**Serialization**

Java serialization is the process, which is used to serialize object in java by storing object’s state into a file with extension **.ser** and recreating object's state from that file, this reverse process is called deserialization.

The Java Serialization API provides a standard mechanism for developers to handle object serialization using Serializable and Externalizable interface

**What is Serialization in Java**

Object Serialization in Java is a process used to convert Object into a binary format, which can be persisted into disk or sent over network to any other running [Java virtual machine](http://javarevisited.blogspot.sg/2011/11/hotspot-jvm-options-java-examples.html); the reverse process of creating object from binary stream is called deserialization in Java. Java provides Serialization API for serializing and deserializing object, which includes java.io.Serializable, java.io.Externalizable, ObjectInputStream and ObjectOutputStream etc. Java programmers are free to use default Serialization mechanism which Java uses based upon structure of class but they are also free to use their own custom binary format, which is often advised as Serialization best practice, Because serialized binary format becomes part of Class's exported API and it can potentially break [Encapsulation in Java](http://javarevisited.blogspot.sg/2012/03/what-is-encapsulation-in-java-and-oops.html) provided by private and [package-private fields](http://javarevisited.blogspot.sg/2012/05/how-to-access-private-field-and-method.html).   
**How to make a Java class Serializable?**   
Implements java.io.Serializable interface and JVM will take care of serializing object in default format. Decision to making a [Class](http://javarevisited.blogspot.sg/2011/10/class-in-java-programming-general.html) Serializable should be taken concisely because though near term cost of making a Class Serializable is low, long term cost is substantial and it can potentially limit your ability to further modify and change its implementation because like any public API, serialized form of an object becomes part of public API and when you change structure of your class by implementing addition interface, adding or removing any field can potentially break default serialization, this can be minimized by using a custom binary format but still requires lot of effort to ensure backward compatibility. One example of How Serialization can put constraints on your ability to change class is**SerialVersionUID**. If you do not explicitly declare SerialVersionUID then JVM generates its based upon structure of class, which depends upon interfaces a class implements, and several other factors which is subject to change. Suppose you implement another interface than [JVM](http://javarevisited.blogspot.sg/2011/12/jre-jvm-jdk-jit-in-java-programming.html) will generate a different SerialVersionUID for new version of class files and when you try to load old object. Object serialized by old version of your program you will get **InvalidClassException**.

**Question 1) What is the difference between Serializable and Externalizable interface in Java?**

Externalizable provides us writeExternal() and readExternal() method which gives us flexibility to control java serialization mechanism instead of relying on Java's default serialization.

**Question 2) How many methods Serializable has? If no method then what is the purpose of Serializable interface?**

Serializable interface exists in [**java.io**](http://java.io/) package and forms core of java serialization mechanism. It does not have any method and called [Marker Interface in Java](http://javarevisited.blogspot.sg/2012/01/what-is-marker-interfaces-in-java-and.html). When your class implementsjava.io.Serializable interface it becomes Serializable in Java and gives compiler an indication that use Java Serialization mechanism to serialize this object.

**Question 3) What is serialVersionUID? What would happen if you don't define this?**

SerialVersionUID is an ID which is stamped on object when it get serialized usually hashcode of object, you can use tool serialver to see serialVersionUID of a serialized object . SerialVersionUID is used for version control of object. you can specify serialVersionUID in your [class file](http://javarevisited.blogspot.sg/2012/05/10-points-about-class-file-in-java.html) also. Consequence of not specifying serialVersionUID is that when you add or modify any field in class then already serialized class will not be able to recover because serialVersionUID generated for new class and for old serialized object will be different. Java serialization process relies on correct serialVersionUID for recovering state of serialized object and throws java.io.InvalidClassException in case of serialVersionUID mismatch; to learn more about serialVersionUID.

**Question 4) While serializing you want some of the members not to serialize? How do you achieve it?**

If you don't want any field to be part of object's state then declare it either static or transient based on your need and it will not be included during Java serialization process.  
  
**Question 5) What will happen if one of the members in the class doesn't implement Serializable interface?**   
If you try to serialize an object of a class which implements Serializable, but the object includes a reference to an non- Serializable class then a ‘NotSerializableException’ will be thrown at runtime and this is why I always put a *SerializableAlert* (comment section in my code) , one of the [code comment best practices](http://javarevisited.blogspot.sg/2011/08/code-comments-java-best-practices.html), to instruct developer to remember this fact while adding a new field in a Serializable class.

**Question 6) If a class is Serializable but its super class in not, what will be the state of the instance variables inherited from super class after deserialization?**

Java serialization process only continues in object hierarchy till the class is Serializable i.e. implements Serializable [interface in Java](http://javarevisited.blogspot.sg/2012/04/10-points-on-interface-in-java-with.html) and values of the instance variables inherited from super class will be initialized by calling constructor of Non-Serializable Super class during deserialization process. Once the [constructor chaining](http://javarevisited.blogspot.sg/2012/01/what-is-constructor-overloading-in-java.html) will started it would not be possible to stop that, hence even if classes higher in hierarchy implements Serializable interface, there constructor will be executed.

