**Serialization in Java**

Serialization is the process of converting an object's state (including its references) to a sequence of bytes, as well as the process of rebuilding those bytes into a live object at some future time. Simple......Coverting an object to bytes and bytes back to object. So when is serialization used? Serialization is used when you want to persist the object. It is also used by RMI to pass objects between JVMs, either as arguments in a method invocation from a client to a server or as return values from a method invocation. In general, serialization is used when we want the object to exist beyond the lifetime of the JVM.

*Lets see couple of different scenarios (examples) where we use serialization.*

* 1. Banking example: When the account holder tries to withdraw money from the server through ATM, the account holder information along with the withdrawl details will be serialized (marshalled/flattened to bytes) and sent to server where the details are deserialized (unmarshalled/rebuilt the bytes)and used to perform operations. This will reduce the network calls as we are serializing the whole object and sending to server and further request for information from client is not needed by the server.
* 2. Stock example: Lets say an user wants the stock updates immediately when he request for it. To achieve this, everytime we have an update, we can serialize it and save it in a file. When user requests the information, deserialize it from file and provide the information. This way we dont need to make the user wait for the information until we hit the database, perform computations and get the result.

Here are some uses of serialization

* To persist data for future use.
* To send data to a remote computer using such client/server Java technologies as RMI or socket programming.
* To "flatten" an object into array of bytes in memory.
* To exchange data between applets and servlets.
* To store user session in Web applications.
* To activate/passivate enterprise java beans.
* To send objects between the servers in a cluster.

So far we saw what and when serialization is used.

*Lets see now how serialization is performed in java.*

Java provides Serialization API, a standard mechanism to handle object serialization. To persist an object in java, the first step is to flatten the object. For that the respective class should implement "java.io.Serializable" interface. Thats it. We dont need to implement any methods as this interface do not have any methods. This is a marker interface/tag interface. Marking a class as Serializable indicates the underlying API that this object can be flattened.

public class SerialClass implements Serializable {

private Date currentTime;

public SerialClass() {

currentTime = Calendar.getInstance().getTime();

}

public Date getCurrentTime() {

return currentTime;

}

}

Now you marked the object for flattening. Next step is to actually persist the object. To persist an object you need to use node stream to write to file systems or transfer a flattened object across a network wire and have it be rebuilt on the other side. You can use java.io.ObjectOutputStream class, a filter stream which is a wrapper around a lower-level byte stream.

So to write an object you use "writeObject(<<instance>>)" method of "java.io.ObjectOutputStream" class and to read an object you use "readObject()" method of "java.io.ObjectOutputStream" class. "readObject()" can read only serialized object, that means if the class does not implement "java.io.Serializable" interface, "readObject()" cannot read that object.

//Class to persist the time in a flat file time.ser

public class PersistSerialClass {

public static void main(String [] args) {

String filename = "time.ser";

if(args.length > 0){

filename = args[0];

}

PersistSerialClass time = new PersistSerialClass();

FileOutputStream fos = null;

ObjectOutputStream out = null;

try{

fos = new FileOutputStream(filename);

out = new ObjectOutputStream(fos);

out.writeObject(time);

out.close();

}catch(IOException ex){

ex.printStackTrace();

}

}

}

//Class to read the time from a flat file time.ser

public class ReadSerialClass {

public static void main(String [] args) {

String filename = "time.ser";

if(args.length > 0){

filename = args[0];

}

PersistSerialClass time = null;

FileInputStream fis = null;

ObjectInputStream in = null;

try{

fis = new FileInputStream(filename);

in = new ObjectInputStream(fis);

time = (PersistSerialClass)in.readObject();

in.close();

}catch(IOException ex){

ex.printStackTrace();

}catch(ClassNotFoundException cnfe){

cnfe.printStackTrace();

}

// print out restored time

System.out.println("Restored time: " + time.getTime());

// print out the current time

System.out.println("Current time: "

+ Calendar.getInstance().getTime());

}

}

When you serialize an object, only the object's state will be saved, not the object's class file or methods.

