Tutorial - JUnit Testing

Working with Testing Tool: JUnit

Software testing is important in verifying if a piece of code is working correctly and as specified in its requirement. As we continue to work together in teams, it is paramount that changes to codes made by team members are tested regularly and consistently. It may contain errors or what we commonly call *bugs*.

JUnit is a simple, open-sourced framework which allows for the writing and running of repeatable test cases.

This tutorial is going to cover the following topics:

- JUnit core concepts
- JUnit with IntelliJ IDE
- JUnit testing with Gradle
- Test coverage reports with JaCoCo

JUnit Core Concepts

Overview of JUnit

JUnit framework - is a test framework that is designed to write and run tests written in the Java programming language.

Type of testing

Unit test

The single most important testing tool is the *unit test*. A *unit test* checks a single method or a set of cooperating methods. In unit testing, you test the classes in isolation rather than the whole completed program. In that way, you will not get confused by the interactions between the classes in a full program. Once you have tested the individual methods and classes, that is when you can put them into your program.

Integration Test

Aims at testing the behaviour of a component or the integration between a set of components. Sometimes, integration testing is also referred to as functional testing. In this type of testing, the

whole system is tested as for how it is intended to work.

Test Coverage

The extent or percentage of code which is tested by unit tests is typically referred to as *test coverage*.

Annotations - Order of execution/Defining test methods

Annotation	Description
import org.junit.jupyter.api.*	Import statement for using the following annotations
@Test	Denotes that a method is a test method. In JUnit5, these annotation does not declare any attributes since test extensions in JUnit Jupiter operate based on their own dedicated annotations.
@BeforeEach	Executed before each test. Used to prepare test environment such read input data, initialise class.
@AfterEach	Executed after each test.
@BeforeAll	Executed once before the start of all tests.
@AfterAll	Executed once after all tests are finished.
@Disabled	Denotes that the test should be disabled.

A JUnit test uses the annotation *@Test* to mark methods as test methods. Other annotations can be used to configure how or when tests methods are run such as *@BeforeEach*, *@AfterEach* or *@Disable* as mentioned above. Below is an example of how the *@Test* annotation is used in a sample JUnit test case.

Writing Tests

To begin writing a JUnit test case, start by importing the following libraries as shown.

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;
```

In an IDE like IntelliJ, this will prompt an error: You will need to add JUnit to the test path. Since this is not a gradle project yet, goto the line with the imports (which should be grey in colour with junit highlighted in red) and click on the light bulb icon to import junit in the class path. Next, use the annotation *@Test* to identify that this particular method is to be defined as a test method.

```
public class MyTests {

    @Test
    void myTest1() {
    ...
```

```
    }
}
```

In JUnit, you can use the *Assert* class to test certain conditions. These *Assert* statements usually start with *assert*. It allows you to specify an error message, the expected and actual result. An assertion method compares the actual value returned by a test to the expected value. It throws an *AssertException* if the comparison fails.

```
    void myTest1() {

        assertEquals(2, 1 + 1);
        //assertEquals(expected, actual[, message])

    }
```


In the above *Assert* statement, *assertEquals* test that float or double values match. The tolerance refers to the number of decimals which must be the same.


```
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

public class MyTests {

    @Test
    void myTest1() {

        assertEquals(2, 1 + 1);

