Introduction to Object Orientation Programming 2: Lab Sheet 4

This Lab Sheet contains material based on Lecture 4.

The deadline for Moodle submission of this lab exercise is 12:00 on Thursday 24 October 2024.

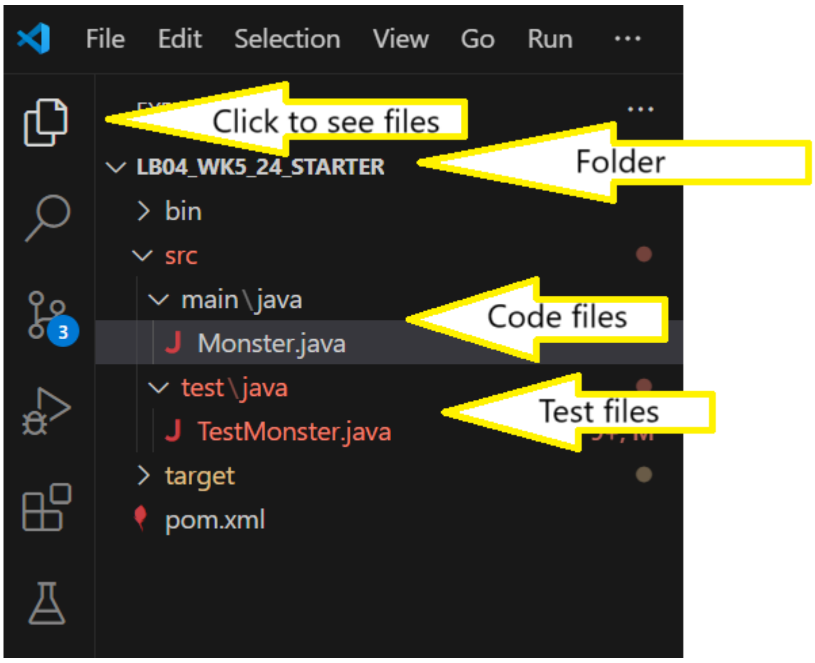
Aims and objectives

- 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 LB04_WK5_24_starter.zip file will not be made available until 17:00 on Thursday 17 October 2024 , as it includes a sample solution to LB03_WK4_24.

1. Download and unzip **LB04_WK5_24_starter.zip file**
2. Launch Visual Studio as in previous lab (see the LB03_WK4_24 lab sheet for details)
3. In Visual Studio, select **File → Open Folder**
4. Go to the location where you unzipped **LB04_WK5_24_starter**, and highlight that folder and click on **Select Folder**
5. You should now be able to see all of your files in the explorer (if not click on **View → Explorer)**



Submission material

This exercise builds on the material that you submitted for LB03_WK4_24, so it might be worth referring back to your work on that lab before beginning this one.

Recall that the **Monster** class developed in LB03_WK4_24 represents a (simplified) record of a monster from a monster battling game: including a type and a list of attacks. 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.

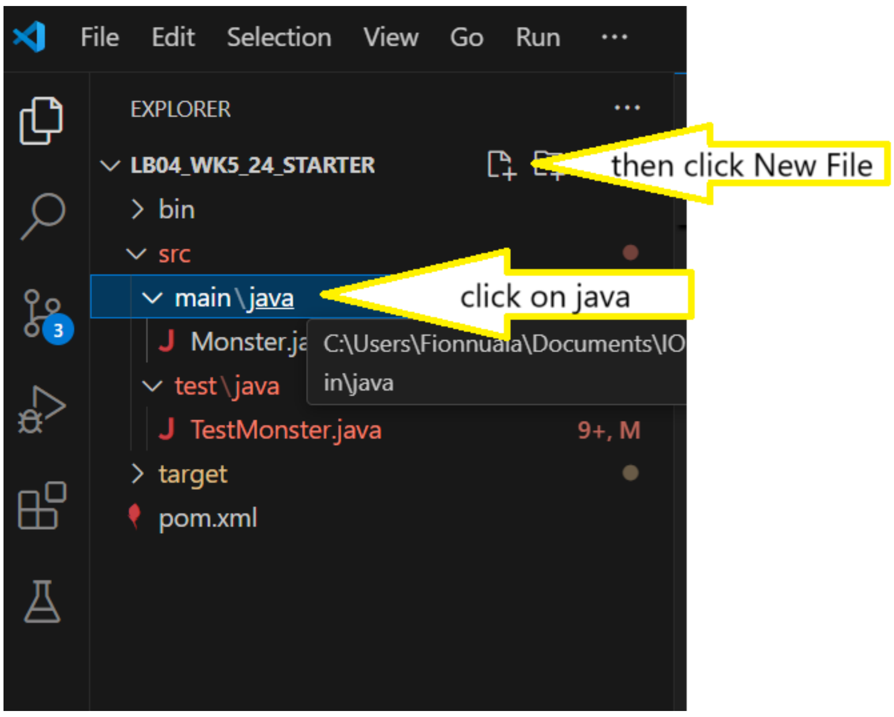
The following sections describe the modifications and additions that must be made as part of this refactoring task. **Please read through the whole specification and make sure that you understand what is involved before beginning.**

