|  |
| --- |
| **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.. |