Serialization:

Java Serialization API allows us to convert an Object to stream that we can send over the network or save it as file or store in DB for later usage. Deserialization is the process of converting Object stream to actual Java Object to be used in our program.

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

Classes ObjectInputStream and ObjectOutputStream are high-level streams that contain the methods for serializing and deserializing an object.

Notice that for a class to be serialized successfully, two conditions must be met:

The class must implement the java.io.Serializable interface.

All of the fields in the class must be serializable. If a field is not serializable, it must be marked transient.

If you want an object property to be not serialized to stream, you can use transient keyword

Serialization with Inheritance

Sometimes we need to extend a class that doesn’t implement Serializable interface. If we rely on the automatic serialization behavior and the superclass has some state, then they will not be converted to stream and hence not retrieved later on.

This is one place, where readObject() and writeObject() methods really help. By providing their implementation, we can save the super class state to the stream and then retrieve it later on. Let’s see this in action.

Notice that order of writing and reading the extra data to the stream should be same. We can put some logic in reading and writing data to make it secure.

Also notice that the class is implementing ObjectInputValidation interface. By implementing validateObject() method, we can put some business validations to make sure that the data integrity is not harmed.

Serialization Proxy Pattern

Java Serialization comes with some serious pitfalls such as;

The class structure can’t be changed a lot without breaking the serialization process. So even though we don’t need some variables later on, we need to keep them just for backward compatibility.

Serialization causes huge security risks, an attacker can change the stream sequence and cause harm to the system. For example, user role is serialized and an attacker change the stream value to make it admin and run malicious code.

Serialization Proxy pattern is a way to achieve greater security with Serialization. In this pattern, an inner private static class is used as a proxy class for serialization purpose. This class is designed in the way to maintain the state of the main class. This pattern is implemented by properly implementing readResolve() and writeReplace() methods.

What is the use of Serialization:

Write to Disk

Store in Memory

Sent byte stream to other platform over network

Save byte stream in DB(As BLOB)

Concept of serialVersionUID :

What is use of serialVersionUID?

serialVersionUID is used to ensure that same class(That was used during Serialization) is loaded during Deserialization.serialVersionUID is used for version control of object.You can read more at serialVersionUID in java serialization

Everytime an object is serialized the java serialization mechanism automatically computes a hash value. ObjectStreamClass's computeSerialVersionUID() method passes the class name, sorted member names, modifiers, and interfaces to the secure hash algorithm (SHA), which returns a hash value.The serialVersionUID is also called suid.

So when the serilaize object is retrieved , the JVM first evaluates the suid of the serialized class and compares the suid value with the one of the object. If the suid values match then the object is said to be compatible with the class and hence it is de-serialized. If not InvalidClassException exception is thrown.

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.

What if an object has a reference to other objects?

When you serialize any object and if it contain any other object reference then Java serialization serialize that object's entire object graph.

What if you don't have access to reference object's source code and ref obj doesn't implement Serializable interface?

You can create another class which extends address and make it serialzable but It can fails in many cases:

What if class is declared as final

What if class have reference to other non serializable object.

solution is you can make it transient.

What if you still want to save state of reference object(e.g above address object):

Java serialization provides a mechnism such that if you have private methods with particular signature then they will get called during serialization and deserialization so we will override writeObject and readObject method of class and they will be called during serialization and deserialization of object.

What if superclass is Serializable?

If superclass is serialzable then all its subclasses are automatically serializable.

What if superclass is not Serializable?

If super class is not serializable then we have to handle it quite differently.

If superclass is not serializable then it must have no argument constructor.

If superclass is not Serializable then all values of the instance variables inherited from super class will be initialized by calling constructor of Non-Serializable Super class during deserialization process.

What if superclass is Serializable but you don't want subclass to be Serializable

If you don't want subclass to serializable then you need to implement writeObject() and readObject() method and need to throw NotSerializableException from this methods.

Can you Serialize static variables?

No,you can't.As you know static variable are at class level not at object level and you serialize a object so you can't serialize static variables.

Saving static variables with each serialized object would have following problems:

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

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

What will be the value of transient variable after de-serialization?

Ans) It’s default value.

e.g. if the transient variable in question is an int, it’s value after deserialization will be zero.

What is the Difference between Externalizable and Serializable Interfaces?

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. One can drastically improve the application performance by implementing the Externalizable interface correctly.

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 are the ways to speed up Object Serialization? How to improve Serialization performance?

- Mark the unwanted or non Serializable attributes as transient.

- Save only the state of the object, not the derived attributes.

- Serialize attributes only with NON-default values.

-Use Externalizable interface and implement the readExternal and writeExternal methods to dynamically identify the attributes to be serialized.

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

- Saving object state to database

- Xml based data transfer

- JSON Data Transfer