# Java Serialization Interview questions

# <http://www.fromdev.com/2012/06/15-java-serialization-interview.html>

# Java Serialization Interview Questions and Answers

# What’s the difference between serialization and marshaling ?

# <http://stackoverflow.com/questions/770474/what-is-the-difference-between-serialization-and-marshaling>

# The term "marshal" is considered to be synonymous with "serialize" in the Python standard library[1], but the terms are not synonymous in the Java-related RFC 2713:

# To "marshal" an object means to record its state and codebase(s) in such a way that when the marshalled object is "unmarshalled", a copy of the original object is obtained, possibly by automatically loading the class definitions of the object. You can marshal any object that is serializable or remote. Marshalling is like serialization, except marshalling also records codebases. Marshalling is different from serialization in that marshaling treats remote objects specially. (RFC 2713)

# To "serialize" an object means to convert its state into a byte stream in such a way that the byte stream can be converted back into a copy of the object.

# <https://en.wikipedia.org/wiki/Marshalling_(computer_science)>

# Note: "Codebase" here is used in its Java-specific meaning, to refer to a list of URLs where the object code can be loaded from, rather than in the more general meaning of codebase which refers to source code.

# What do you mean by Serialization in Java?

# Serialization is a mechanism by which you can save or transfer the state of an object by converting it to a byte stream. This can be done in java by implementing Serialiazable interface. Serializable is defined as a marker interface which needs to be implemented for transferring an object over a network or persistence of its state to a file. Since its a marker interface, it does not contain any methods. Implementation of this interface enables the conversion of object into byte stream and thus can be transferred. The object conversion is done by the JVM using its default serialization mechanism.

# What is purpose of Serialization? Why is serialization required ? What is need to serialization ?

Serialization is required for a variety of reasons. It is required to send across the state of an object over a network by means of a socket. One can also store an object’s state in a file. Additionally, manipulation of the state of an object as streams of bytes is required. The core of Java Serialization is the Serializable interface. When Serializable interface is implemented by your class it provides an indication to the compiler that java Serialization mechanism needs to be used to serialize the object.

Some more explicit explaination

<http://stackoverflow.com/questions/2232759/what-is-the-purpose-of-serialization-in-java>

Let's define serialization first, then we can talk about why it's so useful.

Serialization is simply turning an existing object into a byte array. This byte array represents the class of the object, the version of the object, and the internal state (metadata) of the object. This byte array can then be used between JVM's running the same code to transmit/read the object.

Why would we want to do this?

There are several reasons:

1. Communication: If you have two machines that are running the same code, and they need to communicate, an easy way is for one machine to build an object with information that it would like to transmit, and then serialize that object to the other machine. It's not the best method for communication, but it gets the job done.
2. Persistence: If you want to store the state of a particular operation in a database, it can be easily serialized to a byte array, and stored in the database for later retrieval.

🡪 Save the explicit version of object you want into database with serialization

1. Deep Copy: If you need an exact replica of an Object, and don't want to go to the trouble of writing your own specialized clone() class, simply serializing the object to a byte array, and then de-serializing it to another object achieves this goal.
2. Caching: Really just an application of the above, but sometimes an object takes 10 minutes to build, but would only take 10 seconds to de-serialize. So, rather than hold onto the giant object in memory, just cache it out to a file via serialization, and read it in later when it's needed. 🡪 Giant object can be serialized and store in file instead of store in memory
3. Cross JVM Synchronization: Serialization works across different JVMs that may be running on different architectures.

# What is difference between externalizable and serializable interface ?

# Serializable is a marker interface therefore you are not forced to implement any methods, however Externalizable contains two methods readExternal() and writeExternal() which must be implemented.

# Serializable interface provides a inbuilt serialization mechanism to you which can be in-efficient at times. However Externilizable interface is designed to give you greater control over the serialization mechanism. The two methods provide you immense opportunity to enhance the performance of specific object serialization based on application needs.

# Serializable interface provides a default serialization mechanism, on the other hand, Externalizable interface instead of relying on default Java Serialization provides flexibility to control this mechanism.

# When will you use Serializable or Externalizable interface ? and why ?

# Most of the times when you want to do a selective attribute serialization you can use Serializable interface with transient modifier for variables not to be serialized. However, use of Externalizable interface can be really effective in cases when you have to serialize only some dynamically selected attributes of a large object.

# Lets take an example, Some times when you have a big Java object with hundreds of attributes and you want to serialize only a dozen dynamically selected attributes to keep the state of the object you should use Externalizable interface writeExternal method to selectively serialize the chosen attributes.

# In case you have small objects and you know that most or all attributes are required to be serialized then you should be fine with using Serializable interface and use of transient variable as appropriate.

