Java

- Collections in general
- List
- Static Blocks
- Serialization
- serialVersionUID
- Externalization
- Cloning
- Deep Copy/Shallow Copy

MORE JAVA TOPICS >>

Flex Solutions

- Editable MenuBar item
- Drag - drop and position MenuBar item
- ComboBox dropdown list alignment
- Multiselect ComboBox
- Set Style to ComboBox items
- Set Style to Menu items
- Display multiple icons on Panel titlebar

MORE FLEX SOLUTIONS >>

Sample Programs

- Java program to send email
- Java program to read CSV file
- Java program to Unzip a Zip File
- Check CSS Shapes
- Creating CSS Buttons

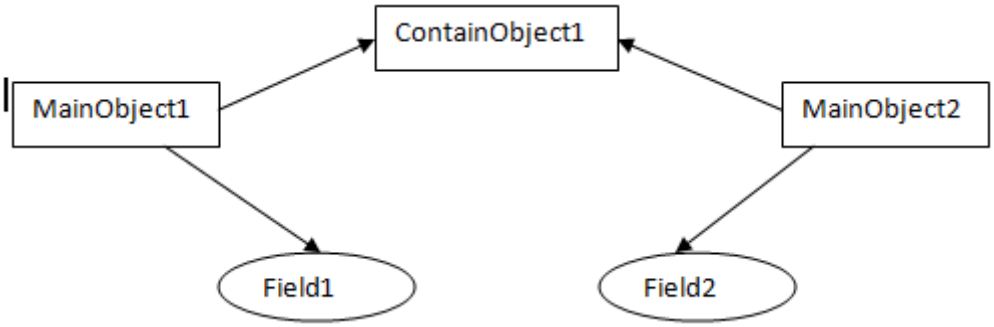
MORE PROGRAMS >>

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.

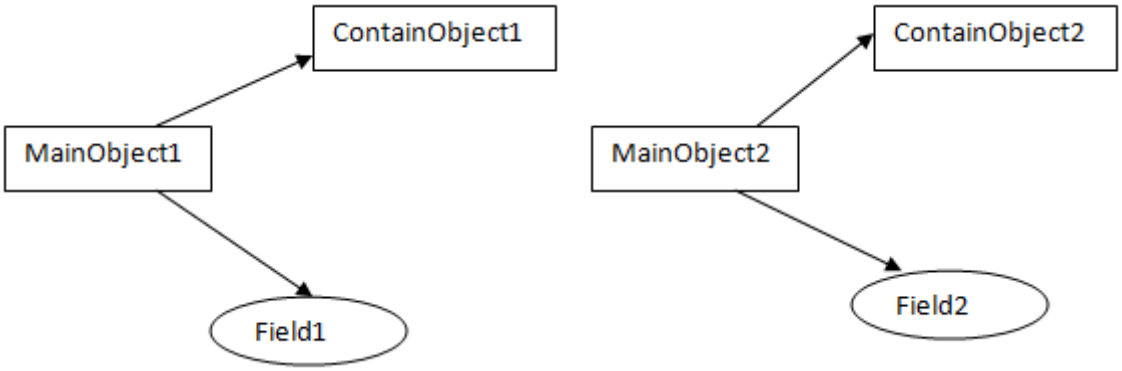


In this figure, the MainObject1 have fields "field1" of int type, and "ContainObject1" of ContainObject type. When you do a shallow copy of MainObject1, MainObject2 is created with "field3" containing the copied value of "field1" and still pointing to ContainObject1 itself. Observe here and you will 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. 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.

```
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.