    }
}
```

Try this exercise out.

Assertions

The example above shows how JUnit testing methods utilise the *assertions*, which are statements that check whether a given condition is true or false. Therefore, a test is considered a failure if the condition is false. Likewise, a test is a success if the conditions in the test methods are true. There are many more *assertion* methods that can be used to test for different conditions of the program. Below is the list of *assertions* methods that you can use.

Method name	**Description**
assertTrue (test, "message")	Checks that boolean condition is true. The method fails if the condition is not true.
assertFalse (test,	Checks that boolean condition is false. The method fails if the

"message")	condition is not false.
assertEquals (expectedValue, value, "message")	Tests that the expectedValue is equal to value. It can also test that float or double values are equal. The method fails if two values are not equal to each other.
assertNotEquals (value1, value2, "message")	Tests that the two values are not equal to each other. The method fails if values are equal to each other.
assertNull (value, "message")	Tests if given value is null. The method fails if the value is not null.
assertNotNull (value, "message")	Tests if given value is not null. The method fails if the value is null.
assertSame (expectedValue, value, "message")	Identical to assertEquals and assertNotEquals. For objects, it checks the two objects using the '==' operator. The method fails if values are not the same.
assertNotSame (value1, value2,"message")	Tests if given value is not the same. The method fails if values are the same.
fail ("message")	Let method fail

JUnit testing with IDE

JUnit is one of the ways for program developers to run multiple test cases efficiently to test the validity of Java programs. Using IntelliJ IDE, this process can even be done automatically.

Exercise:

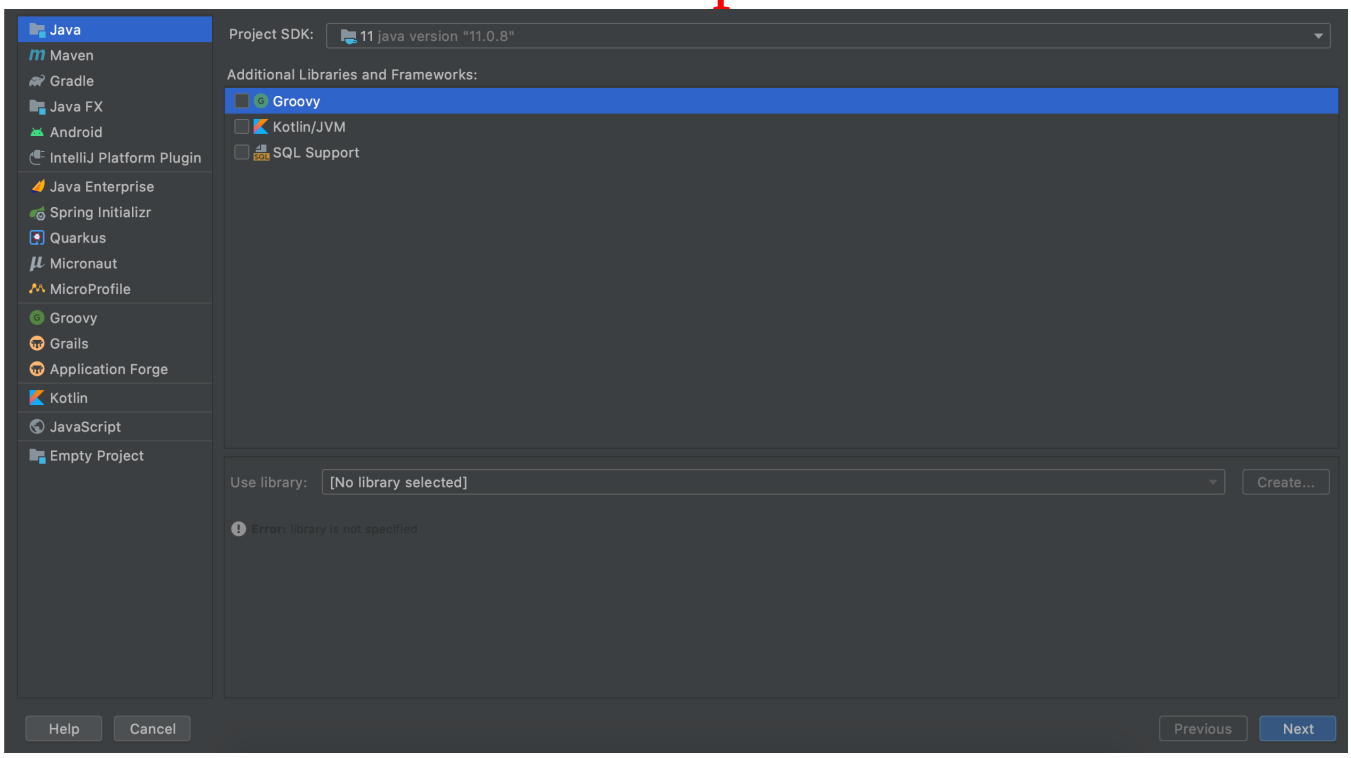
The example above illustrates the use of one of the assertions methods to test your programming codes. In the next exercise, create new test cases using the following assertions methods. An example has been done for you. Continue building the test cases from this sample code.

```
package main;

import java.util.Arrays;
import java.util.List;

public class JUnitDemo {

    public int add (int a, int b) {
         return a + b;
    }

    public int subtract (int a, int b) {
         return a - b;
    }

    public int multiply (int a, int b) {
         return a * b;
    }
```

```
    public int division (int a, int b) {
        return a/b;
    }

    public int square (int a) {
        return a * a;
    }

    public boolean isOddNumber (int a) {
        if (a%2 == 1) {
            return true;
        } else {
            return false;
        }
    }