# What the ways to speed up Object serialization ? How to improve Serialization performance ?

# The default Java Serialization mechanism is really useful, however it can have a really bad performance based on your application and business requirements. The serialization process performance heavily depends on the number and size of attributes you are going to serialize for an object. Below are some tips you can use for speeding up the marshaling and un-marshaling of objects during Java serialization process.

# (a) Mark the unwanted or non Serializable attributes as transient. This is a straight forward benefit since your attributes for serialization are clearly marked and can be easily achieved using Serialzable interface itself.

# (b) Save only the state of the object, not the derived attributes. Some times we keep the derived attributes as part of the object however serializing them can be costly. Therefore consider calcualting them during de-serialization process.

# (c) Serialize attributes only with NON-default values. For examples, serializing a int variable with value zero is just going to take extra space however, choosing not to serialize it would save you a lot of performance. This approach can avoid some types of attributes taking unwanted space. This will require use of Externalizable interface since attribute serialization is determined at runtime based on the value of each attribute.

# (d) Use Externalizable interface and implement the readExternal and writeExternal methods to dynamically identify the attributes to be serialized. Some times there can be a custom logic used for serialization of various attributes.

# What is serialVersionUID ? and why we should use that ? How to generate one ?

# The serialVersionUID represents your class version, and you should change it if the current version of your class is not backwards compatible with its earlier versions. Most of the times, we probably do not use serialization directly. In such cases, I would suggest to generate a default serializable uid by clicking the quick fix option in eclipse.

# What would happen if the serialVersionUID of an object is not defined ?

# If you don't define serialVersionUID in your serilizable class, Java compiler will make one by creating a hash code using most of your class attributes and features. When an object gets serialized, this hash code is stamped on the object which is known as the SerialVersionUID of that object. This ID is required for the version control of an object. SerialVersionUID can be specified in the class file also. In case, this ID is not specified by you, then Java compiler will regenerate a SerialVersionUID based on updated class and it will not be possible for the already serialized class to recover when a class field is added or modified. Its recommended that you always declare a serialVersionUID in your Serializable classes (Then this manually defined serialVersionUID can be used for auto check on whether serializable classes been changed or not).

# Does setting the serialVersionUID class field improve Java serialization performance ?

# Declaring an explicit serialVersionUID field in your classes saves some CPU time only the first time the JVM process serializes a given Class. However the gain is not significant, In case when you have not declared the serialVersionUID its value is computed by JVM once and subsequently kept in a soft cache for future use

# What are alternatives to serialization ? If serialization is not used, is it possible to persist or transfer an object using other approach ?

# (a)Saving object state to database, this is most common technique used by most applications. You can use ORM tools (e.g. hibernate) to save the objects in a database and read them from the database.

# (b)Xml based data transfer is another popular mechanism, and a lot of XML based web services use this mechanism to transfer data over network. Also a lot of tools save XML files to persist data/configurations.

# (c)JSON Data Transfer - is recently popular data transfer format. A lot of web services are being developed in JSON due to its small footprint and inherent integration with web browser due to JavaScript format.

# What are transient variables? What role do they play in Serialization process ?

# Also check <https://www.cs.uic.edu/~troy/fall04/cs441/drake/serialization.html>

# <http://javarevisited.blogspot.com/2011/09/transient-keyword-variable-in-java.html>

# The transient keyword in Java is used to indicate that a field should not be serialized. Once the process of de-serialization is carried out, the transient variables do not undergo a change and retain their default value. Marking unwanted fields as transient can help you boost the serialization performance. Below is a simple example where you can see the use of transient keyword.

# class MyVideo implements Serializable

# {

# private Video video;

# private transient Image thumbnailVideo;

# private void generateThumbnail()

# {

# // Generate thumbnail.

# }

# private void readObject(ObjectInputStream inputStream)

# throws IOException, ClassNotFoundException

# {

# inputStream.defaultReadObject();

# generateThumbnail();

# }

# }

# Why does serialization NOT save the value of static class attributes? Why static variables are not serialized?

# The Java variables declared as static are not considered part of the state of an object since they are shared by all instances of that class. Saving static variables with each serialized object would have following problems

# (a)It will make redundant copy of same variable in multiple objects which makes it in-efficient.

# (b)The static variable can be modified by any object and a serialized copy would be stale or not in sync with current value.

# How to Serialize a collection in java? How to serialize a ArrayList, Hashmap or Hashset object in Java?

# All standard implementations of collections List, Set and Map interface already implement java.io.Serializable. All the commonly used collection classes like java.util.ArrayList, java.util.Vector, java.util.Hashmap, java.util.Hashtable, java.util.HashSet, java.util.TreeSet do implement Serializable. This means you do not really need to write anything specific to serialize collection objects.

# However you should keep following things in mind before you serialize a collection object – a.Make sure all the objects added in collection are Serializable. – b.Serializing the collection can be costly therefore make sure you serialize only required data isntead of serializing the whole collection. - In case you are using a custom implementation of Collection interface then you may need to implement serialization for it.

