Serialization and Deserialization in Java

codementor.io/java/tutorial/serialization-and-deserialization-in-java

Let’s understand this by examining computer games. When we stop or pause a running computer game, it usually starts from the state where it was left off when we play it again. This is possible through the process of Serialization; which is saving the current object state as byte stream into file or database. Then to restore the object’s state when we access the game again is called Deserialization.

Before we dive deeper into it, let’s first understand what streams are in computer systems. A stream is nothing but the sequence of data elements. In a computer system, there is a source that generates data in the form of stream and there is a destination which reads and consumes this stream. Here are the types of streams:

- **Byte Stream**: Byte stream does not have any encoding scheme and it uses 8-bit (or one byte) to store a single character. Byte stream is a low level input output (I/O) operation that can be performed by a JAVA program. To process such byte stream, we need to use a buffered approach for IO operations.
- **Character Stream**: Character stream is composed of the streams of characters with proper encoding scheme such as ASCII, UNICODE, or any other format. Unlike byte stream, the character stream will automatically translate to and from the local character set. JAVA language uses UNICODE system to store characters as character stream.
- **Data Stream**: Data Stream is used to perform binary IO operations for all the primitive data types in Java. We can perform I/O operations in an efficient and convenient way for Boolean, char, byte, short, int, long, float, double, and Strings, etc. by using data stream. It also allows us to read-write Java primitive data types instead of raw bytes.
- **Object Stream**: the state of a JAVA object can be converted into a byte stream that can be stored into a database, file, or transported to any known location like from web tier to app tier as data value object in client-server RMI applications. This process of writing the object state into a byte stream is known as **Serialization**.

Eventually, we can use this byte stream to retrieve the stored values and restore the object’s old state. This process of restoring the object’s old state is known as **Deserialization**.

Similar to a multi-tier JAVA/J2EE application (client-server RMI applications), when we make a remote invocation method or RMI from a web tier to app tier, we need to send the `data v alue object` that transfers the required business information from web tier to app tier after Serialization (we implement `ja va.io.` Serializable (Marker Interface) that we are now going to discuss in detail).

Marker Interface

Marker Interfaces in JAVA are interfaces which do not have any field and method. They are just used to inform the JVM that the class is implementing these interfaces that have special meaning or behavior. Following are the well-known Marker Interfaces.

- `rmi.Remote`
- `io.Serializable`
- `lang.Cloneable`

Note: All Marker interfaces— except serializable interface—are replaced by annotations since JAVA 5.

Data and Object stream Interfaces

The following are the data and object stream interfaces which every Object Stream class implements. Object Stream class implements either of the two interfaces.

- **`ObjectInput`** : It is the sub interface of **`DataInput`**
- **`ObjectOutput`** : It is the sub interface of **`DataOutput`**

Note: All the primitive data I/O methods which are already covered in Data Streams are also implemented in object streams because these interfaces are the sub interface of Data streams interfaces.

Object Streams Classes

The following are the two classes used for Object Streams.

`ObjectInputStream`

- This Java class is responsible for the deserialization of the serialized objects and the primitive data.
- This class helps to read the object from the graph of objects stored while using FileInputStream.
- It has a main method `readObject ()` that is used to deserialize the object. It reads the class of the object, the signature of the class, and the values of the non-transient and non-static fields of the class, and all of its super types. Here is the method.

```
public final Object readObject() throws IOException, ClassNotFoundException
```

`ObjectOutputStream`

- This class is responsible for the serialization of an object. It stores data primitives and a graph of Java object that are available to ObjectInputStream to read data from.
- It has a main `writeObject ()` method that saves the super class and sub class data of a class— or in other words, it serializes the object directly.

```
public final void writeObject(Object object) throws IOException
```

Transient Keyword

In an object, we may not want to save sensitive information such as keys, Password, etc.; so we protect that field from being saved during the process of serialization. Placing the transient keyword before any field of a class object makes sure that the value of such fields won‘t be stored as a part of serialization. And when we deserialize, the value of these transient fields will have the default value (null for a JAVA String primitives). We will understand the complete theory of serialization and deserialization that we just discussed with the help of the following JAVA example.

DataValueObject.java

```
package com.java;

import java.io.Serializable;

public class DataValueObject implements Serializable{

 private static final long serialVersionUID = 1L;
 private String customer;
 private String business;
 transient private String contractID;
 transient private String passKeys;

 public String getCustomer() {
  return customer;
 }
 public void setCustomer(String customer) {
  this.customer = customer;
 }
 public String getBusiness() {
  return business;
 }
 public void setBusiness(String business) {
  this.business = business;
 }
 public String getContractID() {
  return contractID;
 }
 public void setContractID(String contractID) {
  this.contractID = contractID;
 }
 public String getPassKeys() {
  return passKeys;
 }
 public void setPassKeys(String passKeys) {
  this.passKeys = passKeys;
 }
 @Override
    public String toString() {
        String value = "customer : " + customer + "\nbusiness : " + business +
"\ncontractID : " + contractID
                + "\npassKeys : " + passKeys;
        return value;
    }
}
```

Explanation

- It is a Data Value Object class that undergoes serialization and deserialization.
- It has four fields. Two of these fields (`customer` and `business`) are non-transient (the value of these fields can be saved to the byte stream during serialization) and the other two fields (`contractID` and `passKeys`) are transient (the value of these fields cannot be saved to the byte stream during serialization).
- It has `toString` method that is overridden and prints the value of these four fields.