When you serialize the above example class, the serialized class will look like below. Surprising.. isn't it? Yes, when you serialized a 2 byte object, you see 51 bytes serialized file. How did it convert to 51 bytes file? To know this,

*Let's see step by step on how the object is serialized and de-serialized.*

So when an object is serailized

* First it writes out the serialization stream magic data - What is serialization stream magic data? This is nothing but the STREAM\_MAGIC and STREAM\_VERSION data (static data) so that JVM can deserialize it when it has to. The STRAM\_MAGIC will be "AC ED" and the STREAM\_VERSION will be the version of the JVM.
* Then it writes out the metadata (description) of the class associated with an instance. So in the below example after writing out the magic data, it writes out the description of "SerialClass" class. What does this description include? It includes the length of the class, the name of the class, serialVersionUID (or serial version), various flags and the number of fields in this class.
* Then it recursively writes out the metadata of the superclass until it finds java.lang.object. Again in the below example, it writes out the description of "SerialParent" and "SerialParentParent" classes.
* Once it finishes writing the metadata information, it then starts with the actual data associated with the instance. But this time, it starts from the top most superclass. So it starts from "SerialParentParent", then writes out "SerialParent".
* Finally it writes the data of objects associated with the instance starting from metadata to actual content. So here it starts writing the metadata for the class Contain and then the contents of it as usual recursively.

class SerialParentParent implements Serializable {

int parentParentVersion = 10;

}

class SerialParent implements Serializable {

int parentVersion = 11;

}

class Contain implements Serializable{

int containVersion = 20;

}

public class SerialClass extends SerialParent implements Serializable {

int version = 12;

Contain con = new Contain();

public int getVersion() {

return version;

}

public static void main(String args[]) throws IOException {

FileOutputStream fos = new FileOutputStream("temp.ser");

ObjectOutputStream oos = new ObjectOutputStream(fos);

SerialClass st = new SerialClass();

oos.writeObject(st);

oos.flush();

oos.close();

}

}

*How to customize the default protocol?*

MMmmmm.. Now its getting more interesting. Lets say, you need to perform some specific operations in the constructor when you are instantiating the class but you cant perform those operations when you deserialize the object because constructor wont be called when an object is de-serialized. Here we are restoring an object but not reconstructing an object. Then how will you call or perform those operations when you desrialize the object? Well, you have a way here and its simple too. You can enhance the normal process by providing two methods inside your serializable class. Those methods are:  
  
*private void writeObject(ObjectOutputStream out) throws IOException;*   
  
*private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException;*   
  
Notice that both methods are declared private and ofcourse they must be declared private, proving that neither method is inherited and overridden or overloaded. The trick here is that the virtual machine will automatically check to see if either method is declared during the corresponding method call. The virtual machine can call private methods of your class whenever it wants but no other objects can. Thus, the integrity of the class is maintained and the serialization protocol can continue to work as normal.

public class SerialClass implements Serializable {

private Date currentTime;

public SerialClass() {

calculateCurrentTime();

}

public Date getCurrentTime() {

return currentTime;

}

private Date calculateCurrentTime(){

currentTime = Calendar.getInstance().getTime();

}

private void writeObject(ObjectOutputStream out) throws IOException {

out.defaultWriteObject();

}

private void readObject(ObjectInputStream in)

throws IOException, ClassNotFoundException{

// our "pseudo-constructor"

in.defaultReadObject();

// now perfrom same operation you need to do in constructor

calculateCurrentTime();

}

}

Ooops. I mentioned earlier that for a class to be serializable either the class should implement "Serializable" interface or one of its super class should implement "Serializable" interface. Now what if you dont want to serialize one of the sub class of a serializable class? You have a way here tooo. To stop the automatic serialization, you can once again use the private methods to just throw the NotSerializableException in your class.

private void writeObject(ObjectOutputStream out) throws IOException{

throw new NotSerializableException("Dont Serialize");

}

private void readObject(ObjectInputStream in) throws IOException{

throw new NotSerializableException("Dont Serialize");

}

*Well... One more way to serialize the object - the Externalizable Interface*

Again there is one more way to serialize the object - create your own protocol with the Externalizable interface. Instead of implementing the Serializable interface, you can implement Externalizable, which contains two methods:   
  
*public void writeExternal(ObjectOutput out) throws IOException;*   
  
*public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException;*   
  
The Externalization is discussed as separate topic. Check it out [here](http://www.jusfortechies.com/java/core-java/externalization.php) or check the menu.

*How not to serialize some fields in a serializble object?*

Sometimes you dont want to serialize/store all the fields in the object. Say some fields you want to hide to preserve the privacy or some fields you may want to read only from master data, then you dont seriaalize them. To do this, you just need to declare a field as ***transient*** field.

transient private int checkPoint;

Also the static fields are not serialized. Actually there is no point in serializing static fields as static fields do not represent object state but represent class state and it can be modified by any other object. Lets assume that you have serialized a static field and its value and before deserialization of the object, the static field value is changed by some other object. Now the static field value that is serialized/stored is no more valid. Hence it make no point in serializing the static field.

