Java Programming 2 – Lab Sheet 4

This Lab Sheet contains material based on Lectures 7—8.

The deadline for Moodle submission of this lab exercise is 5pm on Thursday 18 October 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

- Further practice with object-oriented modelling in Java
- Practice with throwing Exceptions
- Gain experience of **refactoring** – reusing and modifying existing code
- Gain experience of creating new classes from scratch without starter code
- Particular focus on subclasses, abstract classes, and overriding methods
- Introduction to unit testing with the JUnit library

Set up

The Laboratory4.zip file will not be made available until after the tutorial on Friday 12 October, as it includes a sample solution to Laboratory 3.

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 `Laboratory4.zip`, and select that file to open.
4. You should now have one project listed in the Projects window: **Lab4**. 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

This exercise builds on the material that you submitted for Laboratory 3, so it might be worth referring back to your work on that lab before beginning this one.

Recall that the **Monster** class developed in Laboratory 3 represents a (simplified) record of a monster from a monster battling game: including a type, values for hit points and attack points, and a list of types against which the monster is weak. You have been provided with a sample implementation of **Monster**. Your task in this lab is to **refactor** this class into a set of classes that are able to represent every type of monster as its own class, instead of using the **type** field to distinguish them. The information that is common to all monsters will be retained in the **Monster** class, which

will be made **abstract**, while other information that is relevant only to one of the monster types will be put into the appropriate subclass.

The following sections describe the modifications and additions that must be made as part of this refactoring task. **Please read through the whole specification and make sure that you understand what is involved before beginning.**

Subclasses and Fields

You must make the **Monster** class abstract, and then create three concrete subclasses of **Monster**: **FireMonster, WaterMonster,** and **ElectricMonster**, each of which will represent monsters of that specific type only.

- **Monster** should contain all of the fields and methods as previously. You should change the visibility of the fields in **Monster** so that they are visible in the subclass.
- Each of the **Monster** subclasses should be defined as possible:
    - Its constructor should take only two parameters, **hitPoints** and **attackPoints**
    - The constructor should call the superclass constructor with appropriate arguments using the **super** keyword. The additional arguments to pass to the super-class constructor should be as follows:
        - For **FireMonster**: type "Fire", weakness { "Water" }
        - For **WaterMonster**: type "Water", weaknesses { "Fire", "Electric" }
        - For **ElectricMonster**: type "Electric", weaknesses { }

To create a new class in Eclipse: right click on the project name and choose **New – Class** (you can also reach this dialogue by pressing **Ctrl-N** and then choosing **Java** and then **Class** in the resulting wizard). Fill in the class name in the **Name** field and the superclass (if any) in the **Superclass** field. If the superclass is abstract, you can also tick the box beside "Inherited abstract methods" to create initial versions of all abstract methods; you can also add the methods later, so this is not necessary. Press **Finish** (Shortcut: **Alt**-F) and the new class will be added to your project.

MonsterException

You must also create an additional class, **MonsterException**, which is a subclass of **Exception**. This class should consist only of a constructor which takes one parameter, a **String**, and calls the corresponding super-class constructor (i.e, use the constructor of **Exception** which also takes a single **String** parameter).

dodge()

Add an abstract **dodge()** method to the parent **Monster** class – this method should return a **boolean** value and will be implemented in the subclasses to implement the modified **attack()** behaviour described below. The method should have a **protected** access modifier.

The required behaviour for each subclass is as follows – you should add any necessary fields to each subclass to implement this behaviour:

- **FireMonster:** this method should alternatively return **true** and **false** – that is, the first time it is called, it should return **true**, the next call should return **false**, and so on

- **WaterMonster:** this method should return **true** if the monster's hit points are at least 100, and **false** if they are less than 100.
- **ElectricMonster:** this method should always return **false** – that is, an electric monster should never dodge when attacked.

attack()

The final piece of refactoring is to modify the **attack()** method of **Monster** in several ways:

- The return type should be changed from **boolean** to **void**
- The signature should be modified to indicate that the method might throw a **MonsterException**
- Every case where the original method returns **false** should be modified to instead throw a **MonsterException** with an appropriate **String** message.

Also, assuming that no exception is thrown, the behaviour of the **attack()** method should be updated to use **dodge()** as follows:

- First, call **dodge()** on the monster being attacked.
    - If the result is **false**, the attack behaviour as before is implemented.
    - If the result is **true**, no hit points are removed from the monster being attacked, but 20 hit points are removed from the monster doing the attacking. The same rules apply here – if the monster's HP goes below zero, then it should be set to zero.

Unit tests

As you have seen previously, if you introduce a syntax error into the Java code, Eclipse will indicate it with a red X and your code will not run until the error is fixed. But what if your code compiles properly, but you've written it wrong – that is, what if your program compiles and runs, but it does the wrong thing?

Java (and Eclipse) provide a simple way to test for this sort of problem: you can define **unit tests** that describe how your program should run, and then you can use those tests to see if the behaviour is what you expected. This project includes a set of unit tests in the source file **TestMonster.java**. (Don't worry about the details of how the unit tests are written – we will get to that topic later in the course). You can run these tests by right clicking on the class in question and choosing **Run as – JUnit Test** (Shortcut: **Alt-Shift-X, T**). This will run all of the tests in the file and present the results in a new view titled "JUnit".

If all of the tests pass, there will only be green ticks in the JUnit window – but if something goes wrong, then it will show up as a "Failure", and you can look in the "Failure Trace" window to see what the issue is.

Whenever possible from now onwards, I will provide JUnit test cases along with every lab. 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).

Also, while you are testing your code, please **make sure that you do not modify the test cases** – we will be testing your code against the original test cases, so if you modify a test case, your code will likely not pass our tests. If your code is failing a test, you must modify your code to fix the failure rather than changing the tests.

Suggested development process

The initial **Laboratory** project contains two files:

- **Monster.java** – a sample solution to Laboratory 3,
- **TestMonster.java** – a set of JUnit tests that can be used to test that your code works as expected. See the Unit tests section on the previous page for hints on how to use these tests to verify your code.

Note that, as you will not have yet completed the refactoring, the test cases will all fail when you first run them, and many will continue to fail until you have completed the assignment.

You should carry out all of the steps described in the preceding section to refactor the code.

Hints and tips

- The refactoring process will probably "break" the code as you carry out the above steps – for example, if you move some fields but do not yet move the methods that refer to those fields, you will see a lot of errors. You can use these errors to help you decide what edits to make next – but please make sure that the class design in the end product is the same as the diagram on the preceding page.

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 4 Submission**. Click 'Add Submission'. Open Windows Explorer and browse to the folder that contains your Java source code – probably **H:\eclipse-workspace\Lab4\src** -- and drag only the *five* Java files **Monster.java**, **MonsterException.java**, **FireMonster.java, WaterMonster.java,** and **ElectricMonster.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 three .java files are 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 via moodle.

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 refactoring correctly with no bugs, and with correct coding style

4*: you completed the refactoring 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 refactoring

2*: you have made some attempt at both submissions

1*: minimal effort