**Cloning in Java**

In Java, when you assign an object to another variable, only the memory address of the object is copied and hence any changes in the original object will be reflected in the new variable.

MainObject obj1 = new MainObject();

obj2 = obj2;

Here, in this case any changes you make to obj1 will reflect in obj2 and vice versa. Well, if this is what you want, then no issues but if you don’t want the change made in obj2 to be seen in obj1 or any change made in obj1 to be seen in obj2, then cloning comes into picture.

*So what is cloning?*

Well, cloning means creating a copy of the object. The precise meaning of "copy" may depend on the class of the object. The general intent is that, for any object x, the expression:

x.clone() != x

Will be true and that the expression:

x.clone().getClass() == x.getClass()

will be true, but these are not absolute requirements. While it is typically the case that:

x.clone().equals(x)

will be true, and this is also not an absolute requirement.

Let me explain what these rules mean, "x.clone() != x" means, the clones object should not hold the same memory address of the original object. And "x.clone().getClass() == x.getClass()" means, the cloned object and the original object should be of the same type. Well again why its said that these are not absolute requirements because, when you shallow copy the object and if the object hold any references to someother object, then the references are copied. Yes, if you go for a deep copy, the above two rules will satisfy. Now lets see the thrid one, "x.clone().equals(x)" means the cloned object data should be equal to the original one but again this is not absolute requirement. The cloned copy data or the original object data can change any time.

*How do you implement cloning in java?*

Now, getting into it. In java, to do cloning, you must implement clone() method. Lets start with an example code.

public class Color {

private String color;

public Color(String c){

this.color = c;

}

//getters and setters for the fields should go here........

}

public class ColoredCircle implements Cloneable {

private int centerX;

private int centerY;

private Color color;

public ColoredCircle(int x, int y, Color c){

this.centerX = x;

this.centerY = y;

this.color = c;

}

public Object clone() {

try {

return (ColoredCircle)super.clone();

}

catch (CloneNotSupportedException e) {

// This should never happen

}

}

//getters and setters for the fields should go here........

}

public class CloneMain {

public static void main(String [] args) {

Color c = new Color("RED");

ColoredCircle c1 = new ColoredCircle(200,200,c);

ColoredCircle c2 = c1.clone();

}

}

To implement cloning in your class, first, make your object implement Cloneable and then set up the implementation of clone() method with a try-catch block for CloneNotSupportedException. Ok. What will happen when you implement this way? Since the class implements the "Cloneable" interface, it registers itself to be ready for copying itself. So when the object calls "clone()" method, a copy of the object is returned. If you observe here, we are calling "super.clone()" inside clone() method. Because clone() is declared in class Object, it is inherited by every Java object and hence calling super.clone() copies your superclass's fields, and makes bitwise copies of your fields. This is nothing but shallow copy which means when you copy the "ColoredCircle" using clone() method, the fields x and y are copied perfectly with values but field "color" is copied by reference. So any changes you make to the color of the original object will be reflected in the copied object and vice versa. The solution for this is deep copy. Change the Color class to implement "Cloneable" interface and clone() method and call the clone() method of color object inside the clone() method of the ColoredCircle object. Look at the changed Color class and ColoredCircle class below.

public class Color implements Cloneable{

private String color;

public Color(String c){

this.color = c;

}

public Object clone() {

try {

return (Color)super.clone();

}

catch (CloneNotSupportedException e) {

// This should never happen

}

}

//getters and setters for the fields should go here........

}

public class ColoredCircle implements Cloneable {

private int centerX;

private int centerY;

private Color color;

public ColoredCircle(int x, int y, Color c){

this.centerX = x;

this.centerY = y;

this.color = c;

}

public Object clone() {

ColoredCircle coloredCircle = null;

try {

coloredCircle = (ColoredCircle)super.clone();

}

catch (CloneNotSupportedException e) {

// This should never happen

}

coloredCircle.color = (Color) color.clone();

return coloredCircle;

}

//getters and setters for the fields should go here........

}

To know more about Shallow Copy and Deep Copy, check it out [here](http://www.jusfortechies.com/java/core-java/deepcopy_and_shallowcopy.php) or go to "[Deep Copy/Shallow Copy](http://www.jusfortechies.com/java/core-java/deepcopy_and_shallowcopy.php)" article. Ok. Now lets get back to know more about Cloneable interface. The Cloneable interface is a marker interface which means it dont have any methods. So after you implement Cloneable interface, if you dont write clone() method in your class,nothing will happen. The purpose of Cloneable interface is to tell JVM that this class is eligible for cloning. Note that the class Object doesn't implement Cloneable interface, hence if you call the Object class "clone()" method without implementing Cloneable interface, a CloneNotSupportedException will be thrown.

