Singleton Design pattern with Serialization

November 11, 2016 SJ Serialization 0

In this article, we will discuss singleton design pattern with respect to serialization in detail

Let me tell you the scenario I have faced during one of the Java interview for leading investment banker in the market, few years back

What are the things that need to be taken care for making a class as singleton?

- 1st thing make constructor as private such that no one outside the class can create an instance
- 2nd provide public method to return same instance every time

That's fine, **what if I serialize this singleton class and then de-serialize, won't it create new instance?**

Exactly, we are going to discuss above scenario i.e.; how to stop creating a new instance during de-serialization

Before discussing that, we will make our-self *clear few doubts that may arise* (at least I had after giving the interview)

How to check that instance before serialization and instance restored after de-serialization are same or different?

We can check using *hashcode* of both the instances

Let us dive-deep and discuss all above things programmatically

Case 1: Hash codes of both instances are different

A simple POJO class called Customer implementing *java.io.Serializable* interface to mark that this class got special ability (i.e.; it can be serialized and de-serialized)

- Consists of private constructor (so that no outside class can construct a new object)
- A public method to return same instance every time (eagerly initialized)
- **Note**: we can also initialize customer lazily, by null checking and initializing afterwards

Customer.java

```
package
in.bench.resources.singleton.serialization;
```

Java JDBC: An example to connect MS Access database
Spring JDBC: An example on JdbcTemplate using Annotation
Oracle OSB 12c: Service Callout and Routing Table example
Oracle OSB 12c: Hello World service with both Business and Proxy Service

```
import java.io.Serializable;

class Customer implements Serializable {

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

        // to always, return same instance
        private volatile static Customer
CUSTOMER = new Customer();

        // private constructor
        private Customer() {
                // private constructor
        }

        // create static method to get same
instance every time
        public static Customer getInstance(){
                return CUSTOMER;
        }

        // other methods and details of this
class
}
```

Test class where both serialization and de-serialization happens in the same class

CustomerSerializeDeSerializeDemo.java

```
package
in.bench.resources.singleton.serialization;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
```

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class CustomerSerializeDeSerializeDemo {

	public static void main(String[] args) {

		// create an customer object
using 3-arg parametrized constructor
		Customer serializeCustomer =
Customer.getInstance();

		// creating output stream
variables
		FileOutputStream fos = null;
		ObjectOutputStream oos = null;

		// creating input stream
variables
		FileInputStream fis = null;
		ObjectInputStream ois = null;

		// creating customer object
reference
		// to hold values after de-
serialization
		Customer deSerializeCustomer =
null;

		try {
			// for writing or saving
binary data
			fos = new
FileOutputStream("Customer.ser");

			// converting java-
object to binary-format
			oos = new
ObjectOutputStream(fos);

			// writing or saving
customer object's value to stream
```

```
oos.writeObject(serializeCustomer);
                        oos.flush();
                        oos.close();


System.out.println("Serialization: "
                                            +
"Customer object saved to Customer.ser file\n");

                        // reading binary data
                        fis = new
FileInputStream("Customer.ser");

                        // converting binary-
data to java-object
                        ois = new
ObjectInputStream(fis);

                        // reading object's
value and casting to Customer class
                        deSerializeCustomer =
(Customer) ois.readObject();
                        ois.close();

                        System.out.println("De-
Serialization: Customer object "
                                            + "de-
serialized from Customer.ser file\n");
                }
                catch (FileNotFoundException
fnfex) {
                        fnfex.printStackTrace();
                }
                catch (IOException ioex) {
                        ioex.printStackTrace();
                }
                catch (ClassNotFoundException
ccex) {
                        ccex.printStackTrace();
                }

                // printing hash code of
serialize customer object
```

```
                System.out.println("Hash code of
the serialized "
                                   + "Customer
object is " + serializeCustomer.hashCode());

                // printing hash code of de-
serialize customer object
                System.out.println("\nHash code
of the de-serialized "
                                   + "Customer
object is " + deSerializeCustomer.hashCode());
        }
}
```

Output:

```
Serialization: Customer object saved to
Customer.ser file

De-Serialization: Customer object de-serialized
from Customer.ser file

Hash code of the serialized Customer object is
26253138

Hash code of the de-serialized Customer object
is 33121026
```

Explanation:

- From above output, it is clear that hahscode of both instances are different which they are 2 different objects
- Hence, making Customer class as singleton design pattern fails
- Although, for every serialization the hash code remain same (until and unless if we change any class details
- But with every de-serialization, hash code of Customer class might change

To suppress this behavior and make Customer class as singleton design pattern, we have provide or override one more method, which we are going to see in the next case

Case 2: Hash codes of both instances are same by implementing readReolve() method

A simple POJO class called Customer implementing *java.io.Serializable* interface to mark that this class got special ability (i.e.; it can be serialized and de-serialized)

- Consists of private constructor (so that no outside class can construct a new object)
- A public method to return same instance every time (eagerly initialized)
- **Note**: we can also initialize customer lazily, by null checking and initializing afterwards
- Finally, it contains ***readResolve()*** method to suppress to create new instance or say returns the same singleton instance every time during de-serialization

Customer.java