Apart from declaring the field as transient, there is another tricky way of controlling what fields can be serialized and what fields cannot be. This is by overriding the writeObject() method while serialization and inside this method you are responsible for writing out the appropriate fields. When you do this, you may have to override readObject() method as well. This sound similar to using Externalizable where you will write writeExternal() and readExternal() methods but anyways lets not take this route as this is not a neat route.

Note that serialization does not care about access modifiers. It serializes all private, public and protected fields.

*Nonserializable objects*

Earlier we discussed about not serializing certain fields in a serializable object and why it may be needed sometimes. But now lets see why certain objects should not be serialized? As you know, the Object class does not implement Serializable interface and hence any object by default is not serializable. To make an object serializable, the respective class should explicitly implement Serializable interface. However certain system classes defined by java like "Thread", "OutputStream", "Socket" are not serializable. Why so? Lets take a step back - now what is the use of serializing the Thread running in System1 JVM using System1 memory and then deserializing it in System2 and trying to run in System2 JVM. Makes no sense right! Hence these classes are not serializable.

Ok. So far so good. Now what if you want to serialize an object containing an instance of Thread? Simple. Declare the Thread instance as transient.

public class SerialClass implements Serializable, Runnable {

transient private Thread myThread;

public PersistentAnimation(int animationSpeed) {

...

...

}

}

*Versioning issues*

One very important item to look at is the versioning issue. Sometimes you wil get "java.io.InvalidClassException" but when you check the class (it will be Serializable class), you will find nothing wrong with it. Then what is causing this exception to be thrown? Ok. Here it is. You create a Serializable class, instantiate it, and write it out to an object stream. That flattened object sits in the file system for some time. Meanwhile, you update the class file by adding a new field. Then try to read the flattened object. InvalidClassException is thrown because all persistent-capable classes are automatically given a unique identifier. If the identifier of the class does not equal the identifier of the flattened object, the exception will be thrown and when you update the class with a new field, a new identifier will be generated.

To fix this issue, manually add the identifier to the class. The identifier that is part of all classes is maintained in a field called serialVersionUID. If you wish to control versioning, you simply have to provide the serialVersionUID field manually and ensure it is always the same, no matter what changes you make to the classfile. More about it is discussed in separate topic. [Check it here.](http://www.jusfortechies.com/java/core-java/serialVersionUID.php)

*Performance Issues/Improvement with Serialization*

The default way of implementing the serialization (by implementing the Serializable interface) has performance glitches. Say you have to write an object 10000 times in a flat file through serialization, this will take much more (alomost double) the time than it takes to write the same object 10000 times to console. To overcome this issue, its always better to write your custom protocol instead of going for default option.

Also always note to close the streams (object output and input streams). The object references are cached in the output stream and if the stream is not closed, the system may not garbage collect the objects written to a stream and this will result in poor performance.

Using Serialization always have performance issues? Nope... Let me give you a situation where it is used for performance improvement. Lets assume you have an application that should display a map and pointing to different areas in the map should highlight those areas in different color. Since all these are images, when you point to each location, loading an image each time will slow the application heavily. To resolve this issue, serialization can be used. So here since the images wont change, you can serialize the image object and the respective points on the map (x and y co-ordinates) and deserialize it as and when necessary. This improves the performance greatly.

*What happens to inner classes? We forgot all about it.*

Yes, you can serialize inner classes by implementing the Serializable interface but it has some problems. Inner classes (declared in a non-static context) will always contain implicit references to their enclosing classes and these references are always non-transient. So, while object serialization process of inner classes, the enclosing classes will also be serialized. Now the problem is that the synthetic fields generated by Java compilers to implement inner classes are pretty much implementation dependent and hence we may face compatibility issues while deserialization on a different platform having a .class file generated by a different Java compiler. The default serialVerionUID may also be different in such cases. Not only this, the names assigned to the local and anonymous inner classes are also implementation dependent. Thus, we see that object serialization of inner classes may pose some unavoidable compatibility issues and hence the serialization of inner classes is strongly discouraged.

**serialVersionUID?**

When you serialize an object using [Serialization](http://www.jusfortechies.com/java/core-java/serialization.php) mechanism (by implementing Serializable interface), there is a possibility that you may face versioning issues and because of these versioning issues, you will not be able to deserialize the object. Thats not a good thing. But first, what is this versioning issue that is troubling your serialization process?

Well, lets say you created a class, instantiated it, and wrote it out to an object stream. That flattened object sits in the file system for some time. Meanwhile, you update the class file, perhaps adding a new field. Now try to read the flattened object. hmmmm.. An exception "java.io.InvalidClassException" will be thrown. You dont understand where it went wrong because the changes in the class seem perfectly fine for you.

