 2. Are these changes reflected in the status of employee 2?

b) If employee 2 had been created as Employee emp2 = emp (assignment method) instead of clone() method, then would changes in attributes of employee 1 be reflected in attributes of employee 2?

c) Now we will consider the concepts of shallow and deep cloning. If a class contains reference variables of another class, then shallow copy will NOT create an independent clone object. For example, consider the code below:

```
import java.lang.*;
import java.util.*;

class Address{
    private int houseNo;

    public int getHouseNo(){
        return houseNo;
    }
    public void setHouseNo(int num){
        this.houseNo = num;
    }
    public Address(int houNo){
        this.houseNo = houNo;
    }
}
public class Employee implements Cloneable{

    private int id;
    private String name;
    private Address address;
    public int getID(){
        return id;
    }
    public void setID(int id){
        this.id = id;
    }
    // getter and setter methods
    public String getName(){
        return name;
    }
    public void setName(String name){
        this.name = name;
    }
    public Address getAddress(){
        return address;
    }
```

```
    public void setAddress(Address address){
        this.address = address;
    }
    public Employee(int id, String name, Address address){
        //super();
        this.id = id;
        this.name = name;
        this.address = address;
    }
    public static void main(String[] args) throws
CloneNotSupportedException{
        Address adr1 = new Address(311);
        Employee emp = new Employee(29, "Singh", adr1);
        Employee emp2 = (Employee) emp.clone();

        System.out.println(emp.getID()+" "+emp.getName()+"
"+emp.getAddress().getHouseNo());
        System.out.println(emp2.getID()+" "+emp2.getName()+"
"+emp2.getAddress().getHouseNo());
    }


}
```

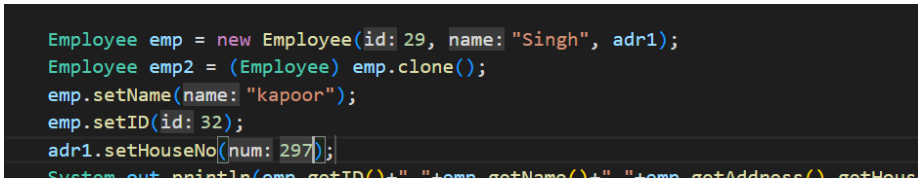
Now if you change all attributes for employee 1, does the address for employee 2 change or remain the same?

d) Now, override the clone() method of the Object class, so that on changing address of employee 1, the change is NOT reflected in address of employee 2.

Solution of Q2:

a) If you change the name and ID of employee 1 using setter methods, the change is NOT reflected in the name and ID of the cloned object.
b) Yes, in this case, the name and ID of employee 2 would also be affected since both are referencing the same object.

c)



```
Employee emp = new Employee(id: 29, name: "Singh", adr1);
Employee emp2 = (Employee) emp.clone();
emp.setName(name: "kapoor");
emp.setID(id: 32);
adr1.setHouseNo(num: 297);
```

If attributes are changed like this, both emp and emp2 will have the same address, though name and ID will be different. In this case, we call emp2 a shallow copy of emp.

d)

```
@Override
protected Object clone() throws CloneNotSupportedException{
    Address addr = new Address(address.getHouseNo());
    Employee emp = new Employee(id, name, addr);
    return emp;
}
```

If the clone() method is overridden in this manner, then change in address of employee 1 will NOT be reflected in address of employee 2. This is called a deep copy, or deep cloning.

Note that if your aim is to clone a certain object to capture its status at that point, and you modify the reference variables afterwards, then you should override the clone() method to create a deep copy, so that those modifications do NOT appear in the cloned object.

This is the difference between shallow and deep cloning.