When u create an object, it exists for as long as u need it, but under no circumstances does it exist when the program terminates. While this makes sense at first, there are situations in which it would be incredibly useful if an object could exist and hold its information even while the program wasn’t running. Then the next time u started the program, the object would be there and it would have the same information it had, the previous time the program was running. Of course, u can get a similar effect by writing the information to a file or to a database, but in the spirit of making everything an object, it would be quite convenient to declare an object to be “persistent” and have all the details taken care of for u.

Java’s Object Serialization allows u to take an “object” that implements “Serializable” interface and turn it into sequence of bytes , that can later be fully restored to regenerate the original object. This is even true across a network , which means that the serialization mechanism automatically compensate for difference in operating system. That is , u can create an object on a “Windows” machine , serialize it and send it across the network to a “Unix” machine, where it will be correctly reconstructed. U don’t have to worry about data representation on the different machines , the bytes ordering or any other details.

Serialization is the process of converting an object's state (including its entire graph) 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.

Object serialization was added to the language to support two major features. **Java’s RMI and Java Beans.**

When **JavaBean** is used , it’s state information is generally configured at design time. This state information must be stored and later recovered when the program is started. Object serialization performs this task.

In case of **RMI** , when object is sent over a network , its copy is sent through serialization mechanism.

Examples of Serialization

*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 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 migrate user session in Web applications.
* To activate/passivate enterprise java beans.
* To send objects between the servers in a cluster.

*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 does 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.

class SerialParentParent implements Serializable {

int parentParentVersion = 10;

}

class SerialParent extends SerialParentParent {

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();

}

}

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

So when an object is serialized

* First it writes out the metadata (description) of the class associated with an instance. So in the above example, 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 above 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.

*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;*   
  
*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 serialize 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.

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