*What is serialVersionUID?*

Before we start discussing about the solution for this problem, lets first see what is actually causing this problem? Why should any change in a serialized class throw InvalidClassException? During object serialization, the default Java serialization mechanism writes the metadata about the object, which includes the class name, field names and types, and superclass. All this information is stored as part of the serialized object. When you deserialize the object, this information is read to reconsitute the object. But to perform the deserialization, the object needs to be identified first and this will be done by serialVersionUID. So everytime an object is serialized the java serialization mechanism automatically computes a hash value using ObjectStreamClass’s computeSerialVersionUID() method by passing the class name, sorted member names, modifiers, and interfaces to the secure hash algorithm (SHA), which returns a hash value, the serialVersionUID.

Now when the serilaized object is retrieved, the JVM first evaluates the serialVersionUID of the serialized class and compares the serialVersionUID value with the one of the object. If the sserialVersionUID 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.

The above issue not only occurs when the object is flattened and saved but also when the object is flattened and sent to other JVMs when you implement RMI. Lets assume you have a client/server environment where client is using SUN's JVM in windows while server is using JRockit in Linux. Client sends a serializable class with default generated serialVersionUID (e.g 123L) to server over socket, while server may generate a different serialVersionUID (e.g 124L) during deserialization process, and raise an unexpected InvalidClassExceptions. Since the default serialVersionUID computation is highly sensitive to class details and may vary from different JVM implementation, an unexpected InvalidClassExceptions will result here.

*What's the solution for this versioning issue?*

The solution is very simple. Instead of relying on the JVM to generate the serialVersionUID, you explicitly mention (generate) the serialVersionUID in your class. The syntax is:   
  
private final static long serialVersionUID = <integer value>   
  
Yes, its a static, private variable in the class. Once you define the serialVersionUID in your class explicitly, you dont need to update it until and unless you make the incompatible changes. Look at the example below that explains the issue and importance of maintaining serialVersionUID.

class TestSUID implements Serializable {

private static final long serialVersionUID = 1L;

private int someId;

public TestSUID (int someId) {

this.someId = someId;

}

public int getSomeId() {

return someId;

}

}

public class SUIDTester {

public static void main(String args[]) throws Exception {

File file = new File("temp.ser");

FileOutputStream fos = new FileOutputStream(file);

ObjectOutputStream oos = new ObjectOutputStream(fos);

TestSUID writeSUID = new TestSUID(1);

oos.writeObject(writeSUID);

oos.close();

FileInputStream fis = new FileInputStream(file);

ObjectInputStream ois = new ObjectInputStream(fis);

TestSUID readSUID = (TestSUID) ois.readObject();

System.out.println("someId : " + readSUID.getSomeId());

ois.close();

}

}

In this example, we have created a Serializable class with serialVersionUID = 1L and saved the "some id" value in the "temp.ser" file. Now change the serialVersionUID value of "TestSUID" class to 2L and try to just read the "temp.ser" file. It will throw "InvalidClassException". The reason is the version change and exactly this is the reason for maintaining the version.

Exception in thread "main" java.io.InvalidClassException:

SerializeMe; local class incompatible: stream classdesc

serialVersionUID = 1, local class serialVersionUID = 2

*When should you update serialVersionUID?*

Adding serialVersinUID manually to the class does not mean that it should never be updated and never need not be updated. There is no need to update the serialVersionUID if the change in the class is compatible but it should be updated if the change is incompatible. What are compatible and incompatible changes? A compatible change is a change that does not affect the contract between the class and the callers.

The compatible changes to a class are handled as follows:

* Adding fields - When the class being reconstituted has a field that does not occur in the stream, that field in the object will be initialized to the default value for its type. If class-specific initialization is needed, the class may provide a readObject method that can initialize the field to nondefault values.
* Adding classes - The stream will contain the type hierarchy of each object in the stream. Comparing this hierarchy in the stream with the current class can detect additional classes. Since there is no information in the stream from which to initialize the object, the class’s fields will be initialized to the default values.
* Removing classes - Comparing the class hierarchy in the stream with that of the current class can detect that a class has been deleted. In this case, the fields and objects corresponding to that class are read from the stream. Primitive fields are discarded, but the objects referenced by the deleted class are created, since they may be referred to later in the stream. They will be garbage-collected when the stream is garbage-collected or reset.
* Adding writeObject/readObject methods - If the version reading the stream has these methods then readObject is expected, as usual, to read the required data written to the stream by the default serialization. It should call defaultReadObject first before reading any optional data. The writeObject method is expected as usual to call defaultWriteObject to write the required data and then may write optional data.
* Removing writeObject/readObject methods - If the class reading the stream does not have these methods, the required data will be read by default serialization, and the optional data will be discarded.
* Adding java.io.Serializable - This is equivalent to adding types. There will be no values in the stream for this class so its fields will be initialized to default values. The support for subclassing nonserializable classes requires that the class’s supertype have a no-arg constructor and the class itself will be initialized to default values. If the no-arg constructor is not available, the InvalidClassException is thrown.
* Changing the access to a field - The access modifiers public, package, protected, and private have no effect on the ability of serialization to assign values to the fields.
* Changing a field from static to nonstatic or transient to nontransient - When relying on default serialization to compute the serializable fields, this change is equivalent to adding a field to the class. The new field will be written to the stream but earlier classes will ignore the value since serialization will not assign values to static or transient fields.

