Lab: Unit Testing

Introduction

For this lab, you will be working on a calculator application. The application itself has already been written, but the unit testing is incomplete.

The calculator only works with string parameters. For example, to add the numbers 100 and 200, you would pass the string '100,200' to the Add method. The calculation methods all return integers.

Starter projects

There is a starter project for this lab, available in C#, Java, and JavaScript versions.

Goals

Your job is to complete the unit testing of the calculator class.

1. Write unit test(s) to demonstrate the correctness of each of these methods:
   - Subtract.
   - Multiply.
   - Divide (including what you expect to happen if you ask for a division by zero).
2. You should aim to write at least five (5) separate unit tests.
3. Try to include tests that show how the calculator methods deal with good input in various formats, as well as input which is badly formatted in various ways.

Rules

1. All tests must pass (without commenting any of them out).
2. Do not update the functions to make the tests pass – the functions should be right. (If a test fails, it probably indicates a problem with the test itself).
3. Remember to keep your git repository updated.

Think about:

- Different types of valid data
- Different types of invalid data
- Boundary conditions

Java notes

The Java starter project uses the **JUnit** framework.

C# notes

The C# starter project uses the **NUnit** framework.

After you open the solution, you may need to right-click on the solution in solution explorer, then click **Restore NuGet packages**. After that, right-click solution and select **Clean build**, then right click solution and select **Rebuild**. After that, you should be able to run your tests OK.

JavaScript notes

The JavaScript starter project uses the **chai** and **mocha** unit testing frameworks.

After opening the JavaScript project, do:

```
npm install
```

To run your tests, do:

```
npm test
```

JavaScript developers

To create your own project from scratch, open Visual studio code and then follow these steps:

1- Create a directory and open it in vs-code

2- Press **Ctrl-'**

This will open a command window

3- type **`npm init`**
Press enter for all questions. You can fill these later.,

4- Open **package.json**

5- Edit the line with "**main**" entry to
"**main": "calculator.js**",

also change this part:

```
"scripts": {
      "test": "mocha"
},
```

6- Create a file called calculator.js

```
class Calculator {
        constructor() {

        add(a, b) {
                return a + b;
        }
        mult(a, b) {
                return a * b;
        }
}

module.exports = Calculator;
```

7- Install the required mocha and chai libraries:
`npm install mocha chai --save-dev`

Note: The **`--save-dev`** will install these only for development purpose and they are not installed for production.
View the file **package.json** and observe the entries for mocha and chai at the bottom of the file.

8- Create a folder called "**test**"
By default, mocha will look for this folder. This is also a good practice to separate your code for testing into a folder.

9- Create a file called calculatorTest.js in the "test" folder and Write:

```
const chai = require('chai');
const assert = require('chai').assert;
const expect = chai.expect;
const Calculator = require('../calculator');

describe('calculator tests', function() {
    it('calculator returns the right value for valid parameters',
        function() {
            const calc = new Calculator();
            const result = calc.add(10, 22);
            expect(result).to.be.equal(32)
        });
});
```

10- Run your test by typing: **`npm test`**

Lab Readable Tests

Introduction

A set of unit tests have already been written for you, but they could be better – in particular, they could be more **readable**.

Starter projects

There is a starter project for this lab, available in C#, Java and JavaScript versions.

Goals

Your job is to analyse the provided unit tests, and think of ways to make the tests more readable.

Apply your improvements to the tests, then make sure they all still pass when run.

Think about:

- Do the names of each test follow a consistent, useful naming convention?
- Is there any repeated code? How could you remove the duplication yet still have the tests function the same as before?
- How many things are being tested in each test? If there is more than one assert statement, is that acceptable?
- Does each test clean up well enough after itself?

Java notes

You can define a method that runs before every test using this code:

```
@Before
public void setUp() {
  // Code goes here...
}
```

You can define a method that runs after every test using this code:

```
@After
public void tearDown() {
  // Code goes here...
}
```

C# notes

You can define a method that runs before every test using this code:

```
[SetUp]
public void SetUp()
{
  // Code goes here...
}
```

You can define a method that runs after every test using this code:

```
[TearDown]
public void TearDown()
{
  // Code goes here...
}
```

JavaScript notes

You can define a method that runs before every test using this code:

```
beforeEach(() => {
  // Code goes here...
});
```

You can define a method that runs after every test using this code:

```
afterEach(() => {
  // Code goes here...
});
```