THE UNIVERSITY OF WESTERN ONTARIO

Computer Science 2212b - Winter 2014
Introduction to Software Engineering

Lab 7 - Data Persistence with Java Object Serialization

Part I
Introduction to Java Object Serialization

Estimated Time: 60-120 minutes (depending on your reading speed)

1 Overview

In this lab, you will read an excerpt from the Wrox Press book, *Professional Java JDK 6 Edition*, by Richardson et al.

You will then take some code that has been provided to you, and modify it so that it can persist data to a file, and load that data back from the file.

2 Prerequisites

1. Download and read the file `professional_jdk_6_chapter-5.pdf` from the course web site. It is being provided under the Fair Dealing Exception and/or Education Exception to the Canadian Copyright Act.
   - You **do not** have to read the entire chapter. It is being provided in case you are interested in more advanced, flexible methods of object serialization.
   - You should read pages 1-10 of the PDF.
   - You can skim pages 11-20, but try to follow the examples they're showing as the Swing example, in particular, is pertinent to your project.
   - You should read pages 21-26 (pay careful attention to the discussion of versioning).
   - Whether or not you read the rest is up to you.

3 Getting Started

1. Change to your individual Git repository, create a `lab7` directory, and change to it:

```
$ cd ~/courses/cs2212/labs
$ mkdir lab7
$ cd lab7
```

2. Download the `pom.xml` file from the following Gist: https://gist.github.com/jsuwo/8951417
3. Copy the `pom.xml` file to your `lab7` directory.
4. Customize the `groupId`, `artifactId`, and `mainClass` elements appropriately.
5. Observe that the `maven-compiler-plugin` is being used to specify that the compilation process should output code targeted to JDK 7. This is because the code uses a feature that is new to JDK 7 (`App.java` uses a `String` object as the expression of its `switch` statement).
   - Note that, for this to work, I had to set the `JAVA_HOME` instance variable properly to point to a JDK 7 installation. Without doing so, Maven was looking at an older version of the JDK on my system, and I was getting errors when attempting to run `mvn package`.

6. Create the directory structure to host our project:

```
$ mkdir -p src/{main,test}/{java,resources}
```

7. Create the directory structure to house your Java package:

```
$ mkdir -p src/main/java/ca/uwo/csd/cs2212/USERNAME
```

   Again, `USERNAME` should be your UWO username in **lowercase**.

8. Download `App.java`, `Person.java`, and `Address.java` from the aforementioned Gist. Copy them to your `src/main/java/ca/uwo/csd/cs2212/USERNAME` directory.

9. Customize the `package` statement in each of these files appropriately.

4 Serializing Data

In this section, you'll modify `App.java` and fill in the details of the `storePerson` method to serialize a `Person` object to disk.

The `main` method has already been completed for you. A user running the program can pass the following arguments:

```
$ java -jar target/jshantz4-lab7-1.0-SNAPSHOT.jar store person.dat Jeff "123 Penny Lane" London England
    555-555-555 123-123-123 ...
```

If the first argument to the program is `store`, then the program expects that the next arguments will be as follows:

- `filename`
- `name`
- `street`
- `city`
- `country`
- `phoneNumber1`
- `phoneNumber2`
- ...

Note that there may be only one phone number specified, or there may be multiple phone numbers – as many as the user likes.

The `storePerson` method takes a `String[] args` array containing all the parameters passed to the program. You can assume that there is a correct number of parameters passed – you do not have to perform any validation on the arguments.

1. Fill in the details of the `storePerson` method. It should:
   - Create a new `Person` object, initialized with all the details passed to the program.
   - Serialize the object to the file specified.
2. Using the knowledge that you gained from the reading in Section 2, modify the `Person` and `Address` classes so that they can be serialized.
   - Do **not** worry about versioning at this point.