# Is it possible to customize the serialization process? How can we customize the Serialization process?

# Yes, the serialization process can be customized. When an object is serialized, objectOutputStream.writeObject (to save this object) is invoked and when an object is read, ObjectInputStream.readObject () is invoked. What most people do not know is that Java Virtual Machine provides you with an option to define these methods as per your needs. Once this is done, these two methods will be invoked by the JVM instead of the application of the default serialization process. Classes that require special handling during the serialization and deserialization process must implement special methods with these exact signatures:

# private void writeObject(java.io.ObjectOutputStream out)

# throws IOException

# private void readObject(java.io.ObjectInputStream in)

# throws IOException, ClassNotFoundException;

# private void readObjectNoData()

# throws ObjectStreamException;

# How can a sub-class of Serializable super class avoid serialization? If serializable interface is implemented by the super class of a class, how can the serialization of the class be avoided?

# In Java, if the super class of a class is implementing Serializable interface, it means that it is already serializable. Since, an interface cannot be unimplemented, it is not possible to make a class non-serializable. However, the serialization of a new class can be avoided. For this, writeObject () and readObject() methods should be implemented in your class so that a Not Serializable Exception can be thrown by these methods. And, this can be done by customizing the Java Serialization process. Below the code that demonstrates it

# class MySubClass extends SomeSerializableSuperClass {

# private void writeObject(java.io.ObjectOutputStream out)

# throws IOException {

# throw new NotSerializableException(“Can not serialize this class”);

# }

# private void readObject(java.io.ObjectInputStream in)

# throws IOException, ClassNotFoundException {

# throw new NotSerializableException(“Can not serialize this class”);

# }

# private void readObjectNoData()

# throws ObjectStreamException; {

# throw new NotSerializableException(“Can not serialize this class”);

# }

# }

# What changes are compatible and incompatible to the mechanism of java Serialization?

# This is one of a difficult and tricky questions and answering this correctly would mean you are an expert in Java Serialization concept. In an already serialized object, the most challenging task is to change the structure of a class when a new field is added or removed. As per the specifications of Java Serialization, addition of any method or field is considered to be a compatible change whereas changing of class hierarchy or non-implementation of Serializable interface is considered to be a non-compatible change. You can go through the Java serialization specification for the extensive list of compatible and non-compatible changes. If a serialized object need to be compatible with an older version, it is necessary that the newer version follows some rules for compatible and incompatible changes. A compatible change to the implementing class is one that can be applied to a new version of the class, which still keeps the object stream compatible with older version of same class.

# Some Simple Examples of compatible changes are:

# (a)Addition of a new field or class will not affect serialization, since any new data in the stream is simply ignored by older versions. the newly added field will be set to its default values when the object of an older version of the class is un marshaled.

# (b)The access modifiers change (like private, public, protected or default) is compatible since they are not reflected in the serialized object stream.

# (c)Changing a transient field to a non-transient field is compatible change since it is similar to adding a field.

# (d)Changing a static field to a non-static field is compatible change since it is also similar to adding a field.

# Some Simple Examples of incompatible changes are:

# (a)Changing implementation from Serializable to Externalizable interface can not be done since this will result in the creation of an incompatible object stream.

# (b)Deleting a existing Serializable fields will cause a problem.

# (c)Changing a non-transient field to a transient field is incompatible change since it is similar to deleting a field.

# (d)Changing a non-static field to a static field is incompatible change since it is also similar to deleting a field.

# (e)Changing the type of a attribute within a class would be incompatible, since this would cause a failure when attempting to read and convert the original field into the new field.

# (f)Changing the package of class is incompatible. Since the fully-qualified class name is written as part of the object byte stream.

# Explain marshall and unmarshall in java

# <http://javaquestionsinterview.blogspot.com/2012/03/java-interview-questions-can-you_19.html>

# Marshalling:

# Marshalling creates an XML document from a content tree

# To marshal a content tree

# Create a JAXBContext object

# Create a Marshaller object (Marshaller marshaller = jaxbContext.createMarshaller();)

# Set required properties using setProperty method of Marshaller marshaller.setProperty(Marshaller.JAXB\_FORMATTED\_OUTPUT, new Boolean(true));

# Call the marshal method marshaller.marshal(collection, new FileOutputStream("jaxbOutput.xml"));

# Unmarshalling:

# Unmarshalling an XML document means creating a tree of content objects that represents the content and organization of the document

# To unmarshal an XML document,

# Create a JAXBContext object(

# JAXBContext jaxbContext = JAXBContext.newInstance("package name ");

# Create an Unmarshaller object

# Unmarshaller unmarshaller = jc.createUnmarshaller();

# Call the unmarshal method unmarshaller.unmarshal(new File( "xml name"));

# Use the get methods in the schema-derived classes to access the XML data