Attack class

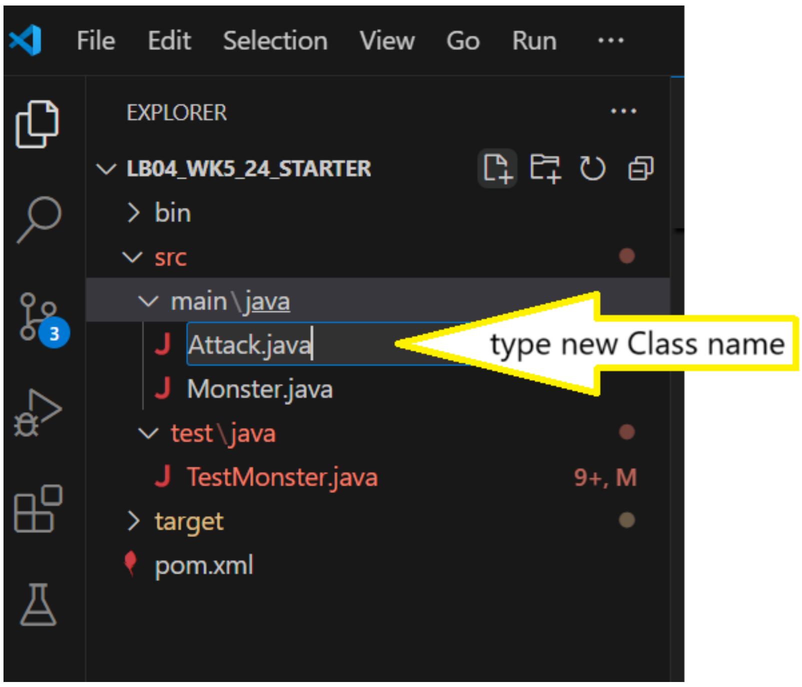
You should create an **Attack** class to hold the details of an attack. This class should have two fields, a string representing the name and an integer representing the points; it should also have a constructor that sets those two fields, getter methods for both fields, as well as an overridden **toString()** implementation that produces a nicely formatted string representation of the attack.

You should change the **Monster** class so that it stores an array of **Attack** objects instead of the two parallel arrays of attack names and attack points, and should change the **Monster** constructor to use an array of **Attack** instead of the two arrays as now.

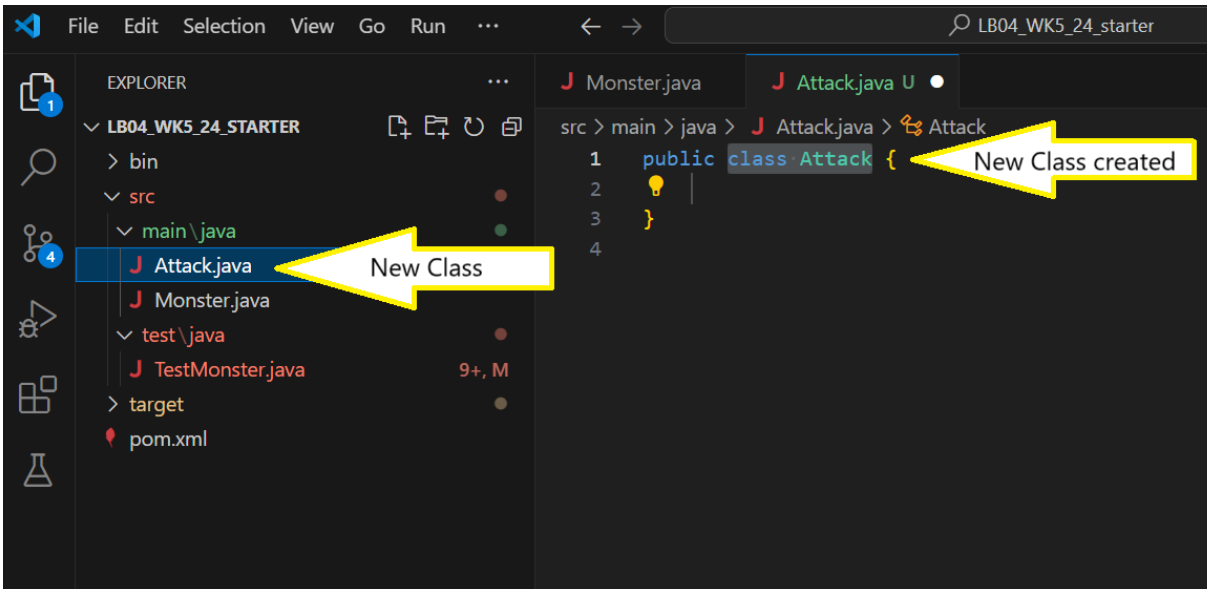
To create a new class in Visual Studio: In the explorer under **scr** click on the project **java** in **main\java** and choose **New File** icon.



