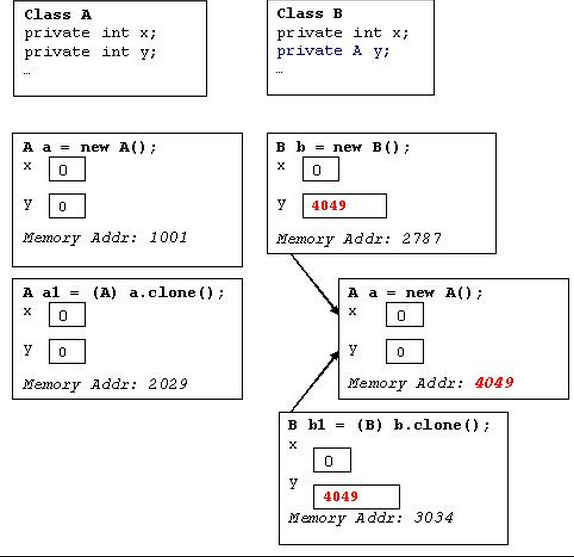
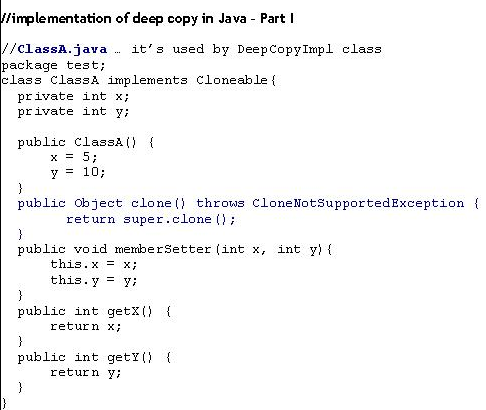
**Object Cloning:**

**What is the role of the clone() method in Java?**  
  
protected Object clone() throws CloneNotSupportedException - this method is used to create a copy of an object of a class which implements Cloneable interface. By default it does field-by-field copy as the Object class doesn't have any idea in advance about the members of the particular class whose objects call this method. So, if the class has only primitive data type members then a completely new copy of the object will be created and the reference to the new object copy will be returned. But, if the class contains members of any class type then only the object references to those members are copied and hence the member references in both the original object as well as the cloned object refer to the same object.  
  
**Cloneable interface**  
  
We get [CloneNotSupportedException](http://java.sun.com/j2se/1.3/docs/api/java/lang/CloneNotSupportedException.html) if we try to call the clone() method on an object of a class which doesn't implement the Cloneable interface. This interface is a marker interface and the implementation of this interface simply indicates that the Object.clone() method can be called on the objects of the implementing class.  
  
**Example:** how cloning works in Java?  
  
Class A {  
...  
}  
A objA = new A();  
A objACloned = (A) objA.clone();  
  
Now, objA != objACloned - this boolean expression will always be true as in any case a new object reference will be created for the cloned copy.  
  
objA.getClass() == objACloned.getClass() - this boolean expression will also be always true as both the original object and the cloned object are instances of the same class (A in this case).  
  
Initially, objA.equals(objACloned) will return true, but any changes to any primitive data type member of any of the objects will cause the expression to returnfalse. It's interesting to note here that any changes to the members of the object referenced by a member of these objects will not cause the expression to returnfalse. Reason being, both the copies are referring to same object as only the object references get copied and not the object themselves. This type of copy is called **Shallow Copy.**

**This can be understood easily by looking at the following memory diagram:-**



**Shallow Copy**  
  
This is a result of the default cloning functionality provided by the Object.clone() method if the class has non-primitive data type members as well. Shallow Copy concept is not applicable to the classes having only primitive data type members as in that case the default cloning will also result into a Deep Copy only.  
  
In case of Shallow Copy, the cloned object also refers to the same object to which the original object refers as only the object references gets copied and not the referred objects themselves. That's why the name Shallow Copy.   
  
**Deep Copy**  
  
We need to override the clone() method for the classes having non-primitive type members to achieve Deep Copy as Deep Copy requires the member objects to be cloned as well, which is not done by the default cloning mechanism. Why is it not done by the default cloning? Because clone() is a method of the Object class and at that level it's not known what a typical class can have as its members and hence only a field-by-field copy approach has been provided as the default cloning mechanism.  
  
