Java Programming 2 – Lab Sheet 4

This Lab Sheet contains material based on the lectures up to and including the material on exceptions and programming style.

The deadline for Moodle submission of this lab exercise is 4:30pm on Thursday 22 October 2020.

Aims and objectives

- Using the Eclipse IDE to develop more complex Java code
- Refactoring existing code
- Using interfaces
- Working with exceptions

Background: Type effectiveness

This exercise builds on the material that you submitted for Laboratory 3. While you will make various changes to the structure of the code, the main modification will be the addition of **type effectiveness**. This section provides a short summary of the relevant aspects.

Every Monster has one or two types, and every Move has exactly one type. When a Monster attacks another Monster using a specific Move, the impact of that attack depends on four factors:

- The type of the Move
- The type(s) of the attacking Monster
- The type(s) of the defending Monster
- The base power of the Move itself

Every Move type is effective against certain types, and less effective against other types. The effectiveness can be one of three levels:

- Not very effective
- Effective
- Super effective

For this lab, we will consider only the following five types: *Normal, Fire, Water, Electric, Grass*, and will use the following effectiveness chart. On the chart, the attacking type is shown down the side, and the defending type is shown across the top. If an entry is blank, that indicates normal effectiveness. For example, the table shows that a Fire-type Move is super effective against a Grass type Monster, while a Grass-type Move is not very effective against a Fire-type Monster.

	Normal	Fire	Water	Electric	Grass
Normal					
Fire		Not very	Not very		Super
Water		Super	Not very		Not very
Electric			Super	Not very	Not very
Grass		Not very	Super		Not very

These effectiveness values are treated as multipliers on the power of a Move: a “not very effective” move would have its power multiplied by 0.5, while a “super effective” Move would have its power multiplied by 2.0. If the defending Monster has two types, the effectiveness is determined individually for each type and multiplied together.

As a final factor, if a Monster uses a Move that matches one of its own types, the power of that Move is boosted by a factor of 1.5 – for example, this would apply if a Normal-type Monster uses a Normal-type Move, or a Grass+Water-type Monster uses a Grass-type or Water-type Move.

As a concrete example, consider the following two Monsters with the given Moves:

Heliolisk (Electric, Normal)

- Thunderbolt (Electric): 90
- Quick Attack (Normal): 40
- Hyper Beam (Normal): 150

Ludicolo (Grass, Water)

- Energy Ball (Grass): 100
- Hydro Pump (Water): 110
- Fire Punch (Fire): 75

If Heliolisk attacks Ludicolo with Thunderbolt, the effective power would be computed as follows:

90	*base power of Thunderbolt*
* 0.5	*Electric is not very effective against Grass*
* 2.0	*Electric is super effective against Water*
* 1.5	*Same-type attack bonus for Electric*
135	***Effective power***

Similarly, if Ludicolo attacks Heliolisk with Fire Punch, the effective power would be:

75	*Base power of Fire Punch*
* 1.0	*Fire has normal effectiveness against Electric*
* 1.0	*Fire has normal effectiveness against Normal*
* 1.0	*No same-type bonus*
75	***Effective power***

Submission material

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

Recall that the **Monster** and **Move** classes developed in Laboratory 3 represent a (simplified) record of a monster from a monster battling game, including one or two types and up to four moves that can be used in battle. You have been provided with a sample solution to Lab 3. Your task in this lab is to **refactor** these classes to reflect the fact that the type – of Monsters and of Moves – is of crucial importance in the battling game, and should be represented in a more principled way than in the original implementation. You will also add error checking throughout to ensure that the values for types and move powers are valid.

To start with, you should import the file **Lab4.zip** into Eclipse – see the "Eclipse intro" sheet for instructions on how to do this. Once you have imported it, you should have a project **Lab4** in your workspace containing four files in the **src** directory: **Monster.java, Move.java, Main.java, TestMonster.java**.

It is suggested that you use the provided versions of Monster.java and Move.java as the starting point for your work, but if you would prefer to start with your own implementation that is acceptable. Just be sure that the behaviour described here is all implemented properly.

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.**

TypedItem interface

First, you need to implement a **TypedItem** interface to represent an object in the system that has one or more types. This interface should include two instance methods, as follows:

- **boolean hasType (String type);**
- **String[] getTypes();**

To add a new interface in Eclipse, use the **File** menu, choose **New**, then **Interface**. Give the interface a name in and then press **Finish**.

Using the TypedItem interface in the Monster and Move classes

The next step is to modify the **Monster** and **Move** classes so that they implement the **TypedItem** interface. You should modify the methods of both classes so that they use only the above methods for manipulating types, and remove any existing type-related methods that do not use those methods (e.g., **Move.getType()**).

Add static methods to TypedItem

Next, you should add the following two **static** methods to the **TypedItem** interface. As these methods are static, you will be able to write the method bodies even though **TypedItem** is an interface.