Incompatible changes to classes are those changes for which the guarantee of interoperability cannot be maintained. The incompatible changes that may occur while evolving a class are:

* Deleting fields - If a field is deleted in a class, the stream written will not contain its value. When the stream is read by an earlier class, the value of the field will be set to the default value because no value is available in the stream. However, this default value may adversely impair the ability of the earlier version to fulfill its contract.
* Moving classes up or down the hierarchy - This cannot be allowed since the data in the stream appears in the wrong sequence.
* Changing a nonstatic field to static or a nontransient field to transient - When relying on default serialization, this change is equivalent to deleting a field from the class. This version of the class will not write that data to the stream, so it will not be available to be read by earlier versions of the class. As when deleting a field, the field of the earlier version will be initialized to the default value, which can cause the class to fail in unexpected ways.
* Changing the declared type of a primitive field - Each version of the class writes the data with its declared type. Earlier versions of the class attempting to read the field will fail because the type of the data in the stream does not match the type of the field.
* Changing the writeObject or readObject method so that it no longer writes or reads the default field data or changing it so that it attempts to write it or read it when the previous version did not. The default field data must consistently either appear or not appear in the stream.
* Changing a class from Serializable to Externalizable or visa-versa is an incompatible change since the stream will contain data that is incompatible with the implementation in the available class.
* Removing either Serializable or Externalizable is an incompatible change since when written it will no longer supply the fields needed by older versions of the class.
* Adding the writeReplace or readResolve method to a class is incompatible if the behavior would produce an object that is incompatible with any older version of the class.

*How to generate a serialVersionUID?*

There are two ways to generate the serialVersionUID.

* Go to commanline and type "serialver <>. SerialVersionUID wil be generated. Copy, paste the same into your class.   
  In Windows, generate serialVersionUID using the JDK's graphical tool like so : use Control Panel | System | Environment to set the classpath to the correct directory  
  run serialver -show from the command line  
  point the tool to the class file including the package, for example, finance.stock.Account - without the .class  
  (here are the serialver docs for both Win and Unix)
* One way is through Eclipse IDE. After you implement Serializable interface and save the class, eclipse will show a warning asking you to add the serialVersionUID and it provides you the option to generate it or use the default one. Click on the link to generate the serialVersionUID and it will generate it for you and adds it to the class.

*Finally few guidelines for serialVersionUID :*

* always include it as a field, for example: "private static final long serialVersionUID = 7526472295622776147L; " include this field even in the first version of the class, as a reminder of its importance
* do not change the value of this field in future versions, unless you are knowingly making changes to the class which will render it incompatible with old serialized objects
* new versions of Serializable classes may or may not be able to read old serialized objects; it depends upon the nature of the change; provide a pointer to Sun's guidelines for what constitutes a compatible change, as a convenience to future maintainers

**Externalization in Java**

Before going into what externalization is, you need to have some knowledge on what [serialization](http://www.jusfortechies.com/java/core-java/serialization.php) is because externalization is nothing but serialization but an alternative for it and Externalizable interface extends Serializable interface. Check [Serialization](http://www.jusfortechies.com/java/core-java/serialization.php) article for information on serialization. Just as an overview, Serialization is the process of converting an object's state (including its references) to a sequence of bytes, as well as the process of rebuilding those bytes into a live object at some future time. Serialization can be achieved by an object by implementing Serializable interface or Externalizable interface.

*Well, when serialization by implementing Serializable interface is serving your purpose, why should you go for externalization?*

Good question! Serializing by implementing Serializable interface has some issues. Lets see one by one what they are.