Generally when you invoke clone() method in your object, it should either:

* 1. return an Object reference to a copy of the object upon which it is invoked, or
* 2. throw CloneNotSupportedException

One more point. It is not mandatory to implement clone() method in your class when you want to clone the object. If you dont, since the clone() method in class Object is declared protected, only subclasses and members of the same package will be able to invoke clone() on the object. Thats why to enable any class in any package to access the clone() method, you'll have to override it and declare it public.

*Let me tell you couple of gotchas with Cloneable interface*

1. When you dont know whether you can call the clone() method of a particular class as you are not sure if it is implemented in that class, you can check with checking if the class is instance of "Cloneable" interface as below.

if(obj1 instanceof Cloneable){

obj2 = obj1.clone();

}

2. As Cloneabe interface is a marker interface (dont have any methods in it), you can never cast an object to Cloneable and call clone() method. Check below.

//Dont do this. Cloneabe dont have any methods

obj2 = (Cloneable)obj1.clone();

*Well, clone has its own problems*

First one is that no constructor is called on the object being cloned. As a result, it is your responsibility, to make sure all the members have been properly set. Lets consider a class keeping track of the total number of objects of that type, using a static int member. Generally in the constructors you would increase the count. But, when you clone the object, since no constructor is called, the count will not get updated and wont reflect the correct number of objects.

Second one is, if the class has final fields, these canï¿½t be given a value in the clone method. This leads to problems with properly initializing the objectï¿½s final fields. If the final field is referring to some internal state of the object, then the cloned object ends up sharing the internal state and this surely is not correct for mutable objects.

The solution for these problems is "copy-constructor".

*What is copy-constructor?*

A copy constructor is one that takes object of its own type as a single parameter.

public class Person

{

private Car car;

private String name;

public Person(Car c, String n)

{

car = c;

name = n;

}

public Person(Person p)

{

name = p.name;

car = new Car(p.car);

// we assume we have a copy constructor for Car

}

public String toString()

{

return "This is person has " + car;

}

}

public class Car

{

public Car() {}

}

public class CopyConstructorMain {

public static void main(String [] args) {

Person person1 = new Person(new Car(), "Ben");

Person person2 = new Person(person1);

System.out.println(person1);

System.out.println(person2);

}

}

Output is:

----------

This is person has Car@4edbb0

This is person has Car@3f75e0

In this example, in class Person, "public Person(Person p)" is the copy constructor. This is used to create a copy of person1 object. The copy constructor worked perfectly fine in this exmaple. But if the variable "car" in Person class is actually an instance of a subclass ï¿½ then you will get the wrong answer; what is called a slice. Check below example and you will understand what I meant by a problem. Lets say you have a subclass of Car - "Mitsubishi".

public class Mitsubishi extends Car

{

public Mitsubishi() {}

}

public class CopyConstructorMain {

public static void main(String [] args) {

Person person1 = new Person(new Mitsubishi(), "Ben");

Person person2 = new Person(person1);

System.out.println(person1);

System.out.println(person2);

}

}

Output is:

----------

This is person has Mitsubishi@2e4560

This is person has Car@3f34d4

The expected answer in this example is "Mitsubishi" object for both Person objects but you got the wrong answer. One way, to fix this is check the instance of the object and type cast accordingly.

public Person(Person p)

{

name = p.name;

if (p.car instanceof Mitsubishi)

car = new Mitsubishi((Mitsubishi)p.car);

else

car = new Car(p.car);

}

Now this is not a perfect fix, as the code needs to be modified if another type of Car (a derived class of Car) is introduced in the system.

A good way to copy an object is to let the object do it. In other words, rather than the Person creating a Car (using another instance of Car), i.e., new Car(car);, why not ask the car to make a copy of itself?

public class Car

{

public Car() {}

public Car(Car c) {...} // Assume proper copying of the Car

}

public class Mitsubishi extends Car

{

public Mitsubishi() {}

public Mitsubishi(Mitsubishi m) {...}

}

And the best way to fix this is combine copy-constructor and clone. This will work perfectly.

public class Person implements Cloneable

{

private final Car car;

private String name;

public Person(Car c, String n)

{

car = c;

name = n;

}

public Person(Person p)

{

Car instCar = null;

try

{

instCar = (Car) p.car.clone();

}

catch(CloneNotSupportedException e) {}

car = instCar;

name = p.name;

}

public Object clone()

{

return new Person(this);

}

public String toString()

{

return "This is person has " + car;

}

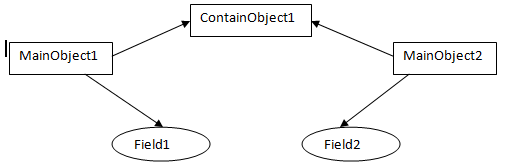
}

**Deep Copy And Shallow Copy**

Lets first separate it out and see what each one means.