3. Package your application as a JAR and run it. Ensure that it generates an output file. Take a look at the output file.

```
$ mvn package
$ java -jar target/jshantz4-lab7-1.0-SNAPSHOT.jar store person.dat Jeff "123 Penny Lane" London England
    555-555-555 123-123-123
$ cat person.dat
```

Of course, the output file is in a binary format, so there's not much to see, but you may see a few strings in there.

4. Run the `strings` command on the output file:

```
$ strings person.dat
```

Notice that it prints all of your data. Do **not** be fooled by binary formats like this. They are **not** encrypted, and therefore not secure. If the data to be stored was sensitive, then we would need to encrypt the contents of the file stream being written to disk. This is beyond the scope of this lab, but is mentioned so that the reader is not fooled into a false sense of security provided by a binary format.

5 Deserializing Data

In this section, you'll modify `App.java` and fill in the details of the `loadPerson` method to deserialize a `Person` object from disk.

A user running the program can pass the following arguments:

```
$ java -jar target/jshantz4-lab7-1.0-SNAPSHOT.jar load person.dat
```

If the first argument to the program is `load`, then the program expects that the next argument will be the filename from which the object is to be loaded.

The `loadPerson` method takes a `String[] args` array containing all the parameters passed to the program. You can assume that there is a correct number of parameters passed and that the file specified exists – you do not have to perform any validation on the arguments.

1. Fill in the details of the `loadPerson` method. It should:
   - Deserialize the `Person` object from the specified file
   - Print the result of calling the `toString` method on the `Person` object.
2. Package your application as a JAR and run it. Ensure that it prints the correct data.

```
$ mvn package
$ java -jar target/jshantz4-lab7-1.0-SNAPSHOT.jar load person.dat
```

6 Versioning

1. Edit `Person.java` and add the following line to the list of instance variables in the class:

```
private String gender;
```

2. Package your application as a JAR and try to load the previously serialized object:

```
$ mvn package
$ java -jar target/jshantz4-lab7-1.0-SNAPSHOT.jar load person.dat
```

Notice that an `InvalidClassException` is thrown. This is because the version of the serialized object does not match the version of the class – we *changed* the `Person` class.

3. Edit `Person.java` and remove the `gender` instance variable.

4. Add the following line to `Person.java`:

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

   This explicitly specifies the version number of the class.

5. Package your application as a JAR, delete the serialized object, and try storing data again:

```
$ mvn package
$ rm person.dat
$ java -jar target/jshantz4-lab7-1.0-SNAPSHOT.jar store person.dat Jeff "123 Penny Lane" London England
    555-555-555 123-123-123
```

6. Edit `Person.java` again and add back the `gender` instance variable:

```
private String gender;
```

7. Package your application as a JAR and try to load the previously serialized object:

```
$ mvn package
$ java -jar target/jshantz4-lab7-1.0-SNAPSHOT.jar load person.dat
```

   This time, we don't get an error. This is because, even though we changed the class, the version number that we specified in the class has not changed. When Java deserializes the object, it will simply set `gender` to null.

8. Edit `Person.java` and change the version number to `2L`.

9. Package your application as a JAR and try to load the previously serialized object:

```
$ mvn package
$ java -jar target/jshantz4-lab7-1.0-SNAPSHOT.jar load person.dat
```

   Once again, we get an error, since the version number of the class differs from that of the object serialized to disk. Hence, if we want to maintain backwards compatibility and be able to load objects serialized with a previous definition of a class, we need to ensure that we keep the class' version number constant, even if we make changes to the structure of the class itself.

   If we don't specify a version, Java will simply generate a version number for the class by *hashing* it. Every time we change that class, we'll be unable to load objects of the class that were previously serialized to disk. Hence, we should get in the habit of *always* defining `serialVersionUID` in a class that implements `Serializable`.

7 Submitting Your Lab

Commit your code and push to GitHub. To submit your lab, create the tag `lab7` and push it to GitHub. For a reminder on this process, see Lab 1.