Tutorial Link https://course.testpad.chitkara.edu.in/tutorials/Input Output in Java : Object Serialization and Object Streams/5b766e964227cf4f24a3070b

TUTORIAL

# Input Output in Java : Object Serialization and Object Streams

Topics

1.2 ObjectInputStream & ObjectOutputStream

1.5 Serializable & Externalizable Interfaces

1.6 Externalizable Interface

Data streams (DataInputStream and DataOutputStream) allow you to read and write primitive data (such as int, double) and String, rather than individual bytes. Object streams (ObjectInputStream and ObjectOutputStream) go one step further to allow you to read and write entire objects (such as Date, ArrayList or any custom objects).

Object serialization is the process of representing a "particular state of an object" in a serialized bit-stream, so that the bit stream can be written out to an external device (such as a disk file or network). The bit-stream can later be re-constructed to recover the state of that object. Object serialization is necessary to save a state of an object into a disk file for persistence or sent the object across the network for applications such as Web Services, Distributed-object applications, and Remote Method Invocation (RMI).

In Java, object that requires to be serialized must implement java.io.Serializable or java.io.Externalizable interface. Serializable interface is an empty interface (or tagged interface) with nothing declared. Its purpose is simply to declare that particular object is serializable.

## ObjectInputStream & ObjectOutputStream

The ObjectInputStream and ObjectOutputStream can be used to serialize an object into a bit-stream and transfer it to/from an I/O streams, via these methods:

```
public final Object readObject() throws IOException,
ClassNotFoundException;
public final void writeObject(Object obj) throws IOException;
```

ObjectInputStream and ObjectOutputStream must be stacked on top of a concrete implementation of InputStream or OutputStream, such as FileInputStream or FileOutputStream. For example, the following code segment writes objects to a disk file. The ".ser" is the convention for serialized object file type.

```
ObjectOutputStream out =
   new ObjectOutputStream(
      new BufferedOutputStream(
         new FileOutputStream("object.ser")));
out.writeObject("The current Date and Time is "); // write a
String object
out.writeObject(new Date());                      // write a Date
object
out.flush();
out.close();
```

To read and re-construct the object back in a program, use the method readObject(), which returns an java.lang.Object. Downcast the Object back to its original type.

```
ObjectInputStream in =
   new ObjectInputStream(
      new BufferedInputStream(
         new FileInputStream("object.ser")));
String str = (String)in.readObject();
Date d = (Date)in.readObject(new Date());  // downcast
in.close();
```

Below is an example for object serialization: -

```java
import java.io.*;

class CodeQuotientSerialized implements Serializable
```

```
{
  private String msg;
  public CodeQuotientSerialized(String msg) {
    this.msg = msg;
  }
  public String getMsg() {
    return msg;
  }
}

class Main
{
  public static void main(String[] args)
  {
    String filename = "objects.ser";
    int numObjs = 5;

    try(ObjectOutputStream out = new
ObjectOutputStream( new BufferedOutputStream( new
FileOutputStream(filename))))
    {
      // Create an array of 5 CodeQuotientSerialized
      CodeQuotientSerialized[] objs = new
CodeQuotientSerialized[numObjs];
      for (int i = 0; i < numObjs; ++i)
        objs[i] = new CodeQuotientSerialized("Code
Quotient-"+i); // Starting at AA

      // Write the objects to file, one by one.
      for (int i = 0; i < numObjs; ++i)
        out.writeObject(objs[i]);

      // Also can write the entire array in one go with
      // out.writeObject(objs);
      out.flush();
    } catch (IOException ex) {
      ex.printStackTrace();
    }

    // Read raws bytes and print in Hex
    try (BufferedInputStream in = new
BufferedInputStream( new FileInputStream(filename))) {
      int inByte;
      while ((inByte = in.read()) != -1) {
        System.out.printf("%02X ", inByte);   // Print Hex codes
      }
      System.out.println();System.out.println();
    } catch (IOException ex) {
      ex.printStackTrace();
    }

    // Read objects
    try (ObjectInputStream in = new ObjectInputStream( new BufferedInputStream( new FileInputStream(filename)))) {
      // Read back the objects, cast back to its original type as individual objects.
      CodeQuotientSerialized objIn;
      for (int i = 0; i < numObjs; ++i) {
        objIn = (CodeQuotientSerialized)in.readObject();
        System.out.println(objIn.getMsg());
      }
      // Can be read in as a array of object as below
      /* CodeQuotientSerialized[] objArrayIn;
      objArrayIn = (CodeQuotientSerialized[])in.readObject();
      for (CodeQuotientSerialized o : objArrayIn) {
        System.out.println(o.getMsg());
      } */
    } catch (ClassNotFoundException|IOException ex) {
      ex.printStackTrace();
    }
  }
}

