Java Programming 2 – Lab Sheet 6

This Lab Sheet contains material based on the lectures up to and including the content on file I/O and Swing programming.

The deadline for Moodle submission of this lab exercise is 4:30pm on Thursday 5 November 2020.

Aims and objectives

- Reading from and writing to text files
- Writing event handlers using Swing
- Continued practice with classes from the Collections framework

Set up

1. Download **Laboratory6.zip** from Moodle. **(This file will be provided on Friday.)**
2. Launch Eclipse as in previous labs
3. 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**).
4. 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.
5. You should now have one project listed in the Projects window: **Lab6**. Ensure that the checkbox 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 slightly modified versions of the **Move, Monster** and **Trainer** classes from Lab 5 (in the **monster** package), as well as GUI classes to display and manipulate **Trainer** and **Monster** objects (in the **gui** package). The **test** package contains JUnit test cases.

Your tasks are as follows:

- To update **Move**, **Monster**, and **Trainer** so that they are able to create strings that can be written to a text file, and so that strings from a text file can be parsed to recreate the objects
- To write the callback methods in the **TrainerFrame** and **AddMonsterDialog** classes to implement appropriate behaviour.

Implementing toStringForFile() and parse* methods

If you look at the provided **Move** class, you will see that two new methods have been added:

- The instance method **toStringForFile()** returns a String containing all of the properties of the **Move**
- The static method **parseMove()** processes a string which is assumed to have been created by **toStringForFile()** and returns a new **Monster** object with the given properties.

These methods together allow instances of the **Move** class to be written to and recreated from text files.

The first task is to add similar methods to the **Monster** and **Trainer** classes to allow objects of each class to be converted to and from a string representation, so that those objects can also be saved to and loaded from text files. As the structure of **Monster** and **Trainer** are more complex than that of **Move**, the methods for both classes will use a List<String> rather than a String on its own.

For **Monster**, the methods required are as follows:

- **List<String> toStringForFile()** – this method should return a list of Strings corresponding to the properties of the Monster. You should use **Move.toStringForFile()** to create the strings corresponding to the moves of the monster. This method is **not static**.
- **static Monster parseMonster (List<String> spec)** should read a list of String objects and recreate the corresponding Monster object. You should use **Move.parseMove()** to reconstruct the Move objects.

You should implement the above two methods together – that is, you should ensure while implementing **toStringForFile()** that its output can be successfully parsed by **parseMonster**. The specific format of the output does not matter – the important thing is that these two methods work together.

Once you have implemented the required behaviour for **Monster**, you should do the same for **Trainer**. The method signatures are as follows:

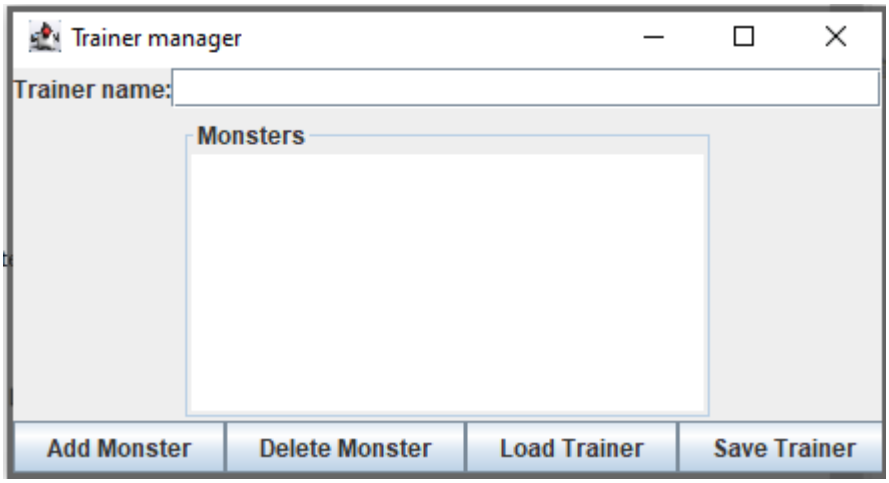
- **List<String> toStringForFile()** – this method should return a list of Strings corresponding to the properties of the Trainer (i.e., the name and the list of Monsters). You should use **Monster.toStringForFile()** to create the strings corresponding to the monsters. This method is **not static**.
- **static Trainer parseTrainer (List<String> spec)** should read a list of String objects and recreate the corresponding Trainer objects. You should use **Monster.parseMonster()** to reconstruct the Monster objects.

