CSCU9A3 Practical 2

Unit Testing and JUnit

Intro

In this practical class, we are going to learn how to configure it to use JUnit and how to use it to create your own unit tests.

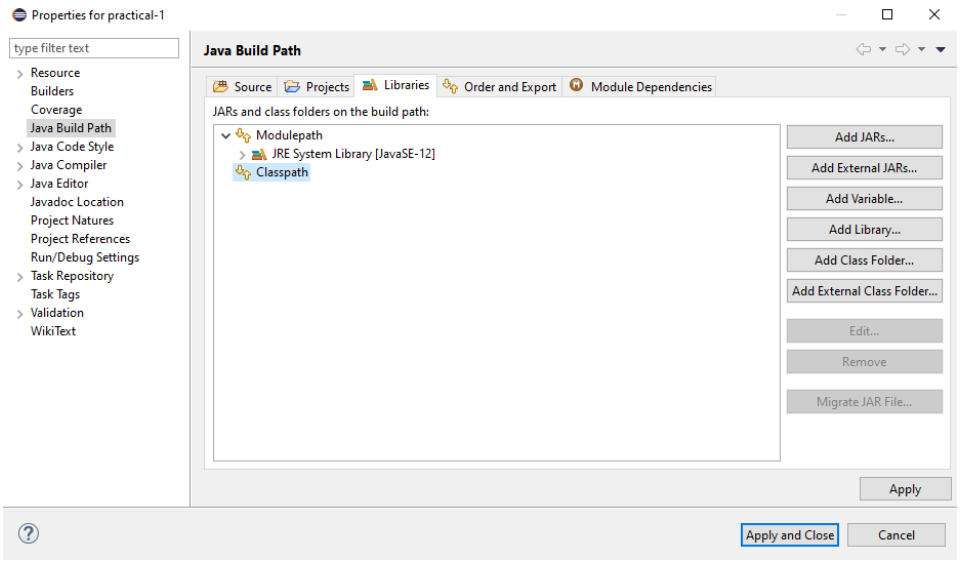
Getting Started

A skeleton class for this practical has been created for you. It can be download on Canvas.

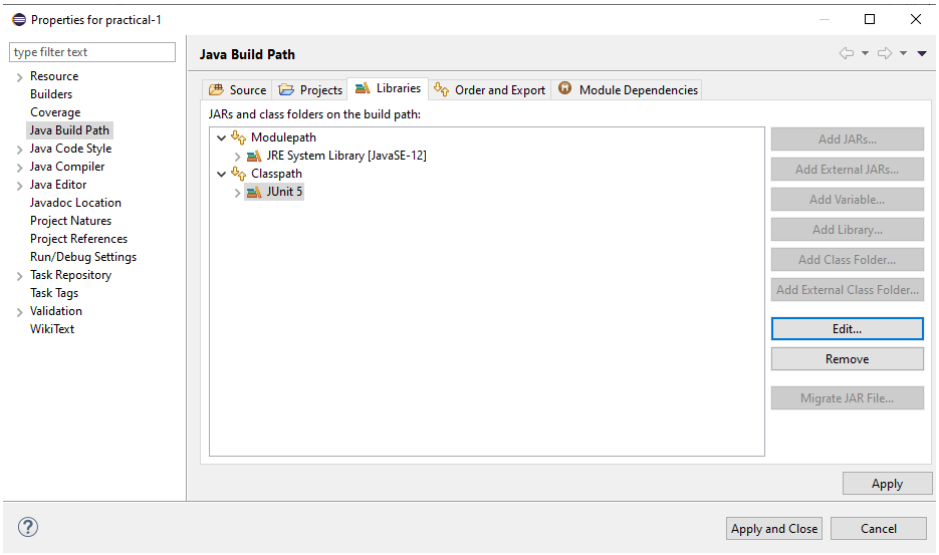
Copy these files to a new project directory and open them in your desired IDE. If you want to use Eclipse but don't remember how to import the code in this IDE, check the last practical. Below is an introduction on how to configure JUnit in your Eclipse project.

Configuring JUnit

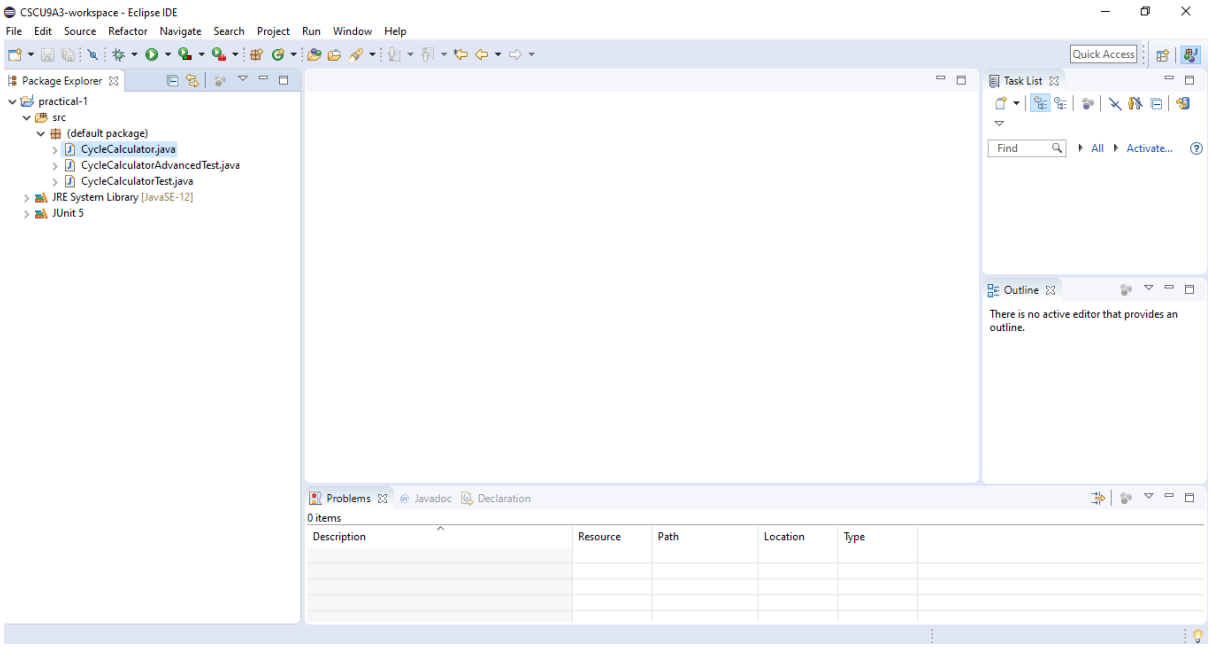
After importing the code related to this practical, you will notice that there are some errors in this workspace. This is because we are using JUnit to perform unit testing. Therefore, we should import this library. Right click on the project *Build Path* > *Configure Build Path.* On the new window, go to the tab *Libraries*:



