Java Programming 2 – Lab Sheet 6

This Lab Sheet contains material based on Lectures 11—12.

The deadline for Moodle submission of this lab exercise is 5pm on Thursday 1 November 2018.

Default login details: Username = matric + 1st initial of family name, password = last 8 digits of library barcode (please change your password if you haven't done so already!).

Aims and objectives

- Implementing interfaces, particularly the **Comparable** interface
- Reading and writing files on the file system
- Further practice writing code with less concrete specifications and on finding solutions in the Java library.

Set up

1. Launch Eclipse as in previous labs (see the Laboratory 3 lab sheet for details)
2. In Eclipse, select **File** → **Import** … (Shortcut: **Alt-F, I**) to launch the import wizard, then choose **General** → **Existing Projects into Workspace** from the wizard and click **Next** (Shortcut: **Alt-N**).
3. Choose **Select archive file** (Shortcut: **Alt-A**), click **Browse** (Shortcut: **Alt-R**), go to the location where you downloaded `Laboratory6.zip`, and select that file to open.
4. You should now have one project listed in the Projects window: **Lab6**. Ensure that the checkboxes beside this project is selected (e.g., by pressing **Select All** (Shortcut: **Alt-S**) if it is not), and then press **Finish** (Shortcut: **Alt-F**).

Submission material

Again, this lab builds on the work we have done in previous labs. As part of the starter code, you have been provided with the **Monster** and **Trainer** classes from lab 5

Your tasks are as follows:

- To update the **Monster** class to ensure that monsters can be sorted, using the **Comparable** interface
- To update the **Trainer** class to allow **Trainer** objects to be saved to and written from files on the file system

Note that as part of implementing this, you may need to add other fields or methods to the **Monster** class – this is fine, as long as everything is properly documented and the test cases pass.

Sorting Monsters

Using the **Comparable** interface, implement a comparison method for **Monster** objects that sorts as follows:

1. Monsters should be sorted in **decreasing order of hit points.**
2. For two monsters with the same hit points, sort in **decreasing order of attack points**.
3. If two monsters have the same hit points and attack points, sort in **alphabetical order based on type**.

As a concrete example, consider the following list of **Monster** objects:

- **M1**: Fire monster, ap:150, hp:150
- **M2**: Water monster, ap:150, hp:150
- **M3:** Electric monster, ap:100, hp:200
- **M4**: Electric monster, ap:150, hp:100
- **M5**: Electric monster, ap:150, hp:100

After sorting, the above Monsters should be in the either of the following orders:

- **M3, M1, M2, M4, M5**.
- **M3**, **M1, M2, M5, M4**

Both are equally valid because **M4** and **M5** are identical as far as the sorting procedure is concerned.

The JUnit tests will provide various tests that you can use to verify that your comparison method is working as expected.

Saving and loading Trainer

This set of methods should allow **Trainer** objects to be saved to a file on the local file system and loaded from a previously saved file. The methods are as follows:

- **public void saveToFile (String filename) throws IOException** – this **instance** method should save the current **Trainer** to a file at the given location, or throw an appropriate exception if saving is not possible.
- **public static Trainer loadFromFile (String filename) throws IOException** – this **static** method should load a **Trainer** object from the given file (which is assumed to have been created with the **saveToFile()** method above) and return the loaded **Trainer**.

It is up to you to define a format for the saved trainers – the only requirement is that the **loadFromFile()** method should be able to re-create a trainer from the information written by the **saveToFile()** method. One possible implementation strategy could be the following:

- For the **saveToFile()** method:
    - Open the output file for writing
    - Write the **Trainer** name to the file
    - For each **Monster** in the trainer's list
        - Create a string representing the fields of that set
        - Write that string to the output file followed by a new line
- For the **loadFromFile()** method:

- Read the contents of the file into memory
- Use the first line of the file to obtain the name and create a new **Trainer** object
- For each remaining line in the file:
  - Extract the information from the line (e.g., by using methods such as **String.split()**, **String.substring()**, **String.indexOf()**, or similar built-in facilities)
  - Create a **Monster** based on the extracted details and add it to the trainer's set
- At the end, return the **Trainer**

Other strategies also exist – the only thing that will be tested is that your **saveToFile()** and **loadFromFile()** methods must work together successfully. Some other possible implementation techniques include:

- Making use of the **java.util.Properties** class (see https://docs.oracle.com/javase/8/docs/api/java/util/Properties.html for documentation)
- Making use of the **java.io.ObjectInputStream/java.io.ObjectOutputStream** classes and the **Serializable** interface (there is a reasonably good explanation at http://www.oracle.com/technetwork/articles/java/javaserial-1536170.html)

As part of your implementation, you might find it useful to **add additional properties and/or methods to the Monster class**. This is fine.

Testing your code

As in the previous labs, a set of JUnit test cases are provided to check the behaviour of your classes, in the file **test/TrainerTest.java** – please see the lab sheet for Lab 4 for instructions on using the test cases. You can use the test cases to verify that your code behaves as expected before submitting it. But note that **the test cases may not test every single possibility** – just because your code passes all test cases does not mean that it is perfect (although if it fails a test case you do know that there is almost certainly a problem).

If you want, as in previous labs, you can also write a class with a **main** method to test your code directly.

How to submit

You should submit your work before the deadline no matter whether the programs are fully working or not. Before submission, make sure that your code is properly formatted (e.g., by using **Ctrl-Shift-F** to clean up the formatting), and also double check that your use of variable names, comments, etc is appropriate. **Do not forget to remove any "Put your code here" or "TODO" comments!**

When you are ready to submit, go to the JP2 moodle site. Click on **Laboratory 6 Submission**. Click 'Add Submission'. Open Windows Explorer and browse to the folder that contains your Java source code – probably **H:\eclipse-workspace\Submission6\src** -- and drag only the *two* Java files **monster/Monster.java** and **trainer/Trainer.java** into the drag-and-drop area on the moodle submission page. **Your markers only want to read your java files, not your class files.** Then click the blue save changes button. Check the .java file is uploaded to the system. Then click **submit assignment** and fill in the non-plagiarism declaration. Your tutor will inspect your files and return feedback to you.

Outline Mark Scheme

Your tutor will mark your work and return you a score in the range "Excellent" (*****) to "Very poor" (*). Example scores might be:

5*: you completed the lab correctly with no bugs, and with correct coding style

4*: you completed the lab correctly, but with stylistic issues – or there are minor bugs in your submission, but no style problems

3*: there are more major bugs and/or major style problems, but you have made a good attempt at the lab

2*: you have made some attempt

1*: minimal effort