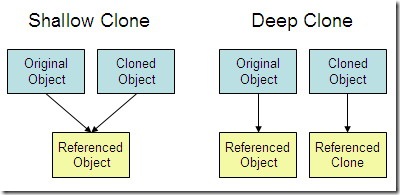
Shallow Copy VS Deep Copy in Java – 2024

**Shallow Copy Vs Deep Copy**



**What is Shallow Copy?**

Shallow copy is a mirror object where both original object and mirror object refer to the same referenced object.

Here is an example of Shallow Copy implementation

**public** **class** Employee **implements** Cloneable {

**private** String name;

**private** Address adrs;

**public** Employee(String name, Address adrs) {

**this**.name = name;

**this**.adrs = adrs;

}

// getter/setter methods

**public** Object clone() {

//shallow copy

**try** {

**return** **super**.clone();

} **catch** (CloneNotSupportedException e) {

**return** **null**;

}

}

@Override

**public** String toString() {

**return** "Employee{" +

"name='" + name + '\'' +

", adrs=" + adrs +

'}';

}

}

**public** **class** Address {

**private** String cityName;

**public** Address(String cityName) {

**this**.cityName = cityName;

}

// getter/setter methods

@Override

**public** String toString() {

**return** "Address{" +

"cityName='" + cityName + '\'' +

'}';

}

}

Real Time Example: Twin brothers having the same address.

**public class** ShallowCopyTest {  
 Employee actualEmp = **new** Employee("John", **new** Address("New York"));

System.***out***.println("Original Employee: " + actualEmp);//Employee{name='John', adrs=Address{cityName='New York'}}

//Clone Object

Employee clonedEmp = (Employee) actualEmp.clone();

System.***out***.println("Cloned Employee: " + clonedEmp);//Employee{name='John', adrs=Address{cityName='New York'}}

System.***out***.println("actualEmp == clonedEmp : "+ (actualEmp == clonedEmp)); //false

System.***out***.println("actualEmp.equals(clonedEmp) : "+ (actualEmp.equals(clonedEmp))); //false

// Modify actual emp object

actualEmp.setName("Dan");

actualEmp.getAdrs().setCityName("New Jersey");

**// After modifying reference object from the actual object**

System.***out***.println("Now Original Employee: " + actualEmp); //Employee{name='Dan', adrs=Address{cityName='New Jersey'}}

System.***out***.println("Now Cloned Employee: " + clonedEmp); //Employee{name='Dan', adrs=Address{cityName='New Jersey'}}

// **After modifying reference object from the cloned object**

clonedEmp.setName("Smith Clone");

clonedEmp.getAdrs().setCityName("Texas");

System.***out***.println("Original Employee: " + actualEmp); //Employee{name='Dan', adrs=Address{cityName='Texas'}}

System.***out***.println("Cloned Employee : " + clonedEmp); //Employee{name='Smith Clone', adrs=Address{cityName='Texas'}}

}  
}

It is clear from the above code that since in case of shallow copy, reference object from both actual object and cloned object refer the same memory location. It means, either actual object or cloned object modifies the reference object, the reference object will have the modified data for both.

**What is Deep Copy?**

**In case of deep copy, we create a clone that is independent of the original object and making changes in the cloned object should not affect the original object.**

**How many ways to create Deep Copy**

1. **Using a special implementation of clone method.**
2. **Using Copy constructor**
3. **Using Serialization-Deserialization**

**Now lets see how to implement them in java.**

**Special Implementation of clone method**

**public** Object clone() {

//Deep copy

**try** {

Address address = **new** Address(**this**.adrs.getCityName());

Employee emp = **new** Employee(**this**.name, address);

**return** emp;

} **catch** (Exception e) {

**return** **null**;

}

}

**Copy Constructor**

// Copy Constructor

**public** Employee(Employee emp) {

**this**.name = emp.name;

**this**.adrs = **new** Address(emp.adrs.getCityName());

}

**How to use**

Employee actualEmp = **new** Employee("John", **new** Address("New York"));

System.out.println("Original Employee: " + actualEmp);//Employee{name='John', adrs=Address{cityName='New York'}}

//Clone Object using copy constructor

**Employee clonedEmp = new Employee(actualEmp);**

**Using Serialization-Deserialization**

In case of serialization and deserialization, ensure that object implements Serializable interface.