**Implementing Deep Copy in Java**  
  
For Deep Copy, we need to ensure that the member classes also implement the Cloneable interface otherwise calling the clone() method on the objects of those classes will result into CloneNotSupportedException. So, to implement Deep Copy, we first need to ensure that all the member classes (at all the levels - like if the member class itself has a member of some class type then that class as well... and so on) are implementing the Cloneable interface. After that we override the clone() method in all those classes (even in the classes where we have only primitive type members otherwise we would not be able to call the protected clone()method of Object class on the instances of those classes inside some other class ... a typical restriction of the protected access. We'll cover this in a separate article) and finally calling clone() method on the object members in the overridden clone() method definition.





|  |
| --- |
| **Cloning Interview questions** |
|  |
| **Q1) What are different types of cloning in Java?**  Ans) Java supports two type of cloning: - Deep and shallow cloning. By default shallow copy is used in Java. Object class has a method clone() which does shallow cloning. |
| **Q2) What is Shallow copy?**  Ans) In **shallow copy** the object is copied without its contained objects. Shallow clone only copies the top level structure of the object not the lower levels. It is an exact bit copy of all the attributes.  Original Figure 1: Original java object obj  The shallow copy is done for obj and new object obj1 is created but contained objects of obj are not copied.  Shallow Copy Figure 2: Shallow copy object obj1  It can be seen that no new objects are created for obj1 and it is referring to the same old contained objects. If either of the containedObj contain any other object no new reference is created |
| **Q3) What is deep copy and how it can be acheived?**  Ans) In **deep copy** the object is copied along with the objects it refers to. Deep clone copies all the levels of the object from top to the bottom recursively.  Original Figure 3 : Original Object obj  When a deep copy of the object is done new references are created.  Deep Copy Figure 4: obj2 is deep copy of obj1  One solution is to simply implement your own custom method (e.g., deepCopy()) that returns a deep copy of an instance of one of your classes. This may be the best solution if you need a complex mixture of deep and shallow copies for different fields, but has a few significant drawbacks:   * You must be able to modify the class (i.e., have the source code) or implement a subclass. If you have a third-party class for which you do not have the source and which is marked final, you are out of luck. * You must be able to access all of the fields of the classâ€™s superclasses. If significant parts of the objectâ€™s state are contained in private fields of a superclass, you will not be able to access them. * You must have a way to make copies of instances of all of the other kinds of objects that the object references. This is particularly problematic if the exact classes of referenced objects cannot be known until runtime. * Custom deep copy methods are tedious to implement, easy to get wrong, and difficult to maintain. The method must be revisited any time a change is made to the class or to any of its superclasses.   Other common solution to the deep copy problem is to use **Java Object Serialization** (JOS). The idea is simple: Write the object to an array using JOSâ€™s **ObjectOutputStream** and then use **ObjectInputStream** to reconsistute a copy of the object. The result will be a completely distinct object, with completely distinct referenced objects. JOS takes care of all of the details: superclass fields, following object graphs, and handling repeated references to the same object within the graph.   * It will only work when the object being copied, as well as all of the other objects references directly or indirectly by the object, are serializable. (In other words, they must implement java.io.Serializable.) Fortunately it is often sufficient to simply declare that a given class implements java.io.Serializable and let Javaâ€™s default serialization mechanisms do their thing. Java Object Serialization is slow, and using it to make a deep copy requires both serializing and deserializing.   There are ways to speed it up (e.g., by pre-computing serial version ids and defining custom readObject() and writeObject() methods), but this will usually be the primary bottleneck. The byte array stream implementations included in the java.io package are designed to be general enough to perform reasonable well for data of different sizes and to be safe to use in a multi-threaded environment. These characteristics, however, slow down ByteArrayOutputStream and (to a lesser extent) ByteArrayInputStream . |
| **Q4) What is difference between deep and shallow cloning?**  Ans) The differences are as follows:   * Consider the class:   public class MyData{ String id; Map myData; } The shallow copying of this object will have new id object and values as “” but will point to the myData of the original object. So a change in myData by either original or cloned object will be reflected in other also. But in deep copying there will be new id object and also new myData object and independent of original object but with same values.   * Shallow copying is default cloning in Java which can be achieved using clone() method of Object class. For deep copying some extra logic need to be provided. |
| **Q5) What are the characteristics of a shallow clone?**  Ans) If we do a = clone(b) 1) Then b.equals(a) 2) No method of a can modify the value of b. |
| **Q6) What are the disadvantages of deep cloning?**  Ans) Disadvantages of using Serialization to achieve deep cloning –   * Serialization is more expensive than using object.clone(). * Not all objects are serializable. * Serialization is not simple to implement for deep cloned object.. |