* Serialization is a recursive algorithm. What I mean to say here is, apart from the fields that are required, starting from a single object, until all the objects that can be reached from that object by following instance variables, are also serialized. This includes the super class of the object until it reaches the "Object" class and the same way the super class of the instance variables until it reaches the "Object" class of those variables. Basically all the objects that it can read. This leads to lot of overheads. Say for example, you need only car type and licence number but using serialization, you cannot stop there. All the information that includes description of car, its parts, blah blah will be serialized. Obviously this slows down the performance.
* Both serializing and deserializing require the serialization mechanism to discover information about the instance it is serializing. Using the default serialization mechanism, will use reflection to discover all the field values. Also the information about class description is added to the stream which includes the descption of all the serializable superclasses, the description of the class and the instance data associated with the specific instance of the class. Lots of data and metadata and again performance issue.
* You know that serialization needs serialVersionUID, a unique Id to identify the information persisted. If you dont explicitly set a serialiVersionUID, serialization will compute the serialiVersionUID by going through all the fields and methods. So based on the size of the class, again serialization mechanism takes respective amount of time to calculate the value. A third performance issue.

Above three points confirm serialization has performance issues. Apart from performance issues,

* When an object that implements Serializable interface, is serialized or de-serialized, no constructor of the object is called and hence any initialization which is done in the constructor cannot be done. Although there is an alternative of writing all initialization logic in a separate method and call it in constructor and readObject methods so that when an object is created or deserialized, the initialization process can happen but it definitely is a messy approach.

The solution for all the above issues is ***Externalization***. Cool. Here enters the actual topic.

*So what is externalization?*

Externalization is nothing but serialization but by implementing Externalizable interface to persist and restore the object. To externalize your object, you need to implement Externalizable interface that extends Serializable interface. Here only the identity of the class is written in the serialization stream and it is the responsibility of the class to save and restore the contents of its instances which means you will have complete control of what to serialize and what not to serialize. But with serialization the identity of all the classes, its superclasses, instance variables and then the contents for these items is written to the serialization stream. But to externalize an object, you need a default public constructor.

Unlike Serializable interface, Externalizable interface is not a marker interface and it provides two methods - writeExternal and readExternal. These methods are implemented by the class to give the class a complete control over the format and contents of the stream for an object and its supertypes. These methods must explicitly coordinate with the supertype to save its state. These methods supersede customized implementations of writeObject and readObject methods.

How serialization happens? JVM first checks for the Externalizable interface and if object supports Externalizable interface, then serializes the object using writeExternal method. If the object does not support Externalizable but implement Serializable, then the object is saved using ObjectOutputStream. Now when an Externalizable object is reconstructed, an instance is created first using the public no-arg constructor, then the readExternal method is called. Again if the object does not support Externalizable, then Serializable objects are restored by reading them from an ObjectInputStream.

Lets see a simple example.

import java.io.\*;

public class Car implements Externalizable {

String name;

int year;

/\*

\* mandatory public no-arg constructor

\*/

public Car() { super(); }

Car(String n, int y) {

name = n;

year = y;

}

/\*\*

\* Mandatory writeExernal method.

\*/

public void writeExternal(ObjectOutput out) throws IOException {

out.writeObject(name);

out.writeInt(year);

}

/\*\*

\* Mandatory readExternal method.

\*/

public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {

name = (String) in.readObject();

year = in.readInt();

}

/\*\*

\* Prints out the fields. used for testing!

\*/

public String toString() {

return("Name: " + name + "\n" + "Year: " + year);

}

}

import java.io.\*;

public class ExternExample {

public static void main(String args[]) {

// create a Car object

Car car = new Car("Mitsubishi", 2009);

Car newCar = null;

//serialize the car

try {

FileOutputStream fo = new FileOutputStream("tmp");

ObjectOutputStream so = new ObjectOutputStream(fo);

so.writeObject(car);

so.flush();

} catch (Exception e) {

System.out.println(e);

System.exit(1);

}

// de-serialize the Car

try {

FileInputStream fi = new FileInputStream("tmp");

ObjectInputStream si = new ObjectInputStream(fi);

newCar = (Car) si.readObject();

}

catch (Exception e) {

System.out.println(e);

System.exit(1);

}

/\*

\* Print out the original and new car information

\*/

System.out.println("The original car is ");

System.out.println(car);

System.out.println("The new car is ");

System.out.println(newCar);

}

}

In this example, class Car implements Externalizable interface which means that car object is ready for serialization. This class have two public methods - "writeExternal" and "readExternal". Unlike Serializable interface which will serialize all the variables in the object with just by implementing the interface, here you have to explicitly mention what fields or variables you want to serialize and the same is done in "writeExternal" and "readExternal" methods. So in the "ExternExample" class, when you write the "Car" object to the OutputStream, the "writeExternal" method is called and the data is persisted. The same applies to "readExternal" method in the Car object i.e., when you read the "Car" object from the ObjectInputStream, "readExternal" method is called.