Over the *Classpath*, click on "Add Library" and select Junit to add this library to your current project. Now, it should be like this:



And your workspace should be error-free:



Now we can finally start think about the problem and the code.

Unit Testing and JUnit

Unit testing is commonly employed to test small parts of the code before all pieces are available or complete. In Java, we have JUnit which allows us to easily perform unit testing. Now that you have imported your code and configured the JUnit, we can start to work.

The class `Converter` implements several methods to convert miles per hour (mph) to kilometres per hour (kph). The methods are:

`mph2kph`: which receives a double that represents the mph, converts it to kph, and returns another double that represents the kph;

`mph2kph_printing`: which receives a double that represents the mph, converts it to kph, and returns a string "x mph = y kph";

`mph2kph_compare`: which receives two double that represent two distinct mph values, converts them to kph, compares them, and returns a Boolean saying if the first is greater or equal than the second value;

`convert_array`: which receives an array with several mph values, converts all of them to kph, and returns another array with the corresponding kph values.

Each of methods requires a corresponding unit testing method. The `ConverterTest` class implements two unit testing methods for the first two methods described above. Let's check them in detail.

The `test_mph2kph` method tests the first method (`mph2kph`). It uses `assertEquals` to make sure that the outcome of the `mph2kph` method is equal to the expected outcome. Note that the expected value is set by the programmer or, in other words, the programmer is responsible for thinking about a test in which he can calculate the expected result in advance and use that to create the test. Study this assertion (which is actually comparing double variables) and its parameters using the documentation (http://junit.sourceforge.net/javadoc/org/junit/Assert.html). Note that there are several types of assertions. We are going to learn and use some of them in this practical.

The `test_mph2kmh_printing` is testing the second method (`mph2kph_printing`). Technically, it is testing if the string returned by the `mph2kph_printing` method is equal the expected string.

An important thing to note in this unit testing class is that we use of some annotations, such as `@Test` and `@BeforeAll`, to distinguish regular from test methods. These annotations inform the compiler that such methods are, in fact, unit testing methods and not common methods. There are several distinct annotations that can be used for different purposes, as can be seen here: https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/package-summary.html

This is a bit different to anything you've done before! Until now, all Java programs started from the main() method. With JUnit, each of the annotated methods is treated like a separate main method. The program will start at the beginning of test_mph2kph, run the content, and finish. Then a new program will start at the beginning of test_mph2kmh_printing, run the content, and finish. This will repeat for the other methods too. Each test is regarded as passing if no exceptions happened while it was running.

Try to run the test class and observe that a new perspective will open showing the test methods and whether they have passed or failed. For now, all tests should have passed.

Your turn

Now, implement two new tests for the remaining two methods. You may use the `assertEquals` methods. However, there are specific assertions for each of the remaining cases or, in other words, there is one assertion specifically to check whether a Boolean is true or false, and another assertion to compare arrays. Try to search and use these assertions.

[Checkpoint]

Once you have finished implementing and testing your test methods, call over a demonstrator to check your code.

Converting Celsius to Fahrenheit

Let's complement our `Converter` class. First, implement a method that converts Celsius to Fahrenheit using this equation:

```
(C * 9.0/5.0) + 32.0
```

Then, implement another method that converts Pounds to Dollars. You can get the real quotation from the internet.

Finally, implement another method that converts seconds into hours, for example 10,000 seconds = 2.777 hours.

After that, modify the `convert_array` so that this method is capable of converting arrays of mph, Celsius, Pounds, and seconds to arrays of kph, Fahrenheit, Dollars, and hours, respectively.

Create the corresponding test methods for each of these modifications.

[Checkpoint]

Once you have finished implementing and testing, call over a demonstrator to check your code.

Creating a Unit Testing for Practical 1

In practical 1, we used the main method to create prints that would "test" the code for us. This is not the best way of doing this! Create a new unit testing class for practical 1. This class should have **ONE** test method for each of the conditions tested within your original main. In other words, a test method to check if setting the competency is working, another to test if setting years of experience is working, and so on.

[Checkpoint]

Once you have finished implementing the testing, run the class to ensure that all tests will pass. Then, call over a demonstrator to check your code.