- **static boolean isValidType (String type)**
- **static double getEffectiveness (String attackType, String defendType)**

The **isValidType** method should return **true** if the parameter is one of the five types mentioned above (*Normal, Fire, Water, Electric, Grass*) and **false** otherwise.

The **getEffectiveness** method should return the appropriate effectiveness value for the two types, based on the chart on page 1. For "super" effectiveness, the return value should be 2.0; for "not-very" effectiveness, the return value should be 0.5; while for "normal" effectiveness, the return value should be 1.0.

Error checking

At the moment, all of the constructors and methods assume that all input (types, move powers, array indices) is valid. You should add checking to the following methods. In all cases, if a check fails, you should throw an **IllegalArgumentException** with an appropriate error message. Note that **IllegalArgumentException** is an unchecked exception, so you should not need to make any other changes to your code.

- **Monster** constructors
    - All types should be valid according to **TypedItem.isValidType**
    - For the two-type constructor, the two types should be different
- **Move** constructor
    - Type should be valid according to **TypedItem.isValidType**
    - Power value should be between 0 and 180 (inclusive)
- **Monster.setMove, Monster.getMove**
    - Index value should be in range (between 0 and 3 inclusive)

Computing effective power and choosing Moves

The final task is to add two methods to **Monster**: one to compute the effective power of a given Move when attacking another Monster, and one to choose the best Move to use. These methods should have the following signature:

- **public double getEffectivePower (Move move, Monster defender)**
- **public Move chooseMove (Monster defender)**

getEffectivePower should compute the effective power of a Move. In summary, the effective power of a move is the base power, multiplied by the type effectiveness of that move against each of the defender's types, multiplied by any same-type attack bonus if relevant. See the worked examples on Page 2.

chooseMove should go through all of the non-null Moves of the Monster and should return the one with the highest effective power against the given defending Monster.

For example, with the Monster examples on Page 2, the expected output is as follows:

- **heliolisk.chooseMove (ludicolo)** returns Hyper Beam
- **heliolisk.chooseMove (heliolisk)** returns Hyper Beam
- **ludicolo.chooseMove (ludicolo)** returns Energy Ball
- **ludicolo.chooseMove (heliolisk)** returns Hydro Pump

If the Monster has no non-null Moves, this method should return null.

Main class

The provided class **Main** (in the file **Main.java**) includes a **main** method which creates several Monsters and Moves and calls **chooseMove** on all of them. This class can be run in Eclipse (**Run -> Run as -> Java application**) and the output is shown in the Console view. You can edit this method as you want, e.g. by adding or removing Monsters or adding calls to other methods, to test your code. You might also want to add debugging println statements to various methods to help in debugging – just don't forget to **remove any such statements before submitting.**

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".

Note that all of the tests will probably fail when you first run them, and will continue to fail until you have completed the refactoring. This is expected.

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.

Hints and tips

1. 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 described above.
2. Don't forget to update the comments to reflect the new behaviour of the Monster and Move classes, and to add appropriate comments to the new methods!
3. Eclipse is able to automatically clean up code formatting and indentation – if you press **Ctrl-Shift-F** or go to **Source – Format** then your current source file will be reformatted. I strongly suggest doing this before submitting to ensure that you do not lose any marks for code style.

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, and also remove any debugging println statements!**

When you are ready to submit, go to the JP2 Moodle site. Click on **Laboratory 4 Submission**. Click 'Add Submission'. Browse to the folder that contains your Java source code – probably **.../eclipse-workspace/Lab4/src/** -- and drag only the *three* Java files **Monster.java, Move.java, TypedItem.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 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" (*). We will automatically execute your submitted code and check its output; we will also look at the code before choosing a final mark.

Example scores might be:

5*: you complete the code correctly with no bugs and correct style

4*: you compute the class mostly correctly with minor bugs OR your code is correct but the style is inappropriate

3*: more major bugs and/or more major style issues

2*: some attempt made

1*: minimal effort

For this assignment (and all future ones), we will also be considering code style – comments, formatting, variable names, etc – in addition to correctness. You will be deducted one star if your code style is not appropriate. Please check with the tutors and demonstrators during your lab session if you are uncertain about style.

Possible extensions

If you have completed the core tasks of the lab and want to try something extra, here are some things to try. Again, we will likely do many of these things over the next few weeks, but you can try your own approach in advance.

- Implement the full type effectiveness chart from https://pokemondb.net/type. See if you can find a good data structure to represent the information in that chart efficiently.
- Implement battling between monsters, incorporating the type effectiveness information.
- In addition to power and type, moves in the full game also have several other features that affect their use in battle, including accuracy, power points ("PP"), and type (physical/special/status). Try modelling all of these additional attributes.