Fill in the file name remembering to start with a capital letter and giving the extension .java



Press enter and a new class with starter file will be created.



```java
public class Attack {

}
```

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 (with the modification above to the representation of attacks). 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 **attacks**
    - 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"
        - For **WaterMonster**: type "Water"
        - For **ElectricMonster**: type "Electric"

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. You should change the visibility of the **dodge()** method so that they are visible in the subclass.

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:

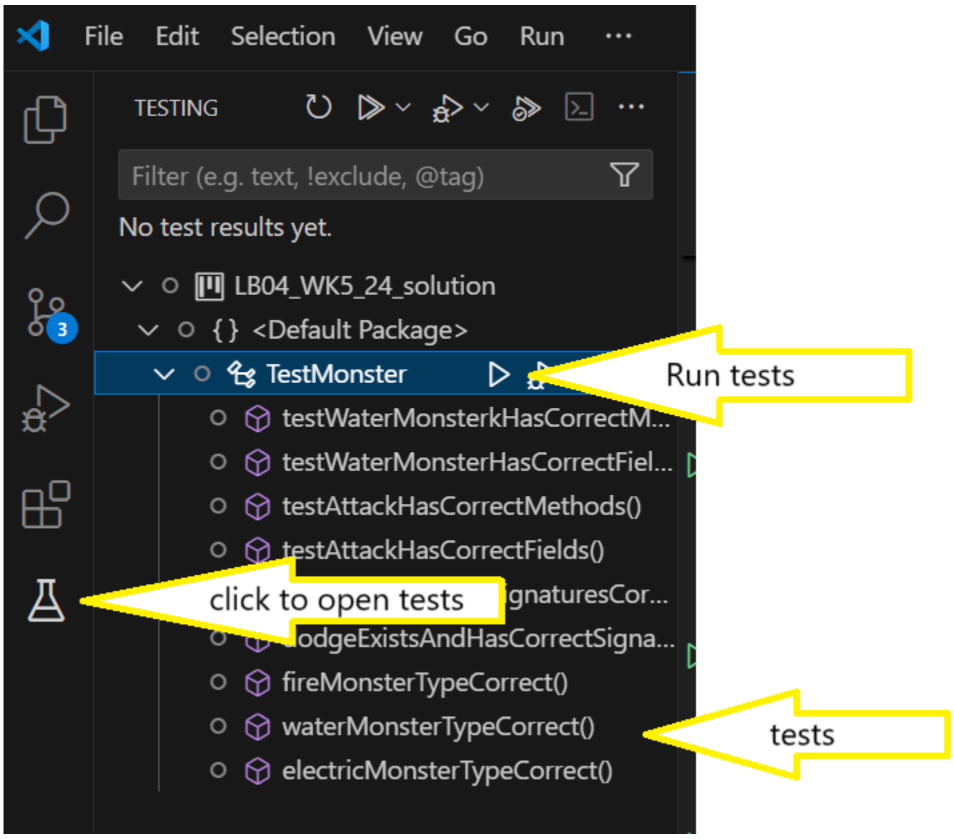
- First, call **dodge()** on the monster being attacked.
    - If the result is **false**, the attack behaviour as before is implemented.

- If the result is **true**, no hit points are removed from the monster being attacked, but 10 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, Visual Studio will indicate it with a squiggly red line under 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?