    public String setText (String a) {
        return a;
    }

    public List<String> setList (String a, String b, String c) {
        List<String> arrayList = Arrays.asList(a, b, c);
        return arrayList;
    }
}
```

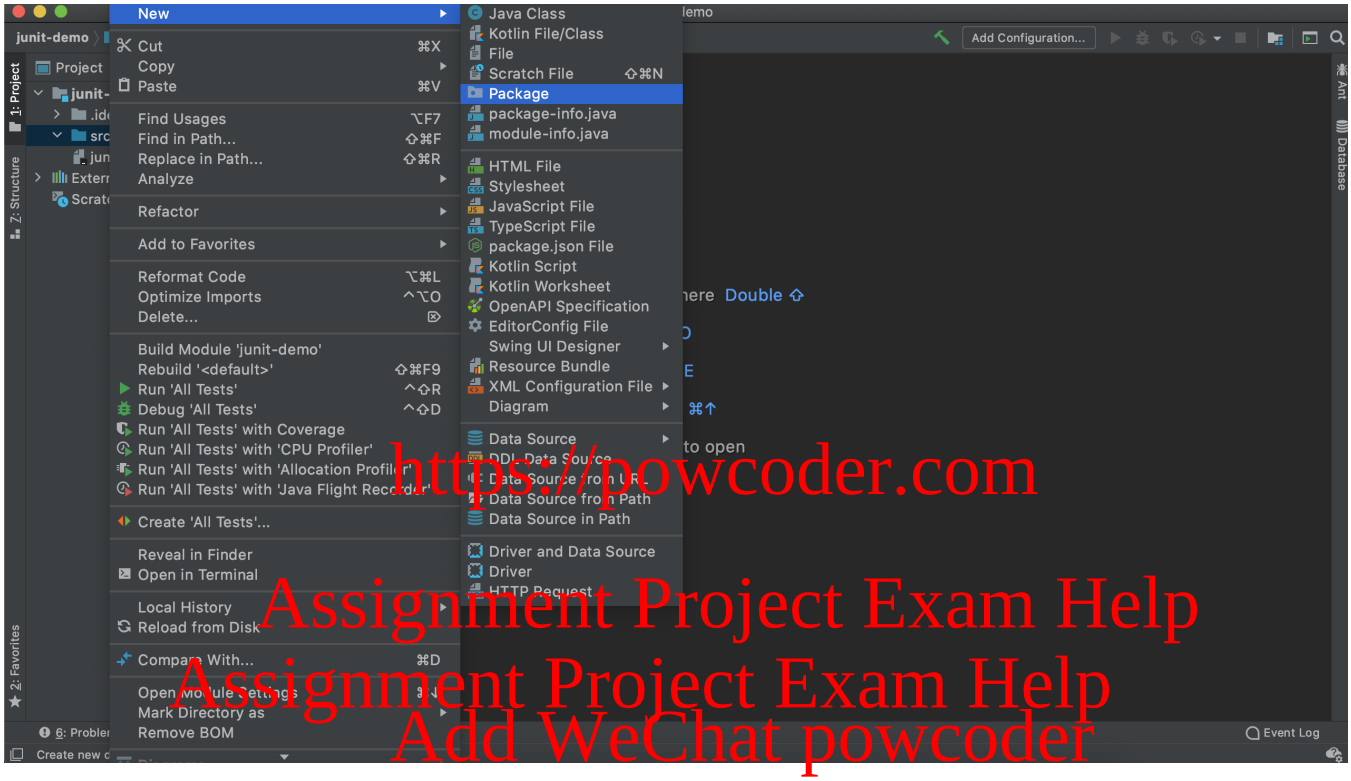

Follow the steps below to begin the exercise.

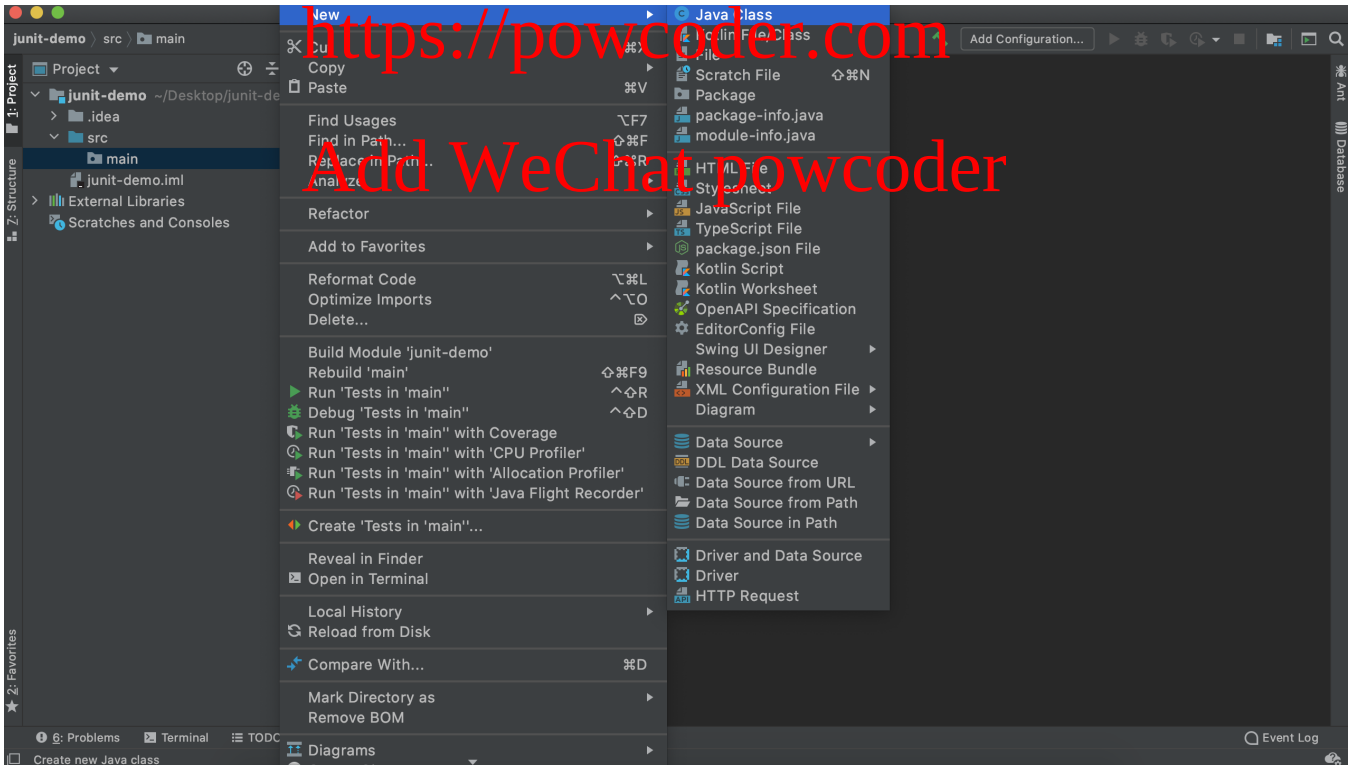
- Create a new Java Project in IntelliJ.

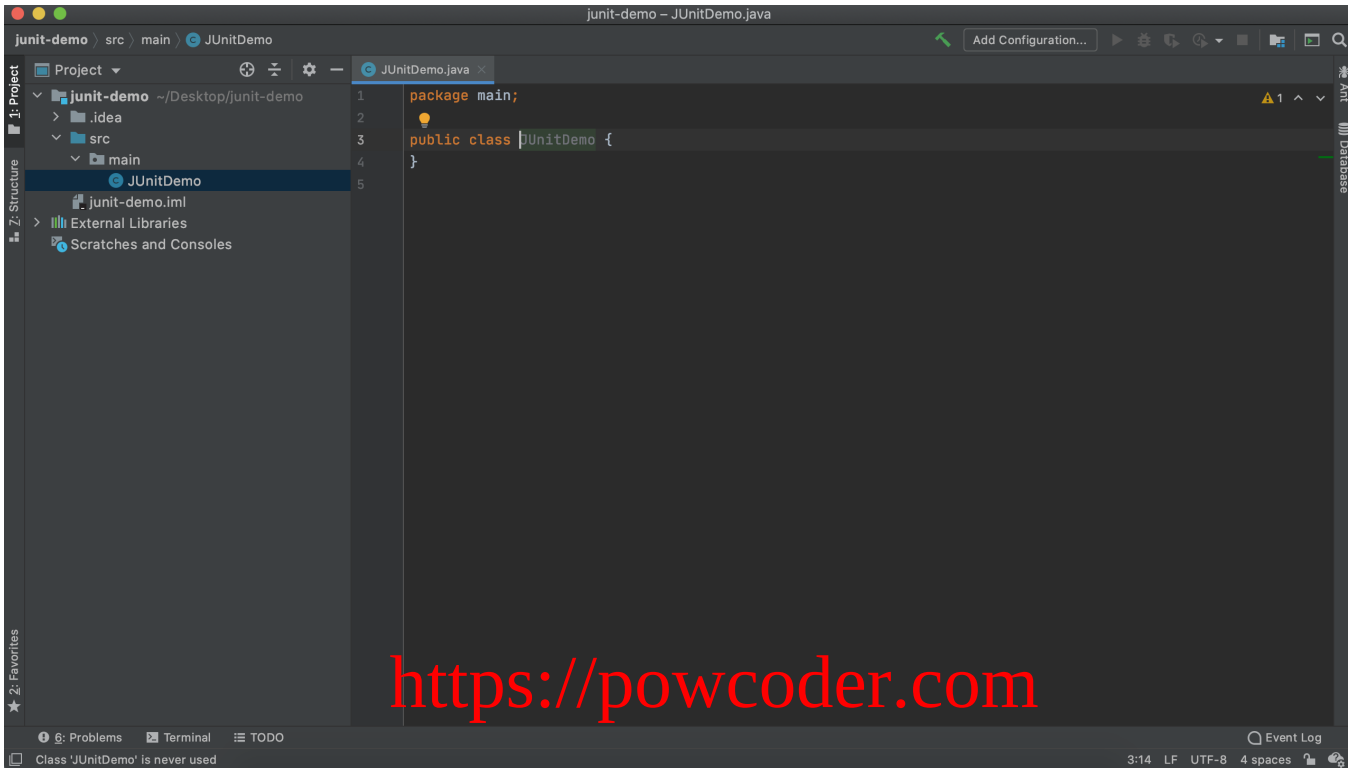


- Keep going next until it prompts you to choose a project name.