*What will happen when an externalizable class extends a non externalizable super class?*

Then in this case, you need to persist the super class fields also in the sub class that implements Externalizable interface. Look at this example.

/\*\*

\* The superclass does not implement externalizable

\*/

class Automobile {

/\*

\* Instead of making thse members private and adding setter

\* and getter methods, I am just giving default access specifier.

\* You can make them private members and add setters and getters.

\*/

String regNo;

String mileage;

/\*

\* A public no-arg constructor

\*/

public Automobile() {}

Automobile(String rn, String m) {

regNo = rn;

mileage = m;

}

}

public class Car extends Automobile implements Externalizable {

String name;

int year;

/\*

\* mandatory public no-arg constructor

\*/

public Car() { super(); }

Car(String n, int y) {

name = n;

year = y;

}

/\*\*

\* Mandatory writeExernal method.

\*/

public void writeExternal(ObjectOutput out) throws IOException {

/\*

\* Since the superclass does not implement the Serializable interface

\* we explicitly do the saving.

\*/

out.writeObject(regNo);

out.writeObject(mileage);

//Now the subclass fields

out.writeObject(name);

out.writeInt(year);

}

/\*\*

\* Mandatory readExternal method.

\*/

public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {

/\*

\* Since the superclass does not implement the Serializable interface

\* we explicitly do the restoring

\*/

regNo = (String) in.readObject();

mileage = (String) in.readObject();

//Now the subclass fields

name = (String) in.readObject();

year = in.readInt();

}

/\*\*

\* Prints out the fields. used for testing!

\*/

public String toString() {

return("Reg No: " + regNo + "\n" + "Mileage: " + mileage +

"Name: " + name + "\n" + "Year: " + year );

}

}

Here the Automobile class does not implement Externalizable interface. So to persist the fields in the automobile class the writeExternal and readExternal methods of Car class are modified to save/restore the super class fields first and then the sub class fields.

*Sounds good! What if the super class implements the Externalizable interface?*

Well, in this case the super class will also have the readExternal and writeExternal methods as in Car class and will persist the respective fields in these methods.

import java.io.\*;

/\*\*

\* The superclass implements externalizable

\*/

class Automobile implements Externalizable {

/\*

\* Instead of making thse members private and adding setter

\* and getter methods, I am just giving default access specifier.

\* You can make them private members and add setters and getters.

\*/

String regNo;

String mileage;

/\*

\* A public no-arg constructor

\*/

public Automobile() {}

Automobile(String rn, String m) {

regNo = rn;

mileage = m;

}

public void writeExternal(ObjectOutput out) throws IOException {

out.writeObject(regNo);

out.writeObject(mileage);

}

public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {

regNo = (String)in.readObject();

mileage = (String)in.readObject();

}

}

public class Car extends Automobile implements Externalizable {

String name;

int year;

/\*

\* mandatory public no-arg constructor

\*/

public Car() { super(); }

Car(String n, int y) {

name = n;

year = y;

}

/\*\*

\* Mandatory writeExernal method.

\*/

public void writeExternal(ObjectOutput out) throws IOException {

// first we call the writeExternal of the superclass as to write

// all the superclass data fields

super.writeExternal(out);

//Now the subclass fields

out.writeObject(name);

out.writeInt(year);

}

/\*\*

\* Mandatory readExternal method.

\*/

public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {

// first call the superclass external method

super.readExternal(in);

//Now the subclass fields

name = (String) in.readObject();

year = in.readInt();

}

/\*\*

\* Prints out the fields. used for testing!

\*/

public String toString() {

return("Reg No: " + regNo + "\n" + "Mileage: " + mileage +

"Name: " + name + "\n" + "Year: " + year );

}

}

In this example since the Automobile class stores and restores its fields in its own writeExternal and readExternal methods, you dont need to save/restore the superclass fields in sub class but if you observe closely the writeExternal and readExternal methods of Car class closely, you will find that you still need to first call the super.xxxx() methods that confirms the statement the externalizable object must also coordinate with its supertype to save and restore its state.

