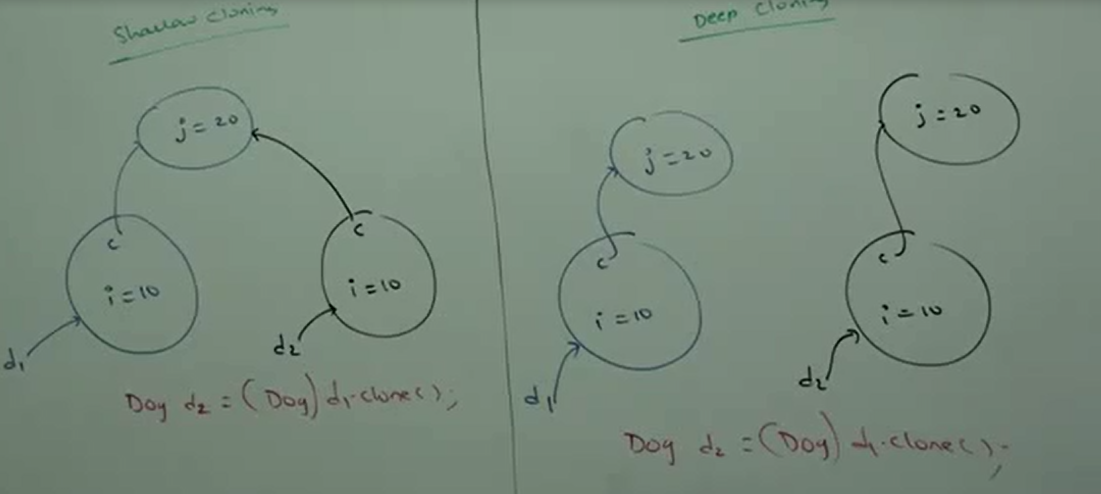
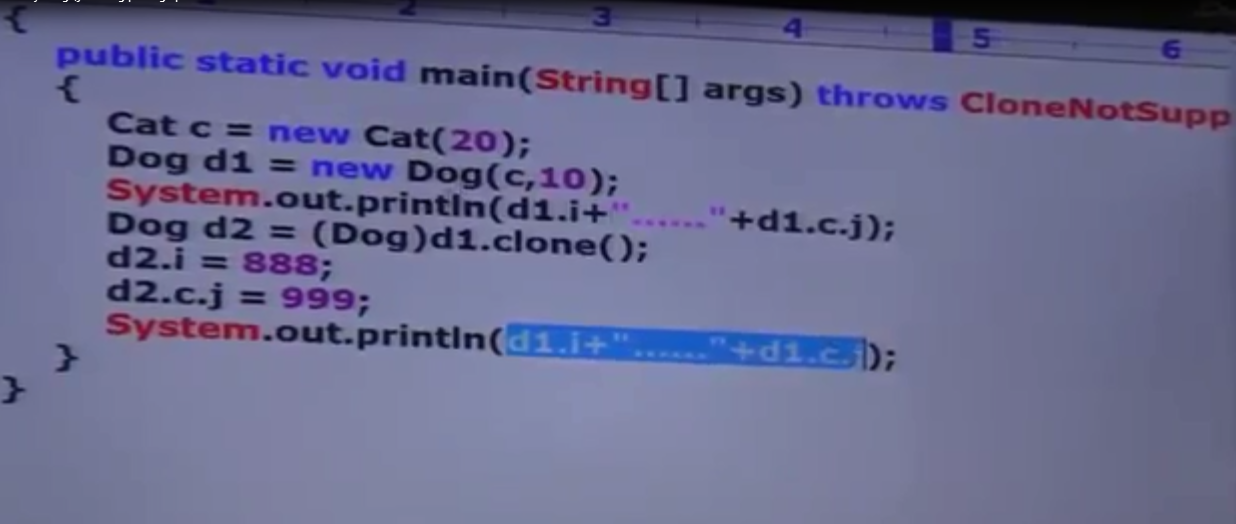
To understand Immutability we need to understand shallow cloning vs deep cloning

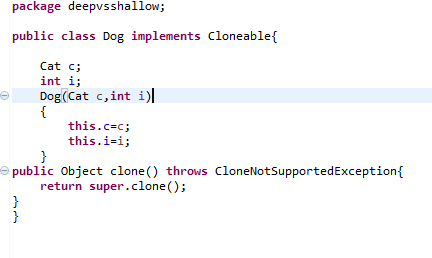
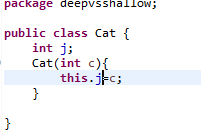
When an object is cloned using .clone() keyword a duplicate copy is created and the duplicate copy contains copies of the primitive datatype. But if the original object contains any instance variable(not primitive type) only the reference is copied to the duplicate object .And the original and the duplicate object contains different reference variable pointing to the same instance of contained object.



The variable c here is not a primitive type but reference to an instance object.

This process is called shallow cloning and the copy is called bitwise copy.

