Introduction to Java Serialization

1. Introduction

Serialization is the conversion of the state of an object into a byte stream; deserialization does the opposite. Stated differently, serialization is the conversion of a Java object into a static stream (sequence) of bytes, which we can then save to a database or transfer over a network.

2. Serialization and Deserialization

The serialization process is instance-independent; for example, we can serialize objects on one platform and deserialize them on another. **Classes that are eligible for serialization need to implement a special marker interface,** *Serializable.*

Both *ObjectInputStream* and *ObjectOutputStream* are high level classes that extend *java.io.InputStream* and *java.io.OutputStream,* respectively. *ObjectOutputStream* can write primitive types and graphs of objects to an *OutputStream* as a stream of bytes. We can then read these streams using *ObjectInputStream*.

The most important method in *ObjectOutputStream* is:

```
public final void writeObject(Object o) throws IOException;Copy
```

This method takes a serializable object and converts it into a sequence (stream) of bytes. Similarly, the most important method in *ObjectInputStream* is:

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

This method can read a stream of bytes and convert it back into a Java object. It can then be cast back to the original object.

Let’s illustrate serialization with a *Person* class. Note that **static fields belong to a class (as opposed to an object) and are not serialized**. Also, note that we can use the keyword *transient* to ignore class fields during serialization:

```
public class Person implements Serializable {
    private static final long serialVersionUID = 1L;
    static String country = "ITALY";
    private int age;
    private String name;
    transient int height;

    // getters and setters
}
```

The test below shows an example of saving an object of type *Person* to a local file, and then reading the value back in:

```
@Test
public void whenSerializingAndDeserializing_ThenObjectIsTheSame() ()
  throws IOException, ClassNotFoundException {
    Person person = new Person();
    person.setAge(20);
    person.setName("Joe");

    FileOutputStream fileOutputStream
      = new FileOutputStream("yourfile.txt");
    ObjectOutputStream objectOutputStream
      = new ObjectOutputStream(fileOutputStream);
    objectOutputStream.writeObject(person);
    objectOutputStream.flush();
    objectOutputStream.close();

    FileInputStream fileInputStream
      = new FileInputStream("yourfile.txt");
    ObjectInputStream objectInputStream
      = new ObjectInputStream(fileInputStream);
    Person p2 = (Person) objectInputStream.readObject();
    objectInputStream.close();

    assertTrue(p2.getAge() == person.getAge());
    assertTrue(p2.getName().equals(person.getName()));
}
```

We used *ObjectOutputStream* for saving the state of this object to a file using *FileOutputStream*. The file *"yourfile.txt"* is created in the project directory. This file is then loaded using *FileInputStream. ObjectInputStream* picks this stream up and converts it into a new object called *p2*.

Finally, we'll test the state of the loaded object, and ensure it matches the state of the original object.

Note that we have to explicitly cast the loaded object to a *Person* type.

3. Java Serialization Caveats

There are some caveats which concern serialization in Java.

3.1. Inheritance and Composition

When a class implements the *java.io.Serializable* interface, all its sub-classes are serializable as well. Conversely, when an object has a reference to another object, these objects must implement the *Serializable* interface separately, or else a *NotSerializableException* will be thrown:

```
public class Person implements Serializable {
    private int age;
    private String name;
    private Address country; // must be serializable too
}
```

If one of the fields in a serializable object consists of an array of objects, then all of these objects must be serializable as well, or else a *NotSerializableException* will be thrown.

3.2. Serial Version UID

The JVM associates a version (*long*) number with each serializable class. We use it to verify that the saved and loaded objects have the same attributes, and thus are compatible on serialization.

Most IDEs can generate this number automatically, and it's based on the class name, attributes, and associated access modifiers. Any changes result in a different number, and can cause an *InvalidClassException*.

If a serializable class doesn't declare a *serialVersionUID*, the JVM will generate one automatically at run-time. However, it's highly recommended that each class declares its *serialVersionUID*, as the generated one is compiler dependent and thus may result in unexpected *InvalidClassExceptions*.

3.3. Custom Serialization in Java

Java specifies a default way to serialize objects, but Java classes can override this default behavior. Custom serialization can be particularly useful when trying to serialize an object that has some unserializable attributes. We can do this by providing two methods inside the class that we want to serialize:

```
private void writeObject(ObjectOutputStream out) throws IOException;
```
Copy

and

```
private void readObject(ObjectInputStream in) 
  throws IOException, ClassNotFoundException;
```
Copy

With these methods, we can serialize the unserializable attributes into other forms that we can serialize:

```
public class Employee implements Serializable {
    private static final long serialVersionUID = 1L;
    private transient Address address;
    private Person person;

    // setters and getters

    private void writeObject(ObjectOutputStream oos) 
      throws IOException {
        oos.defaultWriteObject();
        oos.writeObject(address.getHouseNumber());
    }

    private void readObject(ObjectInputStream ois) 
      throws ClassNotFoundException, IOException {
        ois.defaultReadObject();
        Integer houseNumber = (Integer) ois.readObject();
        Address a = new Address();
        a.setHouseNumber(houseNumber);
        this.setAddress(a);
    }
}
public class Address {
    private int houseNumber;

    // setters and getters
}
```

We can run the following unit test to test this custom serialization:

```
public void whenCustomSerializingAndDeserializing_ThenObjectIsTheSame() 
  throws IOException, ClassNotFoundException {
    Person p = new Person();
    p.setAge(20);
    p.setName("Joe");

    Address a = new Address();
    a.setHouseNumber(1);

    Employee e = new Employee();
    e.setPerson(p);
    e.setAddress(a);

    FileOutputStream fileOutputStream
      = new FileOutputStream("yourfile2.txt");
    ObjectOutputStream objectOutputStream 
      = new ObjectOutputStream(fileOutputStream);
    objectOutputStream.writeObject(e);
    objectOutputStream.flush();
    objectOutputStream.close();

    FileInputStream fileInputStream 
      = new FileInputStream("yourfile2.txt");
    ObjectInputStream objectInputStream
```

```
      = new ObjectInputStream(fileInputStream);
    Employee e2 = (Employee) objectInputStream.readObject();
    objectInputStream.close();

    assertTrue(
      e2.getPerson().getAge() == e.getPerson().getAge());
    assertTrue(
      e2.getAddress().getHouseNumber() == e.getAddress().getHouseNumber());
}
```

In this code, we can see how to save some unserializable attributes by serializing *Address* with custom serialization. Note that we must mark the unserializable attributes as *transient* to avoid the *NotSerializableException.*