*What is Shallow Copy?*

Shallow copy is a bit-wise copy of an object. A new object is created that has an exact copy of the values in the original object. If any of the fields of the object are references to other objects, just the reference addresses are copied i.e., only the memory address is copied.

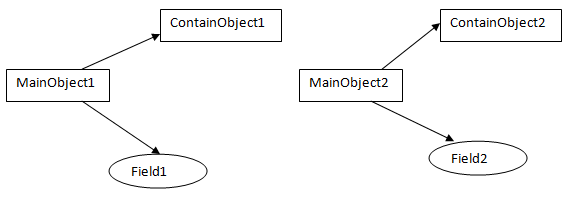


In this figure, the MainObject1 have fields "field1" of int type, and "ContainObject1" of ContainObject type. When you do a shallow copy of MainObject1, MainObject2 is created with "field3" containing the copied value of "field1" and still pointing to ContainObject1 itself. Observe here and you will find that since field1 is of primitive type, the values of it are copied to field3 but ContainedObject1 is an object, so MainObject2 is still pointing to ContainObject1. So any changes made to ContainObject1 in MainObject1 will reflect in MainObject2.

Now if this is shallow copy, lets see what's deep copy?

*What is Deep Copy?*

A deep copy copies all fields, and makes copies of dynamically allocated memory pointed to by the fields. A deep copy occurs when an object is copied along with the objects to which it refers.



In this figure, the MainObject1 have fields "field1" of int type, and "ContainObject1" of ContainObject type. When you do a deep copy of MainObject1, MainObject2 is created with "field3" containing the copied value of "field1" and "ContainObject2" containing the copied value of ContainObject1.So any changes made to ContainObject1 in MainObject1 will not reflect in MainObject2.

Well, here we are with what shallow copy and deep copy are and obviously the difference between them. Now lets see how to implement them in java.

*How to implement shallow copy in java?*

Here is an example of Shallow Copy implementation

class Subject {

private String name;

public String getName() {

return name;

}

public void setName(String s) {

name = s;

}

public Subject(String s) {

name = s;

}

}

class Student implements Cloneable {

//Contained object

private Subject subj;

private String name;

public Subject getSubj() {

return subj;

}

public String getName() {

return name;

}

public void setName(String s) {

name = s;

}

public Student(String s, String sub) {

name = s;

subj = new Subject(sub);

}

public Object clone() {

//shallow copy

try {

return super.clone();

} catch (CloneNotSupportedException e) {

return null;

}

}

}

public class CopyTest {

public static void main(String[] args) {

//Original Object

Student stud = new Student("John", "Algebra");

System.out.println("Original Object: " + stud.getName() + " - "

+ stud.getSubj().getName());

//Clone Object

Student clonedStud = (Student) stud.clone();

System.out.println("Cloned Object: " + clonedStud.getName() + " - "

+ clonedStud.getSubj().getName());

stud.setName("Dan");

stud.getSubj().setName("Physics");

System.out.println("Original Object after it is updated: "

+ stud.getName() + " - " + stud.getSubj().getName());

System.out.println("Cloned Object after updating original object: "

+ clonedStud.getName() + " - " + clonedStud.getSubj().getName());

}

}

Output is:

Original Object: John - Algebra

Cloned Object: John - Algebra

Original Object after it is updated: Dan - Physics

Cloned Object after updating original object: John - Physics

In this example, all I did is, implement the class that you want to copy with Clonable interface and override clone() method of Object class and call super.clone() in it. If you observe, the changes made to "name" field of original object (Student class) is not reflected in cloned object but the changes made to "name" field of contained object (Subject class) is reflected in cloned object. This is because the cloned object carries the memory address of the Subject object but not the actual values. Hence any updates on the Subject object in Original object will reflect in Cloned object.

*How to implement deep copy in java?*

Here is an example of Deep Copy implementation. This is the same example of Shallow Copy implementation and hence I didnt write the Subject and CopyTest classes as there is no change in them.

class Student implements Cloneable {

//Contained object

private Subject subj;

private String name;

public Subject getSubj() {

return subj;

}

public String getName() {

return name;

}

public void setName(String s) {

name = s;

}

public Student(String s, String sub) {

name = s;

subj = new Subject(sub);

}

public Object clone() {

//Deep copy

Student s = new Student(name, subj.getName());

return s;

}

}

Output is:

Original Object: John - Algebra

Cloned Object: John - Algebra

Original Object after it is updated: Dan - Physics

Cloned Object after updating original object: John - Algebra

Well, if you observe here in the "Student" class, you will see only the change in the "clone()" method. Since its a deep copy, you need to create an object of the cloned class. Well if you have references in the Subject class, then you need to implement Cloneable interface in Subject class and override clone method in it and this goes on and on.

