Serialization

1. Introduction
   1. Serialization
   2. De-Serialization
   3. Transient key word
   4. Static vs transient
   5. Final vs transient
2. Object Graph in Serialization
3. Customized Serialization
4. Serialization with Respect to Inheritance
5. Externalization
6. serialVersionUID

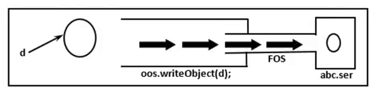
SERIALIZATION

The process of writing state of an object to a file is called Serialization. But strictly speaking it is the process of converting an **object from Java supported form to either file supported form** or Network supported form.

By using FileOutputStream (FOO) and ObjectOutputStream (OOS) classes we can achieve serialization.

By using FileOutputStream we can write binary data but not object directly. ObjectOutputStream we can take object directly. By using OOS’s writeObject method to pass your object.

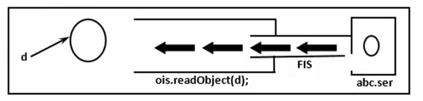
ObjectOutputStream converts object to binary data. FileOutputStream convert binary data to file. Reverse to de-serialization.



DE-SERIALIZATION

The process of reading state of an object from a file is de-serialization. But strictly specking it is the process of converting **an object from either file or network supported form into Java supported** form.

By using FileInputStream and ObjectInputStream classes we can achieve.



SERIALIZATION EXAMPLE

**public** **class** SerializationDemo {

**public** **static** **void** main(String[] args) **throws** FileNotFoundException, IOException, ClassNotFoundException {

Dog dog = **new** Dog("Pup");

System.*out*.println("Object to write ->->"+dog);

ObjectOutputStream objectOutputStream = **new** ObjectOutputStream(**new** FileOutputStream("writeDog.txt"));

objectOutputStream.writeObject(dog);

Dog dog1 = **null**;

ObjectInputStream objectInputStream = **new** ObjectInputStream(**new** FileInputStream("writeDog.txt"));

dog1 = (Dog) objectInputStream.readObject();

System.*out*.println("Object red ->->"+dog1);

}

}

**class** Dog **implements** Serializable{

String name;

Dog(String nName) {

**this**.name = nName;

}

**private** String getName() {

**return** **this**.name;

}

**public** String toString() {

**return** "My dog name is ->-> "+getName();

}

}

TRANSIENT KEYWORD

* It is a keyword applicable only for variable
* At the time of Serialization if we don’t want to save to value of a particular variable to meet security constraints then we should go for transient keyword
* At the time of serialization JVM ignore original value of transient variables and save default value to the file

STATIC VS TRASNIENT

* Static variables are not part of object state and hence they won’t participate in serialization
* Due to this Þclaring a static variable as transient there is no use

FINAL VS TRANSIENT

* Final variable will be participated in serialization directly by their values
* Due to this declaring a final variable as transient there is no use or impact.

SERIALIZATION

* We can serialize any number of objects to the file, but in which order we serialized in the same order only we have to DE serialize. In serialization the order of objects is important.

OBJECT GRAPH IN SERIALIZATION

* Whenever we serialize an object the set of all objects which are reachable from that object will be serialized automatically. This group of objects is nothing but object graph.
* In object graph every object should be serializable. If a least one object is non-serializable then we will get Runtime Exception saying NotSerializableException.

CUSTOMIZED SERIALIZATION

We can implement Customized serialization by using the following 2 methods

* Private void writeObject(ObjectOutputStream os) throws Exception
  + This method will be executed automatically at the time of serialization
  + Hence while performing serialization if we want to do any extra work we have to write corresponding code in this method only
* Private void readObject(ObjectInputStream is) throws Exception
  + This is will be executed automatically at the time of deserialization
  + Hence while performing de-serialization if we want to do any extra work we have to define the corresponding code in this method only

Note:

1. The above methods are callback methods because these methods will be executed automatically by JVM
2. While performing which object serialization we have to do extra work, in the corresponding class we have to define above methods

**class** Account **implements** Serializable{

String userName = "Bob";

**transient** String passWord = "admin123";

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

os.defaultWriteObject();

String epwd = "123"+passWord;

os.writeObject(epwd);

}

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

is.defaultReadObject();

String epwd = (String)is.readObject();

passWord = epwd.substring(3);

}

}

SERIALIZATION WITH RESPECT TO INHERITANCE

Case 1: If the parent is serializable then be default every child serializable. That is serializable nature is inheriting from parent to child.