- Name this project *junit-demo*.

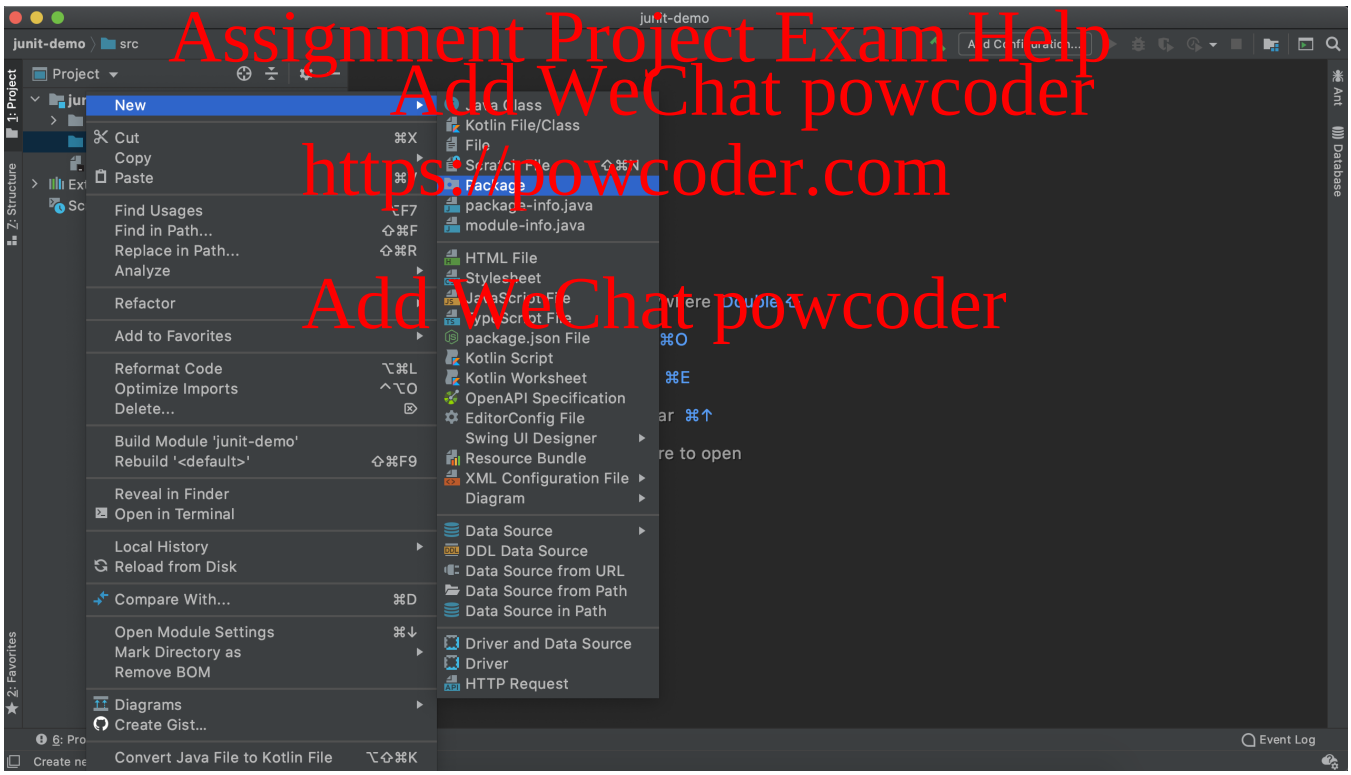
- Click Finish to continue
- Create a folder in src and call it "main". Add the above calculator code in a file called *JUnitDemo.java*.