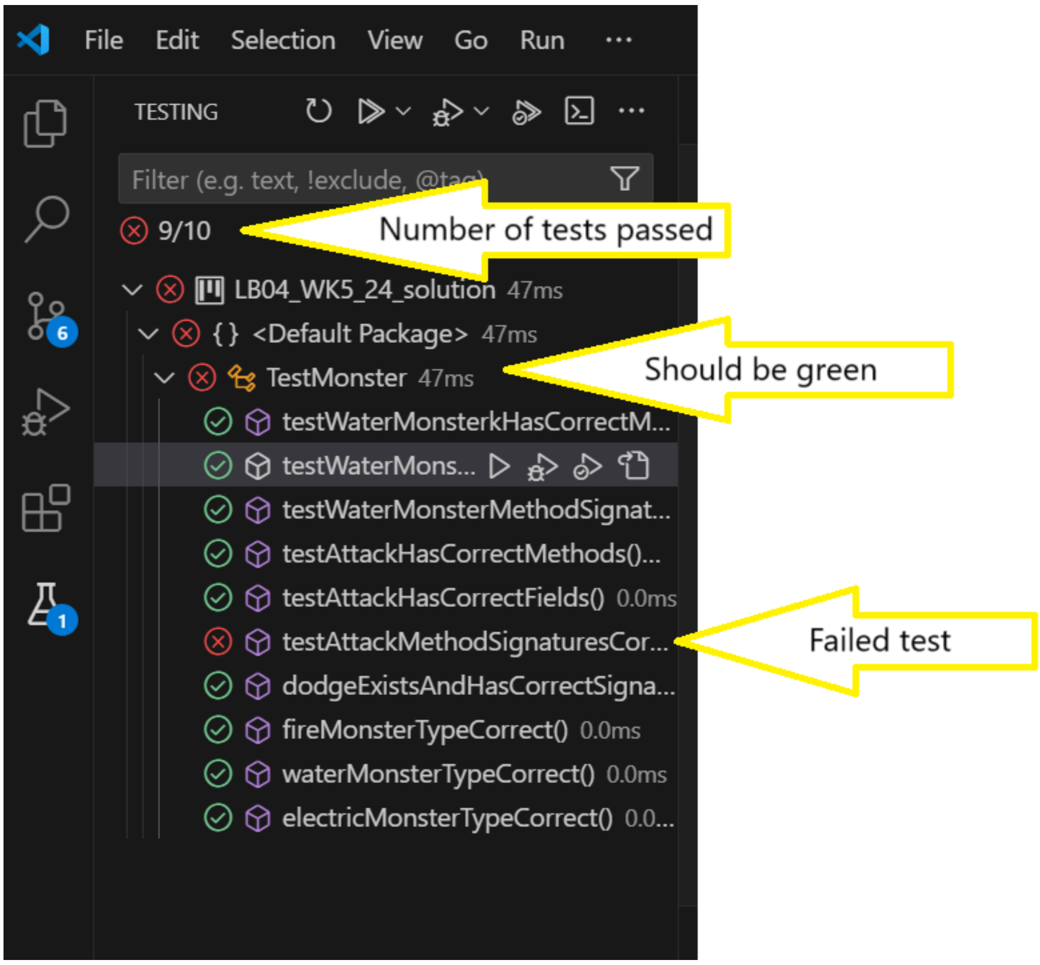
Java (and Visual Studio) provide a simple way to test for this sort of problem: you can define **unit tests** that describe how your program should run, and then you can use those tests to see if the behaviour is what you expected. This project includes a set of unit tests in the source file **TestMonster.java**. (Don't worry about the details of how the unit tests are written – we will get to that topic later). You can run these tests by clicking on the test symbol in the sidebar. This should open the test file and run the tests. You can rerun the tests by clicking on any of the individual arrows of each test



No test results yet.

- LB04_WK5_24_solution
  - { } <Default Package>
    - TestMonster
      - testWaterMonsterkHasCorrectM...
      - testWaterMonsterHasCorrectFiel...
      - testAttackHasCorrectMethods()
      - testAttackHasCorrectFields()
      - ...gnaturesCor...
      - ...odgeExistsAndHasCorrectSigna...
      - fireMonsterTypeCorrect()
      - waterMonsterTypeCorrect()
      - electricMonsterTypeCorrect()

If a tests passes it will have a green tick if it fails it will have a red cross. The number of tests that pass will be displayed at the top of the test. The name of the test will indicate why your code has failed. Fixing it should pass the test.



```
9/10
LB04_WK5_24_solution 47ms
  { } <Default Package> 47ms
    TestMonster 47ms
      testWaterMonsterkHasCorrectM...
      testWaterMons...
      testWaterMonsterMethodSignat...
      testAttackHasCorrectMethods()...
      testAttackHasCorrectFields() 0.0ms
      testAttackMethodSignaturesCor...
      dodgeExistsAndHasCorrectSigna...
      fireMonsterTypeCorrect() 0.0ms
      waterMonsterTypeCorrect() 0.0ms
      electricMonsterTypeCorrect() 0.0...
```

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 is in the correct format. But note that **the test cases may do not test whether your code behaves as expected** – your tutors will be given more test to verify this. It is expected that the tests should pass before you submit your code (otherwise the tutors will not be able to run their tests) but you should submit your code even if they do not pass

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 **LB04_WK5_starter** project contains two files:

- **Monster.java** – a sample solution to LB03_WK4_24,
- **TestMonster.java** – a set of JUnit tests that can be used to test that your code is in the expected format.

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 described above.
- Don't forget to update the comments to reflect the new behaviour of the Monster class!

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 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 IOOP2 moodle site. Click on **LB04_WK5_24_submission**. Click ‘Add Submission’. Open Windows Explorer and browse to the folder that contains your Java source code -- and drag only the *six* Java files **Attack.java, 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 six .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 **A1** to **H**.