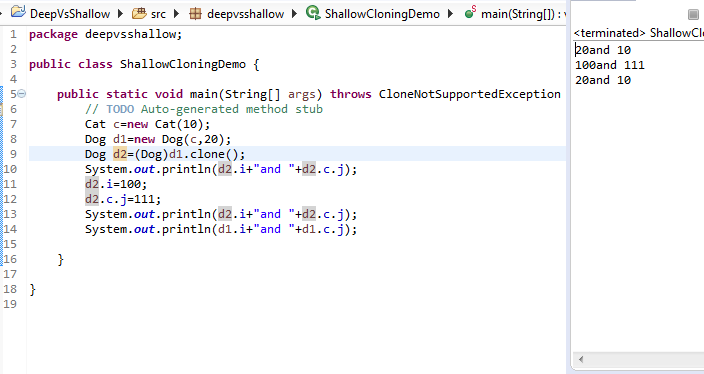


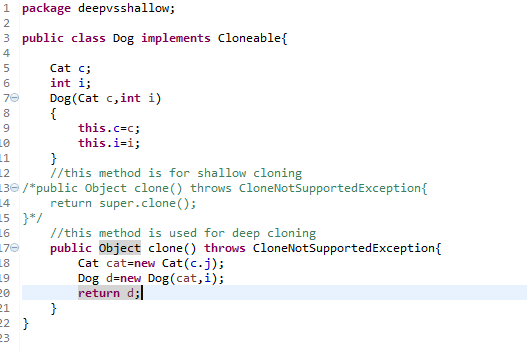


Shallow Cloning is not recommended as modification in the duplicate copy contained object results in modification of the original object and vice versa. In shallow cloning,by using cloned object reference if we perform any change in the contained object,the change will be reflected to the main object.

Deep Cloning: The process of creating exactly duplicate independent copy including contained object is called deep cloning. In deep cloning if the main object contains any primitive variables then in the cloned object duplicate copies will be created. If the main object contains any reference variable then corresponding contained object will be created in the duplicate copy.

Object class clone() method is used for shallow cloning.





If object contains reference variables deep cloning is best choice else we should go for shallow cloning.

To create immutable class in java, you have to do following steps.

1. Declare the class as final so it can’t be extended.
2. Make all fields private so that direct access is not allowed.
3. Don’t provide setter methods for variables
4. Make all **mutable fields final** so that it’s value can be assigned only once.
5. Initialize all the fields via a constructor performing deep copy.
6. Perform cloning of objects in the getter methods to return a copy rather than returning the actual object reference.

To understand points 4 and 5, let’s run the sample Final class that works well and values doesn’t get altered after instantiation.

package com.journaldev.java;

import java.util.HashMap;

import java.util.Iterator;

public final class FinalClassExample {

private final int id;

private final String name;

private final HashMap<String,String> testMap;

public int getId() {

return id;

}

public String getName() {

return name;

}

/\*\*

\* Accessor function for mutable objects

\*/

public HashMap<String, String> getTestMap() {

//return testMap;

return (HashMap<String, String>) testMap.clone();

}

/\*\*

\* Constructor performing Deep Copy

\* @param i

\* @param n

\* @param hm

\*/

public FinalClassExample(int i, String n, HashMap<String,String> hm){

System.out.println("Performing Deep Copy for Object initialization");

this.id=i;

this.name=n;

HashMap<String,String> tempMap=new HashMap<String,String>();

String key;

Iterator<String> it = hm.keySet().iterator();

while(it.hasNext()){

key=it.next();

tempMap.put(key, hm.get(key));

}

this.testMap=tempMap;

}

/\*\*

\* Constructor performing Shallow Copy

\* @param i

\* @param n

\* @param hm

\*/

/\*\*

public FinalClassExample(int i, String n, HashMap<String,String> hm){

System.out.println("Performing Shallow Copy for Object initialization");

this.id=i;

this.name=n;

this.testMap=hm;

}

\*/

/\*\*

\* To test the consequences of Shallow Copy and how to avoid it with Deep Copy for creating immutable classes

\* @param args

\*/

public static void main(String[] args) {

HashMap<String, String> h1 = new HashMap<String,String>();

h1.put("1", "first");

h1.put("2", "second");

String s = "original";

int i=10;

FinalClassExample ce = new FinalClassExample(i,s,h1);

//Lets see whether its copy by field or reference

System.out.println(s==ce.getName());

System.out.println(h1 == ce.getTestMap());

//print the ce values

System.out.println("ce id:"+ce.getId());

System.out.println("ce name:"+ce.getName());

System.out.println("ce testMap:"+ce.getTestMap());

//change the local variable values

i=20;

s="modified";

h1.put("3", "third");

//print the values again

System.out.println("ce id after local variable change:"+ce.getId());

System.out.println("ce name after local variable change:"+ce.getName());

System.out.println("ce testMap after local variable change:"+ce.getTestMap());

HashMap<String, String> hmTest = ce.getTestMap();

hmTest.put("4", "new");

System.out.println("ce testMap after changing variable from accessor methods:"+ce.getTestMap());

}

}

Output of the above immutable class in java example program is:

Performing Deep Copy for Object initialization

true

false

ce id:10

ce name:original

ce testMap:{2=second, 1=first}

ce id after local variable change:10

ce name after local variable change:original

ce testMap after local variable change:{2=second, 1=first}

ce testMap after changing variable from accessor methods:{2=second, 1=first}

Now let’s comment the constructor providing deep copy and uncomment the constructor providing shallow copy. Also uncomment the return statement in getTestMap() method that returns the actual object reference and then execute the program once again.

Performing Shallow Copy for Object initialization

true

true

ce id:10

ce name:original

ce testMap:{2=second, 1=first}

ce id after local variable change:10

ce name after local variable change:original

ce testMap after local variable change:{3=third, 2=second, 1=first}

ce testMap after changing variable from accessor methods:{3=third, 2=second, 1=first, 4=new}

As you can see from the output, HashMap values got changed because of shallow copy in the constructor and providing direct reference to the original object in the getter function.