**Question 7) Can you Customize Serialization process or can you override default Serialization process in Java?**

Yes you can. We all know that for serializing an object ObjectOutputStream.writeObject (saveThisobject) is invoked and for reading object ObjectInputStream.readObject() is invoked but there is one more thing which Java Virtual Machine provides you is to define these two method in your class. If you define these two methods in your class then JVM will invoke these two methods instead of applying default serialization mechanism. You can customize behavior of object serialization and deserialization here by doing any kind of pre or post processing task. Important point to note is making these methods private to avoid being inherited, [overridden or overloaded](http://javarevisited.blogspot.sg/2011/12/method-overloading-vs-method-overriding.html). Since only Java Virtual Machine can call private method integrity of your class will remain and Java Serialization will work as normal. Why should you provide custom serialized form for your object?

**Question 8) Suppose super class of a new class implement Serializable interface, how can you avoid new class to being serialized?**

If Super Class of a Class already implements Serializable interface in Java then its already Serializable in Java, since you can not unimplemented an interface its not really possible to make it Non Serializable class but yes there is a way to avoid serialization of new class. To avoid Java serialization you need to implement writeObject() and readObject() method in your Class and need to throwNotSerializableException from those method.

**Question 9) Which methods are used during Serialization and DeSerialization process in Java?**

readObject(), writeObject(), readExternal() and writeExternal() or not. Java Serialization is done by java.io.ObjectOutputStream class. That class is a filter stream which is wrapped around a lower-level byte stream to handle the serialization mechanism. To store any object via serialization mechanism we call ObjectOutputStream.writeObject(saveThisobject) and to deserialize that object we call ObjectInputStream.readObject() method. Call to writeObject() method trigger serialization process in java. one important thing to note about readObject() method is that it is used to read bytes from the persistence and to create object from those bytes and its return an [Object](http://javarevisited.blogspot.sg/2012/03/10-object-oriented-design-principles.html) which needs to be type cast to correct type.

**Question 10) Suppose you have a class which you serialized it and stored in persistence and later modified that class to add a new field. What will happen if you deserialize the object already serialized?**

**It depends on whether class has its own serialVersionUID or not**. If we don't provide serialVersionUID in our code java compiler will generate it and normally it’s [equal to hashCode of object](http://javarevisited.blogspot.sg/2011/02/how-to-write-equals-method-in-java.html). by adding any new field there is chance that new serialVersionUID generated for that class version is not the same of already serialized object and in this case Java Serialization API will [throw](http://javarevisited.blogspot.sg/2012/02/difference-between-throw-and-throws-in.html) **java.io.InvalidClassException so use**serialVersionUID in code and make sure to keep it same always for a single class.   
**11) What are the compatible changes and incompatible changes in Java Serialization Mechanism?**   
The challenge lies with change in class structure by adding any field, method or removing any field or method is that with already serialized object. As per Java Serialization specification, adding any field or method comes under compatible change and changing class hierarchy or UN-implementing Serializable interfaces some under non-compatible changes.

**12) Can we transfer a Serialized object vie network?**

Yes, you can transfer a Serialized object via network because Java serialized object remains in form of bytes, which can be transmitted via network. You can also store serialized object in Disk or database as Blob.

public class MySingleton implements Serializable{  
 private static MySingleton myInstance;   
 private MySingleton(){   
 }  
  static{  
    myInstance =new MySingleton();  
 }  
 public static MySingleton getInstance(){  
    return MySingleton.myInstance;  
 }  
}

If another class X get the instance of the single and writes it to a file and at a later point deserializes it to obtain another instance we would have two instances which is against the Singleton principle.  
-----------------

The best way to do this is to use the enum singleton pattern:

public enum MySingleton {  
  INSTANCE;  
}  
This guarantees the singleton-ness of the object and provides serializability for you in such a way that you always get the same instance.

More generally, you can provide a readResolve() method like so:

protected Object readResolve() {  
  return myInstance;  
}

The rules for serializing an enum instance differ from those for serializing an "ordinary" serializable object: the serialized form of an enum instance consists only of its enum constant name, along with information identifying its base enum type. Deserialization behavior differs as well--the class information is used to find the appropriate enum class, and the Enum.valueOf method is called with that class and the received constant name in order to obtain the enum constant to return.

Suppose that your application has instantiated the singleton in the normal course of events, and then it deserializes some object graph that includes a copy of a previous instance of the singleton. What can it do?

* If it deserializes the singleton normally, it violates "singleton-ness".
* If it doesn't then the application cannot access the singleton's previous state.

The solution with enum won't work with Singletons managed by Spring, EJB, Guice or any other DI framework. It works only with enums, only because enum is treated specially by the serialization algorithm.

Firstly, *singletons don't need serialization*, because if you deserialized it, and then deserialized singleton != YourSingleton.getInstance(), it would mean that you have two instances of your singleton, which means that YourSingleton isn't singleton at all, which may lead to unpredictable bugs.

However sometimes you need to serialize non-singleton which contains a reference to singleton. The solution is easy:

class NonSingleton implements Serializable {  
   private transient YourSingleton singleton = YourSingleton.getInstance();  
   ...  
}

With Spring:

@Configurable  
class NonSingleton implements Serializable {  
   @Autowired  
   private transient YourSingleton singleton;  
   ...  
}