```

Primitive types and array are, by default, serializable. The ObjectInputStream and ObjectOutputStream implement DataInput and DataOutput interface respectively. You can used methods such as readInt(), readDouble(), writeInt(), writeDouble() for reading and writing primitive types.

**transient & static**

- static fields are not serialized, as it belongs to the class instead of the particular instance to be serialized.
- To prevent certain fields from being serialized, mark them using the keyword transient. This could cut down the amount of data traffic.
- The writeObject() method writes out the class of the object, the class signature, and values of non-static and non-transient fields.

## Serializable & Externalizable Interfaces

When you create a class that might be serialized, the class must implement java.io.Serializable interface. The Serializable interface doesn't declare any methods. Empty interfaces such as Serializable are known as tagging interfaces. They identify implementing classes as having certain properties, without requiring those classes to actually implement any methods.

Most of the core Java classes implement Serializable, such as all the wrapper classes, collection classes, and GUI classes. In fact, the only core Java classes that do not implement Serializable are ones that should not be serialized. Arrays of primitives or serializable objects are themselves serializable.

**Warning Message "The serialization class does not declare a static final serialVersionUID field of type long" (Advanced)**

This warning message is triggered because your class (such as java.swing.JFrame) implements the java.io.Serializable interface. This interface enables the object to be written out to an output stream serially (via method writeObject()); and read back into the program (via method readObject()). The serialization runtime uses a number (called serialVersionUID) to ensure that the object read into the program (during deserialization) is compatible with the class definition, and not belonging to another version. It throws an InvalidClassException otherwise. You have these options:

1. Simply ignore this warning message. If a serializable class does not explicitly declare a serialVersionUID, then the serialization runtime

will calculate a default serialVersionUID value for that class based on various aspects of the class.

2. Add a serialVersionUID (Recommended), e.g.

```
private static final long serialVersionUID = 1L;  // verion 1
```

1. Suppress this particular warning via annotation @SuppressWarmomgs (in package java.lang) (JDK 1.5):

```
@SuppressWarnings("serial")
public class MyFrame extends JFrame { ...... }
```

## Externalizable Interface

The Serializable has a sub-interface called Externalizable, which you could used if you want to customize the way a class is serialized. Since Externalizable extends Serializable, it is also a Serializable and you could invoke readObject() and writeObject(). Externalizable declares two abstract methods:

```
void writeExternal(ObjectOutput out) throws IOException
void readExternal(ObjectInput in) throws IOException,
ClassNotFoundException
```

ObjectOutput and ObjectInput are interfaces that are implemented by ObjectOutputStream and ObjectInputStream, which define the writeObject() and readObject() methods, respectively. When an instance of Externalizable is passed to an ObjectOutputStream, the default serialization procedure is bypassed; instead, the stream calls the instance's writeExternal() method. Similarly, when an ObjectInputStream reads a Exteranlizabled instance, it uses readExternal() to reconstruct the instance. Externalizable is useful if you want complete control on how your objects shall be serialized/deserialized. For example, you could encrypt sensitive data before the object is serialized.