Case 2: Even though parent class doesn’t implement serializable interface we can serialize child class object.

At the time of serialization JVM check is any instance variable is inheriting form non-serialzable parent or not. If any variable inheriting from non-serialzable parent then JVM ignores original value and save default value to the file.

**class** Animal {

**int** i = 10;

Animal() {

System.*out*.println("Animal constructor called");

}

}

**class** Doggy **extends** Animal **implements** Serializable{

**int** j = 20;

Doggy() {

System.*out*.println("Dog constructor called");

}

}

EXTERNALIZATION

In serialization everything takes care by JVM and programmer doesn’t have any control. In serialization total object will be saved always to the file and it is not possible to save part of the object which creates performance problems. To overcome these problems we should go for externalization where everything takes care by programmer and JVM doesn’t have any control.

The main advantage of externalization over serialization is based on our requirement we can save either total object or part of the object so that relatively performance will be improved.

To provide externalizable ability for any Java object compulsory the corresponding class should implements externalizable interface.

Externalizable interface present in java.io.Package and it is the child interface of serializable.

Externalizable interface contains 2 methods.

* writeExternal()
* readExternal()

1. public void writeExternal(ObjectOutput out) throws IOException
   1. This method will be executed automatically at the time of serialization
   2. Within this method we have to write code to save required variables to the file.
2. Public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException
   1. This method will be executed automatically at the time of de-serialzation.
   2. Within this method we have to write code to read required variables from the File and assign to the current object.
   3. **Strictly speaking at the time of deserialization JVM will create a separate new object by executing public no-argument constructor.** On that object readExternal() method will be executed.
   4. Hence every externalizable interface implemented class should compulsory contain public no-argument constructor. Otherwise we will get runtime exception saying InvalidClassException.

**public** **class** ExternalizableDemo **implements** Externalizable{

String s;

**int** i;

**int** j;

**public** ExternalizableDemo() {

System.*out*.println("Public no-Arg constructor");

}

ExternalizableDemo(String s, **int** i, **int** j) {

**this**.s = s;

**this**.i = i;

**this**.j = j;

}

**public** **void** readExternal(ObjectInput in) **throws** IOException,

ClassNotFoundException {

s = (String) in.readObject();

i = in.readInt();

}

**public** **void** writeExternal(ObjectOutput out) **throws** IOException {

out.writeObject(s);

out.writeInt(i);

}

**public** **static** **void** main(String[] args) **throws** FileNotFoundException, IOException, ClassNotFoundException {

ExternalizableDemo t1 = **new** ExternalizableDemo("Durga", 10, 20);

ObjectOutputStream oos = **new** ObjectOutputStream(**new** FileOutputStream("external.txt"));

oos.writeObject(t1);

ObjectInputStream ois = **new** ObjectInputStream(**new** FileInputStream("external.txt"));

ExternalizableDemo t2 = (ExternalizableDemo) ois.readObject();

System.*out*.println(t2.s+ " ... "+ t2.i+ " ... "+t2.j);

}

}

SERIALVERSIONUID

In serialization both sender and receiver need not be same and need not be from the same location and need not to use same machine. Person may be different location may be different and machines may be different.

At the time of serialization JVM will save a unique id with every object. This unique id will be generated by JVM based .class file. At the time of deserialization receiver side JVM will compare object unique id with local .class unique id. If both are matched then only deserialization will be performed otherwise receiver unable to deserialize and we will get Runtime Exception saying InvalidClassException

This unique identifier is nothing but serialVersionUID.

THE PROBLEMS OF DEPENDING ON DEFAULT SERIALVERSIONUID GENERATED BY JVM are:

Both sender and receiver should use same JVM w.r.t to vender and version. If there is any incompatibility in JVM versions, the receiver is unable to deserialize because of different serialversionUID’s. In this case receiver will get RuntimeException saying invalidClassException.

After serialization if we change .class file at receiver side then we can’t perform deserialization because of miss match in serialVersionUID’s of local class of receiver and serialized object.

To generate serialVersionUID internally JVM will use complex Algorithns which may create performance problems.

We can solve above problem by configuring our own serialVersionUID/

Private static final long serialVersionUID = 1L;