*Lets see the difference in sizes when you serialize using Serializable interface and serialize using Externalizable interface*

Let's take a simple case, an object of type SimpleClass with just few fields - firstName, lastName, weight and location, containing data {"Brad", "Pitt", 180.5, {49.345, 67.567}}. When you serialize this object that is about 24 bytes by implementing Serializable interface, it turns into 220 bytes (approx). As it turns out, the basic serialization mechanism stores all kinds of information in the file so that it can deserialize without any other assistance. Look at the format below when the object is serialized and you will understand why it is turned out to 200 bytes.

|  |
| --- |
| Length: 220  Magic: ACED  Version: 5  OBJECT  CLASSDESC  Class Name: "SimpleClass"  Class UID: -D56EDC726B866EBL  Class Desc Flags: SERIALIZABLE;  Field Count: 4  Field type: object  Field name: "firstName"  Class name: "Ljava/lang/String;"  Field type: object  Field name: "lastName"  Class name: "Ljava/lang/String;"  Field type: float  Field name: "weight"  Field type: object  Field name: "location"  Class name: "Ljava/awt/Point;"  Annotation: ENDBLOCKDATA  Superclass description: NULL  STRING: "Brad"  STRING: "Pitt"  float: 180.5  OBJECT  CLASSDESC  Class Name: "java.awt.Point"  Class UID: -654B758DCB8137DAL  Class Desc Flags: SERIALIZABLE;  Field Count: 2  Field type: integer  Field name: "x"  Field type: integer  Field name: "y"  Annotation: ENDBLOCKDATA  Superclass description: NULL  integer: 49.345  integer: 67.567 |

Now if you serialize the same by extending Externalizable interface, the size will be reduced drastically and the information saved in the persistant store is also reduced a lot. Here is the result of serializing the same class, modified to be externalizable. Notice that the actual data is not parseable externally any more--only your class knows the meaning of the data!

|  |
| --- |
| Length: 54  Magic: ACED  Version: 5  OBJECT  CLASSDESC  Class Name: "SimpleClass"  Class UID: 5CB3777417A3AB5BL  Class Desc Flags: EXTERNALIZABLE;  Field Count: 0  Annotation  ENDBLOCKDATA  Superclass description  NULL  EXTERNALIZABLE:  [70 00 04 4D 61 72 6B 00 05 44 61 76 69 73 43 3C  80 00 00 00 00 01 00 00 00 01] |

*Well, externalization has its own limitations*

Externalization efficiency comes at a price. The default serialization mechanism adapts to application changes due to the fact that metadata is automatically extracted from the class definitions (observe the format above and you will see that when the object is serialized by implementing Serializable interface, the class metadata(definitions) are written to the persistent store while when you serialize by implementing Externalizable interface, the class metadata is not written to the persistent store). Externalization on the other hand isn't very flexible and requires you to rewrite your marshalling and demarshalling code whenever you change your class definitions.

As you know a default public no-arg constructor will be called when serializing the objects that implements Externalizable interface. Hence, Externalizable interface can't be implemented by Inner Classes in Java as all the constructors of an inner class in Java will always accept the instance of the enclosing class as a prepended parameter and therefore you can't have a no-arg constructor for an inner class. Inner classes can achieve object serialization by only implementing Serializable interface.

If you are subclassing your externalizable class, you have to invoke your superclass’s implementation. So this causes overhead while you subclass your externalizable class. Observe the examples above where the superclass writeExternal method is explicitly called in the subclass writeExternal method.

Methods in externalizable interface are public. So any malicious program can invoke which results into loosing the prior serialized state.

Once your class is tagged with either Serializable or Externalizable, you can't change any evolved version of your class to the other format. You alone are responsible for maintaining compatibility across versions. That means that if you want the flexibility to add fields in the future, you'd better have your own mechanism so that you can skip over additional information possibly added by those future versions.

*So much of it. Here are some final tips for serialization.*

You can decide whether to implement Externalizable or Serializable on a class-by-class basis. Within the same application, some of your classes can be Serializable, and some can be Externalizable. This makes it easy to evolve your application in response to actual performance data and shifting requirements. You can do the following thing:   
\* Make all your classes implement Serializable.  
\* Then make some of them, the ones you send often and for which serialization is inefficient, implement Externalizable instead.

To reduce memory size:   
\* Write primitives or Strings directly. For example, instead of writing out a contained object, Point (in SimpleClass, we have a field of type Point), write out each of its integer coordinates separately. When you read them in, create a new Point from the two integers. This can be very significant in terms of size: an array of three Points takes 117 bytes; an array of 6 ints takes 51 bytes.   
\* Strings are special-cased and don't carry much of the object overhead; you will normally use them as is. However, the serialized representation of a String is UTF, which works great for ASCII characters, is neutral for most European characters, but causes a 50% increase in size for Japanese and other scripts. If you have significant strings of Asian text you better serialize a char array instead.