**public static byte**[] serializeToByteArray(Employee emp) {  
 **try** (ByteArrayOutputStream bos = **new** ByteArrayOutputStream();  
 ObjectOutputStream oos = **new** ObjectOutputStream(bos)) {  
 oos.writeObject(emp);  
 **return** bos.toByteArray();  
 } **catch** (IOException e) {  
 **throw new** RuntimeException(e);  
 }  
}

public static Object deserializeFromByteArray(byte[] objectBytes) {

try (ByteArrayInputStream bin = new ByteArrayInputStream(objectBytes);

ObjectInputStream oin = new ObjectInputStream(bin)) {

return oin.readObject();

} catch (IOException | ClassNotFoundException e) {

throw new RuntimeException(e);

}

}

**How to use**

Employee actualEmp = **new** Employee("John", **new** Address("New York"));

System.out.println("Original Employee: " + actualEmp);//Employee{name='John', adrs=Address{cityName='New York'}}

// Clone using serialize and deserialize

**byte**[] empBytes = Employee.serializeToByteArray(actualEmp);

Employee clonedEmp = (Employee) Employee.deserializeFromByteArray(empBytes);

Test Program   
**public class** TestDeepCopy {  
  
 **public** **static** **void** main(String[] args) {

// After Deep Cloning

Employee actualEmp = **new** Employee("John", **new** Address("New York"));

System.***out***.println("Original Employee: " + actualEmp);//Employee{name='John', adrs=Address{cityName='New York'}}

// Clone using serialize and deserialize

**byte**[] empBytes = Employee.*serializeToByteArray*(actualEmp);

Employee clonedEmp = (Employee) Employee.*deserializeFromByteArray*(empBytes);

//Clone Object using copy constructor

// Employee clonedEmp = new Employee(actualEmp);

// Employee clonedEmp = (Employee) actualEmp.clone();

System.***out***.println("Cloned Employee: " + clonedEmp);//Employee{name='John', adrs=Address{cityName='New York'}}

System.***out***.println("actualEmp == clonedEmp : "+ (actualEmp == clonedEmp)); //false

System.***out***.println("actualEmp.equals(clonedEmp) : "+ (actualEmp.equals(clonedEmp))); //false

// Modify actual emp object

actualEmp.setName("Dan");

actualEmp.getAdrs().setCityName("New Jersey");

// After modifying reference object from the actual object

System.***out***.println("Now Original Employee: " + actualEmp); //Employee{name='Dan', adrs=Address{cityName='New Jersey'}}

System.***out***.println("Now Cloned Employee: " + clonedEmp); //Employee{name='John', adrs=Address{cityName='New York'}}

// After modifying reference object from the cloned object

clonedEmp.setName("Smith Clone");

clonedEmp.getAdrs().setCityName("Texas");

System.***out***.println("Original Employee: " + actualEmp); //Employee{name='Dan', adrs=Address{cityName='New Jersey'}}

System.***out***.println("Cloned Employee : " + clonedEmp); //Employee{name='Smith Clone', adrs=Address{cityName='Texas'}}

}

}

**Real Time Example: Twin Sister having different address after marriage.**

# **Why Clone method is protected**

1. **All objects need not be cloned**
2. **The method is protected because you shouldn't call it on object**, **you should override it as public**.
3. If Object declared clone() as public, then every class would have to have a public clone() method and every subclass will inherit the clone method which will pollute the intent of the Marker Interface API.

**The essence of marker interface is whether it is doable or not. As an example,** In the corporate world, some management folks say whether it is doable or not, but it is not necessary that we have to do it. Punishable Act, Cognizable offence.

In case of Object class, it is defined as

protected native Object clone() throws CloneNotSupportedException;