```
package
in.bench.resources.singleton.serialization;

import java.io.ObjectStreamException;
import java.io.Serializable;

class Customer implements Serializable {

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

        // to always, return same instance
        private volatile static Customer
CUSTOMER = new Customer();
```

```
        // private constructor
        private Customer() {
                // private constructor
        }

        // create static method to get same
instance every time
        public static Customer getInstance(){
                return CUSTOMER;
        }

        // readResolve method
        private Object readResolve() throws
ObjectStreamException {
                return CUSTOMER;
        }

        // other methods and details of this
class
}
```

Test class where both serialization and de-serialization happens in the same class

CustomerSerializeDeSerializeDemo.java

```
package
in.bench.resources.singleton.serialization;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class CustomerSerializeDeSerializeDemo {

        public static void main(String[] args) {
```

```
                // create an customer object
                Customer serializeCustomer =
Customer.getInstance();

                // creating output stream
variables
                FileOutputStream fos = null;
                ObjectOutputStream oos = null;

                // creating input stream
variables
                FileInputStream fis = null;
                ObjectInputStream ois = null;

                // creating customer object
reference
                // to hold values after de-
serialization
                Customer deSerializeCustomer =
null;

                try {
                        // for writing or saving
binary data
                        fos = new
FileOutputStream("Customer.ser");

                        // converting java-
object to binary-format
                        oos = new
ObjectOutputStream(fos);

                        // writing or saving
customer object's value to stream

oos.writeObject(serializeCustomer);
                        oos.flush();
                        oos.close();


System.out.println("Serialization: "
                                        +
"Customer object saved to Customer.ser file\n");
```

```
                        // reading binary data
                        fis = new
FileInputStream("Customer.ser");

                        // converting binary-
data to java-object
                        ois = new
ObjectInputStream(fis);

                        // reading object's
value and casting to Customer class
                        deSerializeCustomer =
(Customer) ois.readObject();
                        ois.close();

                        System.out.println("De-
Serialization: Customer object "
                                           + "de-
serialized from Customer.ser file\n");
                }
                catch (FileNotFoundException
fnfex) {
                        fnfex.printStackTrace();
                }
                catch (IOException ioex) {
                        ioex.printStackTrace();
                }
                catch (ClassNotFoundException
ccex) {
                        ccex.printStackTrace();
                }

                // printing hash code of
serialize customer object
                System.out.println("Hash code of
the serialized "
                                   + "Customer
object is " + serializeCustomer.hashCode());

                // printing hash code of de-
serialize customer object
                System.out.println("\nHash code
of the de-serialized "
```

```
                                    + "Customer
object is " + deSerializeCustomer.hashCode());
        }
}
```

Output:

```
Serialization: Customer object saved to
Customer.ser file

De-Serialization: Customer object de-serialized
from Customer.ser file

Hash code of the serialized Customer object is
26253138

Hash code of the de-serialized Customer object
is 26253138
```

Explanation:

- From above output, it is clear that hash code of both instances (before & after serialization) are same
- If we de-serialize again one more time, even then we will get same hash code for both instances

References:

https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html
https://docs.oracle.com/javase/7/docs/platform/serialization/spec/serial-arch.html
https://docs.oracle.com/javase/7/docs/api/java/io/ObjectOutputStream.html
https://docs.oracle.com/javase/7/docs/api/java/io/ObjectInputStream.html
https://docs.oracle.com/javase/7/docs/api/java/io/FileOutputS

tream.html
https://docs.oracle.com/javase/7/docs/api/java/io/FileInputStream.html
http://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html#jls-8.3.1.3
https://docs.oracle.com/javase/7/docs/platform/serialization/spec/input.html#5903

Read Also:

- Java Serialization and De-Serialization Tutorial Index
- Serialization and De-Serialization in Java
- Serializable interface
- Transient keyword with Serialization in Java
- Transient keyword with static variable in Serialization
- Transient keyword with final variable in Serialization
- Serializing a variable with transient modifier or keyword
- Order of Serialization and De-Serialization
- Serialization with Aggregation
- Serialization with Inheritance
- Externalizable interface with example
- Serializable v/s Externalizable
- Importance of SerialVersionUID in Serialization
- How to stop Serialization in Java
- How to construct a singleton class in a multi-threaded environment in Java
- How to serialize and de-serialize ArrayList in Java

Happy Coding !!
Happy Learning !!

« How to stop Serialization in Java

Importance of SerialVersionUID in Serialization »

Related Posts:

1. **Importance of SerialVersionUID in Serialization**
2. **How to stop Serialization in Java**
3. **Serialization interview question and answer in Java**

4. **Serialization and De-Serialization in Java**

DESERIALIZATION JAVA SERIALIZABLE SERIALIZATION SERIALVERSIONUID