Again, the methods should be written to work together, and the specific format of the output does not matter – the important thing is that the methods work together properly. That is, **parseTrainer** should be able to process the output of **toStringForFile** and recreate an identical Trainer object.

The provided JUnit tests will verify that your implementations behave as required.

Implementing GUI behaviour

The code you have been given in the **gui** package is the start of a Swing "trainer manager" application. If you run the main method of the **TrainerFrame** class, you will see a window pop up similar to the following:



Note that when you click on the buttons, nothing happens for the moment. Your job is to write the back-end code to turn this window into an interactive GUI application.

Internal structure of TrainerFrame class

There is a lot of code in the **TrainerFrame** class, much of it involved in actually laying out the user interface. Here is a summary of the overall structure:

- Each graphical component on the screen is an instance field – for example, the **addButton** field represents the "Add Monster" button
- The on-screen list of monsters is represented by the **JList** called **monsterList**
- The underlying list of displayed monsters is stored in the **monsterModel** field, which is of type **DefaultListModel<Monster>**.
    - Note: as mentioned in the lecture, **DefaultListModel** is very similar to **ArrayList** – it provides methods for adding and removing elements and other similar operations. The method names are a bit different from **ArrayList**, though, so you might need to read the documentation at https://docs.oracle.com/en/java/javase/15/docs/api/java.desktop/javax/swing/DefaultListModel.html for details
- The **TrainerFrame** constructor creates all of the components and lays them out on the screen, and also sets up the behaviour of the top-level window – e.g., it ensures that the program exits when the window is closed.
- **TrainerFrame** also implements **ActionListener**, which means that it provides an **actionPerformed()** method. This method is registered with all of the on-screen buttons, which ensures that whenever any of the buttons is pressed, the **actionPerformed()** implementation is called.

Behaviour to implement

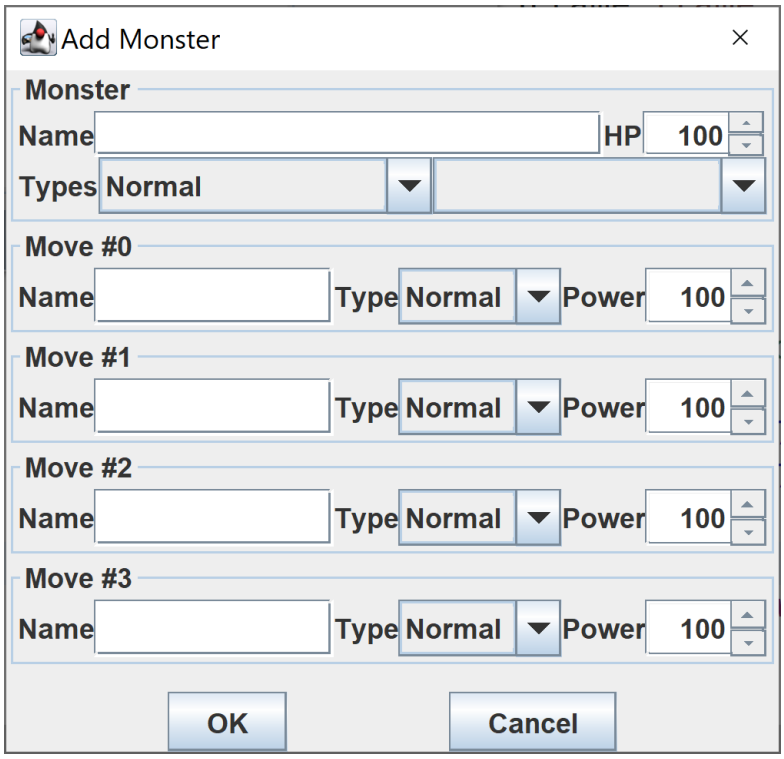
The following is the required behaviour for each button. Note that I have already provided implementations for the "add" and "save" behaviours; you must add the behaviour for the other buttons.

