Serialization & Externalization

# Serialization

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.

The ObjectOutputStream class contains many write methods for writing various data types, but one method in particular stands out:

public final void writeObject(Object x) throws IOException

The above method serializes an Object and sends it to the output stream. Similarly, the ObjectInputStream class contains the following method for deserializing an object:

public final Object readObject() throws IOException, ClassNotFoundException

This method retrieves the next Object out of the stream and deserializes it. The return value is Object, so you will need to cast it to its appropriate data type.

To demonstrate how serialization works in Java, I am going to use the Employee class that we discussed early on in the book. Suppose that we have the following Employee class, which implements the Serializable interface:

public class Employee implements java.io.Serializable

{

public String name;

public String address;

public transient int SSN;

public int number;

public void mailCheck()

{

System.out.println("Mailing a check to " + name + " " + address);

}

}

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 are curious to know if a Java Standard Class is serializable or not, check the documentation for the class. The test is simple: If the class implements java.io.Serializable, then it is serializable; otherwise, it's not.

**Serializing an Object:**

The ObjectOutputStream class is used to serialize an Object. The following SerializeDemo program instantiates an Employee object and serializes it to a file.

When the program is done executing, a file named employee.ser is created. The program does not generate any output, but study the code and try to determine what the program is doing.

**Note:** When serializing an object to a file, the standard convention in Java is to give the file a **.ser**extension.

import java.io.\*;

public class SerializeDemo

{

public static void main(String [] args)

{

Employee e = new Employee();

e.name = "Reyan Ali";

e.address = "Phokka Kuan, Ambehta Peer";

e.SSN = 11122333;

e.number = 101;

try

{

FileOutputStream fileOut =

new FileOutputStream("/tmp/employee.ser");

ObjectOutputStream out = new ObjectOutputStream(fileOut);

out.writeObject(e);

out.close();

fileOut.close();

System.out.printf("Serialized data is saved in /tmp/employee.ser");

}catch(IOException i)

{

i.printStackTrace();

}

}

}

**Deserializing an Object:**

The following DeserializeDemo program deserializes the Employee object created in the SerializeDemo program. Study the program and try to determine its output:

import java.io.\*;

public class DeserializeDemo

{

public static void main(String [] args)

{

Employee e = null;

try

{

FileInputStream fileIn = new FileInputStream("/tmp/employee.ser");

ObjectInputStream in = new ObjectInputStream(fileIn);

e = (Employee) in.readObject();

in.close();

fileIn.close();

}catch(IOException i)

{

i.printStackTrace();

return;

}catch(ClassNotFoundException c)

{

System.out.println("Employee class not found");

c.printStackTrace();

return;

}

System.out.println("Deserialized Employee...");

System.out.println("Name: " + e.name);

System.out.println("Address: " + e.address);

System.out.println("SSN: " + e.SSN);

System.out.println("Number: " + e.number);

}

}

**This would produce the following result:**

Deserialized Employee...

Name: Reyan Ali

Address:Phokka Kuan, Ambehta Peer

SSN: 0

Number:101

**Here are following important points to be noted:**

The try/catch block tries to catch a ClassNotFoundException, which is declared by the readObject() method. For a JVM to be able to deserialize an object, it must be able to find the bytecode for the class. If the JVM can't find a class during the deserialization of an object, it throws a ClassNotFoundException.

Notice that the return value of readObject() is cast to an Employee reference.

The value of the SSN field was 11122333 when the object was serialized, but because the field is transient, this value was not sent to the output stream. The SSN field of the deserialized Employee object is 0.

To save the state of a transient variable:

Make 2 methods as shown below:

**private void writeObject(ObjectOutputStream os)** {

// throws IOException { // 1

try {

**os.defaultWriteObject();** // 2

os.writeInt(theCollar.getCollarSize()); // 3

} catch (Exception e) { e.printStackTrace(); }

}

**private void readObject(ObjectInputStream is) {**

// throws IOException, ClassNotFoundException { // 4

try {

**is.defaultReadObject();** // 5

theCollar = new Collar(is.readInt()); // 6

} catch (Exception e) { e.printStackTrace(); }

}

}

## SerialVersionUID

# Making a class Serializable in Java is very easy, Your Java class just needs to 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 don't 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. **serialVersionUID in Java Serialization**

serialVersionUID is used to ensure that during deserialization the same class (that was used during serialize process) is loaded. This is a one line definition to explain why a serialVersionUID is used?

Apart from the above definition there are quite  a few things to learn from this serialVersionUID. As per javadocs, following is format of serialVersionUID:

## **serialVersionUID Syntax**

ANY-ACCESS-MODIFIER static final long serialVersionUID = 42L;

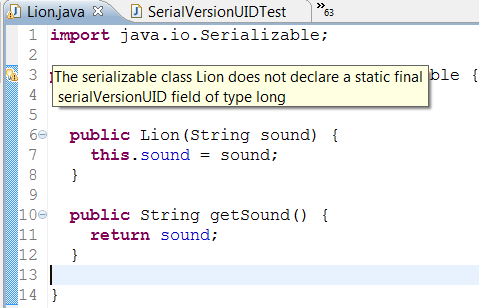
* serialVersionUID is a static final field. You can assign any number of your choice to it. Later I will explain the significance of these two statements.

## **Why serialVersionUID?**

Lets start with annoying warning message you get in your IDE when you declare a class as Serializable.

*The serializable class Lion does not declare a static final serialVersionUID field of type long*

Most of us used to ignore this message as we always do for a warning. My general note is, always pay attention to the java warning messages. It will help you to learn a lot of fundamentals.



serialVersionUID is a must in serialization process. But it is optional for the developer to add it in java source file. If you are not going to add it in java source file, serialization runtime will generate a serialVersionUID and associate it with the class. The serialized object will contain this serialVersionUID along with other data.

Even though serialVersionUID is a static field, it gets serialized along with the object. This is one exception to the general serialization rule that, “static fields are not serialized”.

## **How serialVersionUID works?**

When an object is serialized, the serialVersionUID is serialized along with the other contents.

Later when that is deserialized, the serialVersionUID from the deserialized object is extracted and compared with the serialVersionUID of the loaded class.

If the numbers do not match then, InvalidClassException is thrown.

If the loaded class is not having a serialVersionUID declared, then it is automatically generated using the same algorithm as before.

## **Strongly recommended to declare serialVersionUID**

Javadocs says,

*“the default serialVersionUID computation is highly sensitive to class details that may vary depending on compiler implementations, and can thus result in unexpected InvalidClassExceptions during deserialization”*

Now you know why we should declare a serialVersionUID.

Not only declaring a serialVersionUID is sufficient. You must do the following two things carefully. Otherwise it defeats the purpose of having the serialVersionUID.

serialVersionUID should be maintained. As and when you change anything in the class, you should upgrade the serailVersionUID.  
Try your best to declare a unique serialVersionUID.

## **Demonstrate serialVersionUID**

Initial class to be serialized has a serialVersionUID as 1L.

import java.io.Serializable;

public class Lion implements Serializable {

private static final long serialVersionUID = 1L;

private String sound;

public Lion(String sound) {

this.sound = sound;

}

public String getSound() {

return sound;

}

}

# 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);

}

}