*There is an alternative way for deep copy.*

Yes, there is. You can do deep copy through [serialization](http://www.jusfortechies.com/java/core-java/serialization.php). What does serialization do? It writes out the whole object graph into a persistant store and read it back when needed, which means you will get a copy of the whole object graph whne you read it back. This is exactly what you want when you deep copy an object. Note, when you deep copy through serialization, you should make sure that all classes in the object's graph are serializable. Let me explain you this alternative way with an example. If you want to know about serialization first, check it out [here](http://www.jusfortechies.com/java/core-java/serialization.php).

public class ColoredCircle implements Serializable

{

private int x;

private int y;

public ColoredCircle(int x, int y){

this.x = x;

this.y = y;

}

public int getX(){

return x;

}

public void setX(int x){

this.x = x;

}

public int getY(){

return y;

}

public void setX(int x){

this.x = x;

}

}

public class DeepCopy

{

static public void main(String[] args)

{

ObjectOutputStream oos = null;

ObjectInputStream ois = null;

try

{

// create original serializable object

ColoredCircle c1 = new ColoredCircle(100,100);

// print it

System.out.println("Original = " + c1);

ColoredCircle c2 = null;

// deep copy

ByteArrayOutputStream bos = new ByteArrayOutputStream();

oos = new ObjectOutputStream(bos);

// serialize and pass the object

oos.writeObject(c1);

oos.flush();

ByteArrayInputStream bin =

new ByteArrayInputStream(bos.toByteArray());

ois = new ObjectInputStream(bin);

// return the new object

c2 = ois.readObject();

// verify it is the same

System.out.println("Copied = " + c2);

// change the original object's contents

c1.setX(200);

c1.setY(200);

// see what is in each one now

System.out.println("Original = " + c1);

System.out.println("Copied = " + c2);

}

catch(Exception e)

{

System.out.println("Exception in main = " + e);

}

finally

{

oos.close();

ois.close();

}

}

}

The output is:

Original = x=100,y=100

Copied = x=100,y=100

Original = x=200,y=200

Copied = x=100,y=100

All you need to do here is:

* Ensure that all classes in the object's graph are serializable.
* Create input and output streams.
* Use the input and output streams to create object input and object output streams.
* Pass the object that you want to copy to the object output stream.
* Read the new object from the object input stream and cast it back to the class of the object you sent.

In this example, I have created a ColoredCircle object, c1 and then serialized it (write it out to ByteArrayOutputStream). Then I deserialed the serialized object and saved it in c2. Later I modified the original object, c1. Then if you see the result, c1 is different from c2. c2 is deep copy of first version of c1. So its just a copy and not a reference. Now any modifications to c1 wont affect c2, the deep copy of first version of c1.

*Well this approach has got its own limitations and issues:*

As you cannot serialize a transient variable, using this approach you cannot copy the transient variables.   
Another issue is dealing with the case of a class whose object's instances within a virtual machine must be controlled. This is a special case of the Singleton pattern, in which a class has only one object within a VM. As discussed above, when you serialize an object, you create a totally new object that will not be unique. To get around this default behavior you can use the readResolve() method to force the stream to return an appropriate object rather than the one that was serialized. In this particular case, the appropriate object is the same one that was serialized.  
Next one is the performance issue. Creating a socket, serializing an object, passing it through the socket, and then deserializing it is slow compared to calling methods in existing objects. I say, there will be vast difference in the performance. If your code is performance critical, I suggest dont go for this approach. It takes almost 100 times more time to deep copy the object than the way you do by implementing Clonable interface.

*When to do shallow copy and deep copy?*

Its very simple that if the object has only primitive fields, then obviously you will go for shallow copy but if the object has references to other objects, then based on the requiement, shallow copy or deep copy should be chosen. What I mean here is, if the references are not modified anytime, then there is no point in going for deep copy. You can just opt shallow copy. But if the references are modified often, then you need to go for deep copy. Again there is no hard and fast rule, it all depends on the requirement.

*Finally lets have a word about rarely used option - Lazy copy*

A lazy copy is a combination of both shallow copy and deep copy. When initially copying an object, a (fast) shallow copy is used. A counter is also used to track how many objects share the data. When the program wants to modify the original object, it can determine if the data is shared (by examining the counter) and can do a deep copy at that time if necessary.

Lazy copy looks to the outside just as a deep copy but takes advantage of the speed of a shallow copy whenever possible. It can be used when the references in the original object are not modified often. The downside are rather high but constant base costs because of the counter. Also, in certain situations, circular references can also cause problems.