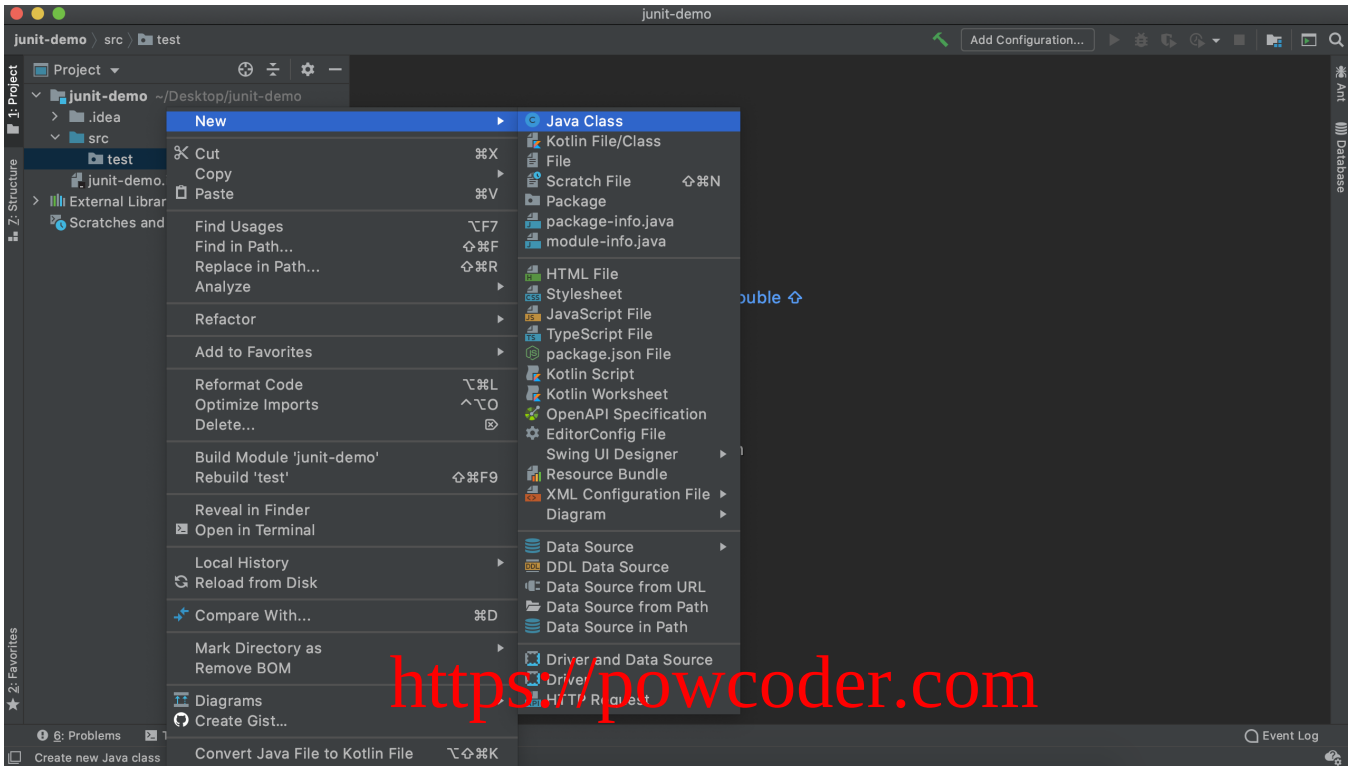




- Now, Create a new folder under src and call it *test*.



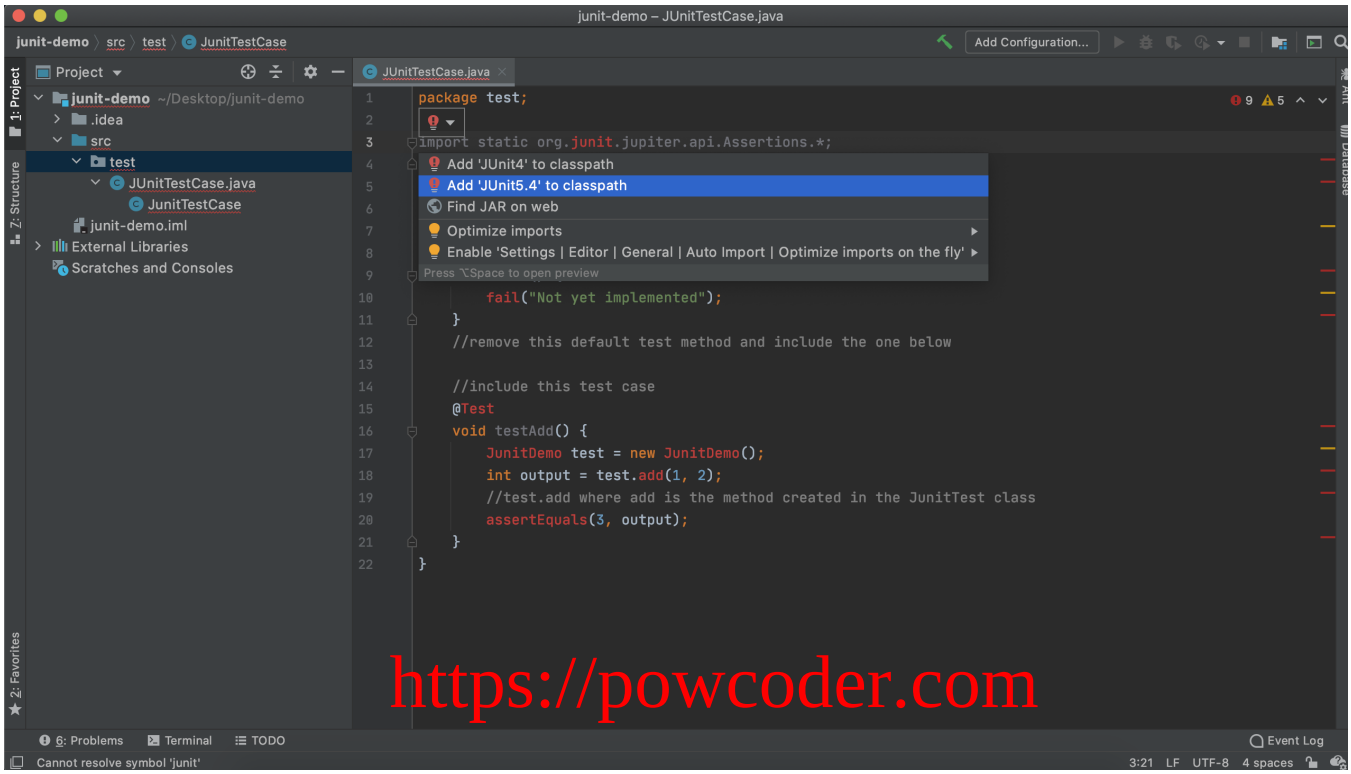
- Create a test class called *JUnitTestCase.java*.



- Add the below code in that file