SerializationDemo.java

```
package com.java;

import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class SerializationDemo {

 public static Object deSerialization(String file) throws IOException,
ClassNotFoundException {
  FileInputStream fileInputStream = new FileInputStream(file);
  BufferedInputStream bufferedInputStream = new
BufferedInputStream(fileInputStream);
  ObjectInputStream objectInputStream = new ObjectInputStream(bufferedInputStream);
  Object object = objectInputStream.readObject();
  objectInputStream.close();
  return object;
 }


 public static void serialization(String file, Object object) throws IOException {
  FileOutputStream fileOutputStream = new FileOutputStream(file);
  BufferedOutputStream bufferedOutputStream = new
BufferedOutputStream(fileOutputStream);
  ObjectOutputStream objectOutputStream = new
ObjectOutputStream(bufferedOutputStream);
  objectOutputStream.writeObject(object);
  objectOutputStream.close();
 }
}
```

Explanation

- This class has two methods, first for deserialization, and the other is for serialization.
- The `deSerialization` method accepts the file path and name which has the byte stream of an object's state and returns the Object that can be down casted to the corresponding serialized class object.
- In this `deSerialization` method; first we open the file in read mode using `FileInputStream` class then we pass the object of this file to `BufferedInputStream` class constructor while instantiating it, which speeds up the reading operation. The object of `BufferedInputStream` class is passed to the constructor of class `ObjectInputStream` while instantiating this class. We can call the `readObject ()` method of this class which will carry out the actual deserialization.
- Lastly, close the `ObjectInputStream` object and return the deserialized data value object.
- The `serialization` method accepts two parameters; i.e. the file path & name, and the data value object that is required to be serialized.

- In this `serialization` method first we open the file in write mode using `FileOutputStream` class then we pass the object of this file to `BufferedOutputStream` class constructor while instantiating it, which speeds up the writing operation. The object of `BufferedOutputStream` class is passed to the constructor of class `ObjectOutputStream` while instantiating this class. We can call the `writeObject (object)` method of this class to serialize the data value object and write the byte stream into the opened file.
- Lastly close the `ObjectOutputStream` object.

SerializationImplementation.java

```
package com.java;

import java.io.IOException;

public class SerializationImplementation {

 public static void main(String args[]) {
  DataValueObject dataValueObject = new DataValueObject();
  dataValueObject.setCustomer("Debbie");
  dataValueObject.setBusiness("JAVA Concepts");
  dataValueObject.setContractID("ZZZZZZ");
  dataValueObject.setPassKeys("!@wer#$");

  try {
   SerializationDemo.serialization("fileToSave.txt", dataValueObject);
   DataValueObject object = (DataValueObject)
SerializationDemo.deSerialization("fileToSave.txt");
   System.out.println(object.toString());
  } catch (IOException exp) {
   exp.printStackTrace();
  } catch (ClassNotFoundException exp) {
   exp.printStackTrace();
  }
 }

}
```

Explanation

In this class, we are testing our serialization and deserialization procedure in the following four steps.

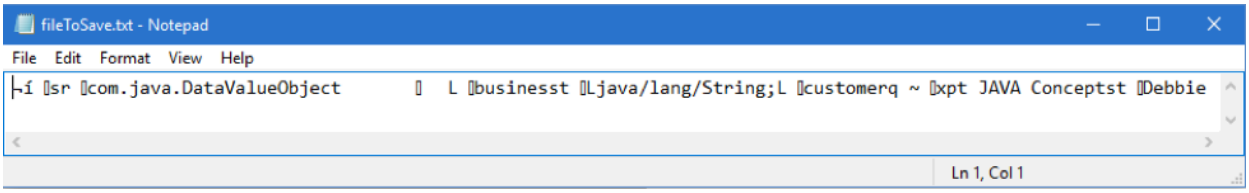
First, we are populating all four fields of the data value object.

Second, we are calling serialization method after passing the file path and name as the first parameter `fileToSave.txt` ; and data value object as the second parameter. This method will write the state of data value object to the byte stream into the file `fileToSave.txt` .

Third, we are calling deserialization method after passing the file path and name as the first parameter `fileToSave.txt` . This will read the stored object's state and return the deserialized object which is down casted to the data value object class.

Lastly, we are printing the `toString ()` method of the data value object class which was overridden.

`fileToSave.txt` FILE content after serialization as byte stream



Inference

As printed on the console below, we can observe that our serialization and deserialization methods in JAVA program were able to serialize and deserialize the data value object, respectively. We can observe that the values of the fields `contractID` and `passKeys` are null (default string value) and were declared as transient, and therefore, were not stored in the file while saving the objects' state into the byte stream.