- **Add**: when this buttons is clicked, your class should create and display a new instance of the provided **AddMonsterDialog** class. The required behaviour of this dialog is specified below. **This behaviour is provided for you.**
- **Delete**: when this button is clicked, you should check whether a monster is selected in the monster list. If none is selected, the button should do nothing; if a monster is selected, then it should be removed from the underlying model (which will remove it from the displayed list).
- **Load Trainer**: when this button is clicked, you should load a new **Trainer** object from the file **trainer.txt** in the current working directory and use its details to populate the on-screen fields. Use **Trainer.parseTrainer()** to read the contents of the file.
- **Save Trainer:** when this button is clicked, you should create a new **Trainer** based on the on-screen fields and save it to the file **trainer.txt** in the current working directory. Use **Trainer.toStringForFile(). This behaviour is provided for you.**

All of the above behaviour should be implemented inside the **TrainerFrame.actionPerformed()** method.

Adding a Monster

As mentioned above, the process of adding a Monster to the list should be handled in the **AddMonsterDialog** – when the "Add" button is pressed, the **MonsterFrame** creates a new **AddMonsterDialog** with appropriate parameters and makes it visible through **setVisible(true)**. Here is what the dialog box looks like on screen:



The following is the structure of **AddMonsterDialog**:

- The associated **DefaultListModel** object is stored in the field **model**, which allows the **AddMonsterDialog** to modify the list of Monsters when needed. (see below).
- Each graphical component is represented by a field – e.g., **type1Combo** is the combo box representing the first type of the monster.
- The graphical components corresponding to the moves are all stored in arrays – for example, **moveNameFields[2]** is the text field corresponding to the name of the move #2, while **movePowerSpinners[1]** is the spinner box corresponding to the power of move #1.
- **AddMonsterDialog** implements **ActionListener**, which means that it also has an **actionPerformed** method. This method is called whenever the "OK" and "Cancel" buttons are pressed.

A skeleton **actionPerformed** method has been created for you, and I have already implemented the behaviour for the "Cancel" button (the dialog box is simply closed by calling **dispose()**).

You should implement the following behaviour to respond to a press on the "OK" button.

1. Get the name of the Monster via **nameField.getText()**
2. Get the hit points of the Monster via **(int)hitPointSpinner.getValue()**.
    a. Since **getValue()** returns an Object, you need to cast the result to an int as above
3. Get the type(s) of the Monster via **(String)type1Combo.getSelectedItem()** and **(String)type2Combo.getSelectedItem()**.
    a. Since **getSelectedItem()** returns an Object, you need to cast the results to a String as above
4. Create a new **Monster** using the appropriate constructor
    a. If **type2** is the empty string, then use the one-type constructor, otherwise use the two-type constructor
5. For each Move number **i** between 0 and 3, do the following:
    a. Check the name via **moveNameFields[i].getText()**. If the name is empty, do not create a Move in this position.
    b. Otherwise, get the other information about the Move from the other fields:
        i. Get the type with **(String)moveTypeCombos[i].getSelectedItem()**
        ii. Get the power with **(int)movePowerSpinners[i].getValue()**
    c. Create a new Move with the given parameters and add it to the Monster at position **i** using **Monster.setMove()**
6. Finally, the newly created **Monster** should then be added to the **DefaultListModel** so that it is displayed properly (use **monster.addElement()**), and the dialog should be closed afterwards using **dispose()**.

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/TestLab6.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. Note that every time you run the tests, they will delete the "trainers.txt" file in your working directory as part of testing your code.

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 **…/eclipse-workspace/Lab6/src**-- and drag only the *five* Java files **monster/Monster.java, monster/Move.java, monster/Trainer.java, gui/TrainerFrame.java** and **gui/AddMonsterDialog.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

Possible (optional) extensions

Here are some additional things to try if you want further practice with GUI programming:

- At the moment, all of the buttons are always enabled, whether the associated action can actually be executed or not. See if you can work out how to enable/disable the buttons dynamically depending on what the user has selected (e.g., **Delete** should only be active when a monster is selected in the main TrainerFrame)
- At the moment, the Trainer objects are only created when loading/saving. Try updating the code to link the ListModel directly to the Trainer object (hint: you probably want to subclass **AbstractListModel**)
- The **Save** and **Load** implementations both use a hard-coded file name. Try updating the code to use a **JFileChooser** instead to choose a file to load or save.
- The current set-up doesn't allow a Monster to be edited after it is created. Try modifying the code so that when you select a Monster from the list, you are able to update its properties.