```
package test;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

class JUnitTestCase {

    @Test
    void test() {
        fail("Not yet implemented");
    }
    //remove this default test method and include the one below
    //the above is how to fail a test. remove the code to have your tests passing

    //include this below test case
    @Test
    void testAdd() {
        JUnitDemo test = new JUnitDemo();
        int output = test.add(1, 2);
        //test.add where add is the method created in the JUnitDemo class
        assertEquals(3, output);
    }
}
```

- You will see many red lines. Hover over one of them until the red light bulb is seen. Click on the light bulb and add Junit 5 to your class path.

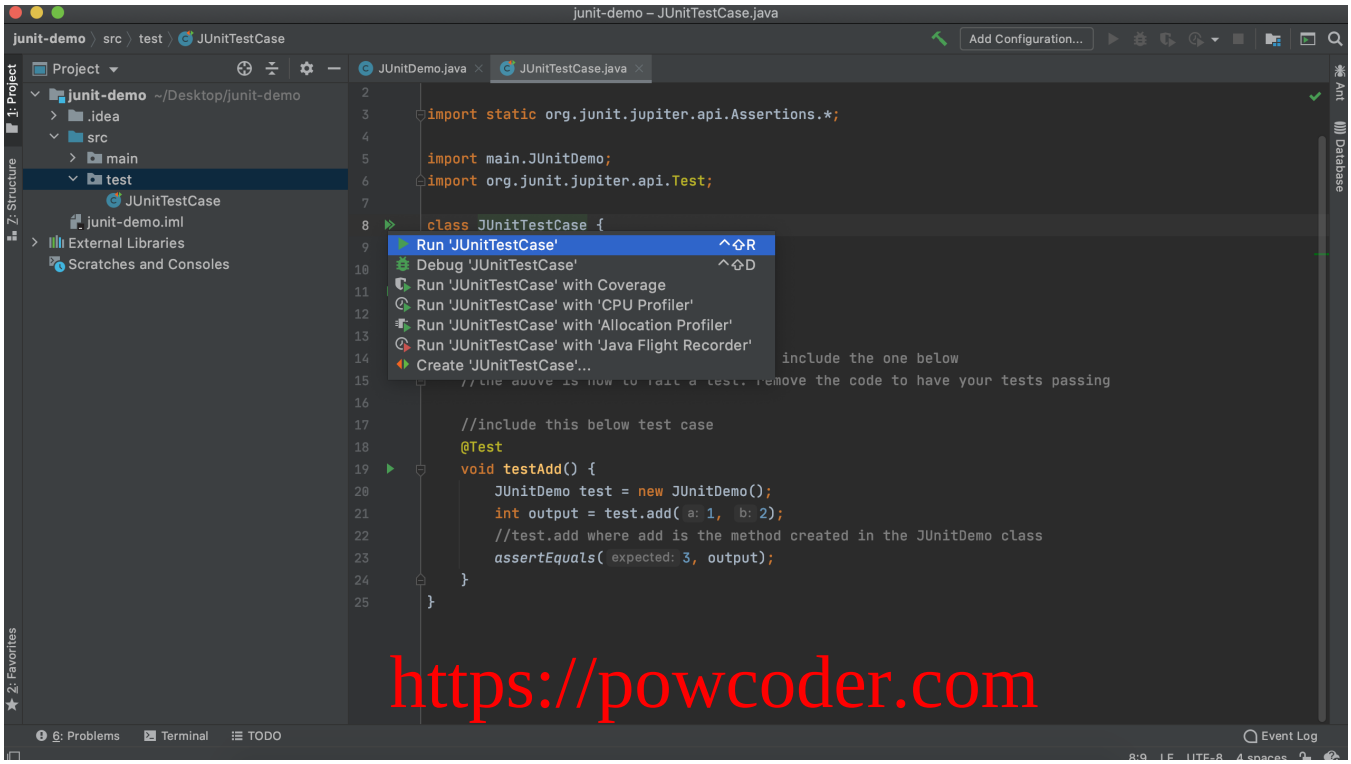


- Remove the default *void test()* method and replace it with the *void testAdd()* method above.

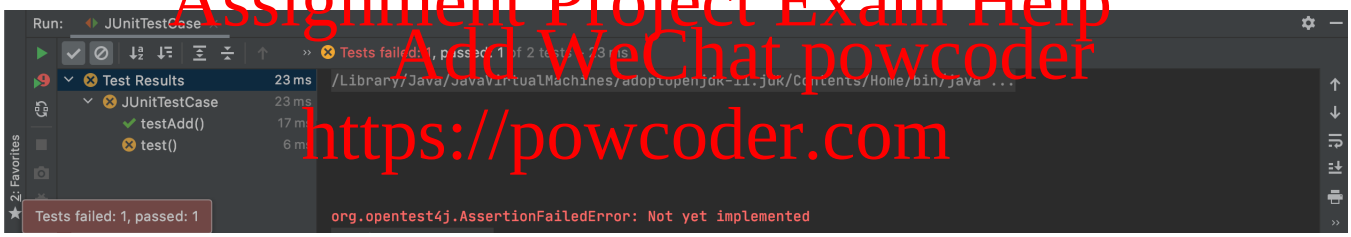
- This example makes use of the *assertEquals* method to check if the add method defined in *JunitTest* class is working properly. The output is the outcome of invoking the add method. This output value is then checked against the expected value to test if it is equal which is what is expected.

- Your next task is to continue adding test cases to the *JunitTestCase.java* file using the following assertions methods:

- *assertFalse*
- *assertEquals*
- *assertNull*
- *assertNotNull*
- *assertSame*
- *assertNotSame*
- You should design a test case that tests all the methods defined in *JUnitDemo.java*. You must include the use of **all** the assertions methods **at least once** in the *JUnitTestCase.java*.
- Finally, run your test cases. This can be done through clicking on the green arrow button besides the test class.

- 

- You will see a failed test (due to the method *test()* in the test class. You must remove that to pass tests.

- 

JUnit testing with Gradle

JUnit testing can also be done if you build your Java project purely using Gradle without the IntelliJ IDE. The JUnit 5 Gradle plugin allows you to create test cases and run them. There are a few steps that you need to perform to configure your Gradle dependencies before being able to start testing. You will be using a command prompt or Windows terminal to manually run the gradle commands.

Create a gradle application through gradle init with gradle version 5.6+. Make sure while doing so, you choose JUnit Jupyter as the test library.

Add the above code and run the application using *gradle build run.*

Writing Test Cases

Before you begin to write test cases, you need to create the following directory to store your test case codes.

- Create the following folder *src/test/java*.

- Add your *JUnitTestCase.java* file into the folder above.
- Next, add the *JUnitTest.java* file we defined in the earlier exercise into *src/main/java* folder.

Running JUnit tests

Now that we have all our java program and the test codes ready, we can run the JUnit test using Gradle.

To run the test command in gradle, key in the following: *gradle clean test*

Check from the gradle output report to see if the test cases are successful or there is a failure.

Test coverage reports with JaCoCo

To determine the percentage of code covered in your tests, we will be using a simple tool called "jacoco" that can be integrated with gradle. This tool also provides an interface for viewing the percentage of tests passed. You can choose to view these results in html, csv or xml formats, and is configurable.

Below is described how to add the plugin:

In your build.gradle under the *plugins* block, add the following line:

```
id: 'jacoco'
```


You can enable the page type for viewing these results. By default, an html format is stored. However you may add more formats. Edit the defined task called jacocoTestReport as follows in your build.gradle:

```
jacocoTestReport {
    reports {
        html.enabled = true
        csv.enabled = true
    }
}
```

By default, the reports are stored in the following file path, once you run a gradle build : YOURBUILDDIR/reports

This can be changed, by adding the following lines in the report section of the jacoTestReport part of the build script:

```
html.destination file("${buildDir}/<custom_directory>")
```

Now, to generate the report, run the following commands:

```
gradle clean build (this command will initialise the directory for storing reports)
gradle test jacocoTestReport (this command will generate reports for both test coverage and c
ode coverage in the reports/ directory)
```

To automatically generate a test report with every "gradle test" command execution, add the following in the build script:

```
test {
    useJUnitPlatform()
    test.finalizedBy jacocoTestReport // <- add this line
}
```

This will be useful for your assignment. For more details, refer to this link: [https://docs.gradle.org/current/userguide/jacoco_plugin.html (https://docs.gradle.org/current/userguide/jacoco_plugin.html)](https://docs.gradle.org/current/userguide/jacoco_plugin.html)

Tasks To Do:

This task is aimed to help you be more fluent with JUnit testing, and gradle as well. Try to finish this help you in your assignment.

- You will code a Calculator program that adds, subtracts, multiplies and divides two numbers. Yo you may write a parser for it if you can.
- Write a class called Calculator.java, with these four functions: add, subtract, divide and multiply.
- Write a Test class called CalculatorTest.java, and write JUnit test cases for the Calculator.java c writing tests.

- Write the gradle build script, you may take help from the above sections.
- Using Jacoco, get the test coverage. Is your test coverage good? Show your work to the tutor.

Additional Resources

- [https://junit.org/junit5/docs/current/user-guide/#writing-tests (https://junit.org/junit5/docs/current/user-guide/#writing-tests)](https://junit.org/junit5/docs/current/user-guide/#writing-tests)
- [http://www.vogella.com/tutorials/JUnit/article.html#testingterminology (http://www.vogella.com/tutorials/JUnit/article.html#testingterminology)](http://www.vogella.com/tutorials/JUnit/article.html#testingterminology) [(https://courses.cs.washington.edu/courses/cse143/11wi/eclipse-tutorial/junit.shtml)](https://courses.cs.washington.edu/courses/cse143/11wi/eclipse-tutorial/junit.shtml)