CS502052: Enterprise Systems Development Concepts

Lab 2: Java Serialization

I. Introduction

In this second lab, we will demonstrate on **Java Serialization**, which represents an object as a sequence of bytes.

II. Java Serialization

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

The two classes, ***ObjectInputStream*** and ***ObjectOutputStream***, are high-level streams that contain the methods for *serializing* and *deserializing* an object.

1. ObjectOutputStream

The ***ObjectOutputStream*** class contains many write methods for writing various data types, but one method stands out:

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

The above method *serializes* an Object and sends it to the output stream.

2. ObjectInputStream

The ***ObjectInputStream*** class contains the following method for deserializing an object:

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

This method retrieves the next ***Object*** out of the stream and *deserializes* it. The return value is ***Object***, so you will need to cast it to its appropriate data type.

3. Example

To demonstrate how serialization works in Java, we are going to use the *Employee* class. Suppose that we have the following *Employee* class, which implements the *Serializable* interface:

```
public class Employee implements Serializable {

    private static final long serialVersionUID = 1L;
    private String name;
    private String address;
    private transient int SSN;

    public Employee(String name, String address, int SSN)
    {
        this.name = name;
        this.address = address;
        this.SSN = SSN;
    }

    public void mailCheck()
    {
        System.out.println("Mailing a check to " + this.name
                + ", at " + this.address);
    }

    public void printInfo()
    {
    public String getName()
    {
    public String getAddress()
    {
    public int getSSN()
    {
}
```

Figure 1 Employee class

Notice that for a class to be serialized successfully, **2** conditions must be met:

- The class must implement the ***java.io.Serializable*** interface.
- All the fields in the class must be serializable. If a field is not serializable, it must be marked transient.
- To avoid *"local class incompatible"* error when you make some changes to class after serializing, then deserializing, you should put: `private static final long serialVersionUID = 1L;`.

If you are curious to know if a Java Standard Class is serializable or not, check the documentation for the class. The test is simple: If the class implements ***java.io.Serializable***, then it is serializable; otherwise, it's not.

Serializing an Object

The ***ObjectOutputStream*** class is used to serialize an ***Object***. The following program instantiates an ***Employee*** object and serializes it to a file.

When the program is done executing, a file named ***employee.ser*** is created. The program does not generate any output, but study the code and try to determine what the program is doing.

Note: When serializing an object to a file, the standard convention in Java is to give the file a ***.ser*** extension.

```
public class Test {

    public static void main(String[] args)
    {
        Employee e = new Employee("John", "TDTU", 111);

        try {
            FileOutputStream file = new FileOutputStream("data/employee.ser");
            ObjectOutputStream out = new ObjectOutputStream(file);

            out.writeObject(e);

            out.close();
            file.close();

            System.out.println("Serialized data is saved in data/employee.ser");

        } catch(IOException i) {
            System.err.println(i.getMessage());;
        }

    }
}
```

Figure 2 SerializeDemo class

Deserializing an Object

The following ***DeserializeDemo*** program *deserializes* the Employee object created in the ***SerializeDemo*** program. Study the program and try to determine its output.

```
public class DeserializeDemo {

    public static void main(String[] args)
    {
        Employee e = null;

        try
        {
            FileInputStream file = new FileInputStream("data/employee.ser");
            ObjectInputStream in = new ObjectInputStream(file);

            e = (Employee) in.readObject();

            in.close();
            file.close();

            System.out.println("Deserialized data completely.");
        } catch(IOException i)
        {
            System.err.println(i.getMessage());
        } catch (ClassNotFoundException ex)
        {
            System.err.println(ex.getMessage());
        }

        System.out.println("Deserialized Employee...");
        System.out.println("Name: " + e.getName());
        System.out.println("Address: " + e.getAddress());
        System.out.println("SSN: " + e.getSSN());
    }

}
```

Figure 3 DeserializeDemo class

This will produce the following result:

```
run:
Deserialized data completely.
Deserialized Employee...
Name: John
Address: TDTU
SSN: 0
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 4 DeserializeDemo output

Here are following important points to be noted:

- The ***try/catch*** block tries to catch a ***ClassNotFoundException***, which is declared by the ***readObject()*** method. For a JVM to be able to deserialize an object, it must be able to find the bytecode for the class. If the JVM can't find a class during the deserialization of an object, it throws a ***ClassNotFoundException***.
- Notice that the return value of ***readObject()*** is cast to an ***Employee*** reference.
- The value of the ***SSN*** field was *"111"* when the object was serialized, but because the field is ***transient***, this value was not sent to the output stream. The ***SSN*** field of the deserialized ***Employee*** object is ***0***.

III. Exercises

1. The above sample program, we only work on one instance of *Employee* class. Now, suppose we have multiple instances of *Employee*, store in *ArrayList*. You re-write the demo program to work with multiple instances. Assuming there are three instances:

```
ArrayList<Employee> arr = new ArrayList<>();
arr.add(new Employee("Kelvin", "TDTU", 1));
arr.add(new Employee("Harry", "TDTU", 2));
arr.add(new Employee("Jeremy", "TDTU", 3));
```

2. Suppose you have an interface of *Instrument* that has a *play()* method. You then program three classes to implement this interface, i.e., *Guitar*, *Piano*, *Trumpet*. Let's demonstrate the Java Serialization in this situation.
3. Applying Java Serialization in this tutorial to serialize and deserialize a picture. That means, given a picture (in any format), serialize the picture from one computer and then deserialize this in another one.