# ICT3101 Lab 1

The laboratories, for the first half of this module will be focusing on practicing what was covered in lectures. Primarily, this practice will allow familiarisation with skills and absorption of the knowledge surrounding software testing.

Although laboratory material does not need to be submitted, its content will be assessed via the weekly quizzes (RATs). You may work together with your teammates or separately for the lab exercises.

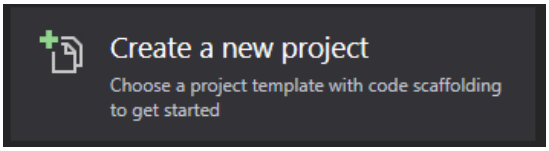
The main focus of this laboratory is to understand and practice Unit Testing. Note also, that this will be somewhat of a revision exercise from ICT2101 where tests were briefly introduced. **Please ensure you have Visual Studio 2019 installed** with the ability to create **.Net Core applications**. I.E., when you install VS ensure that you select the checkbox for .Net Core development, as we'll be using it.

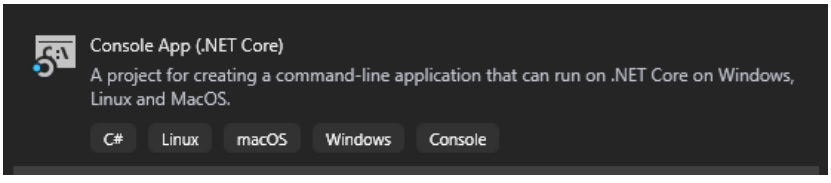
(VS 2017 should work fine, but VS 2019 is recommended—note that VS Code seems to have most of the packages we need, but it's not been tested for the labs. Hence, use at your own peril.)

Begin by setting up a .Net Core project named "ICT3101_Calculator". We are starting by using some basic, and common code to allow us to add in Unit tests and understand how they work.

Steps:

## Setup

1. Open Visual Studio 2019
2. 

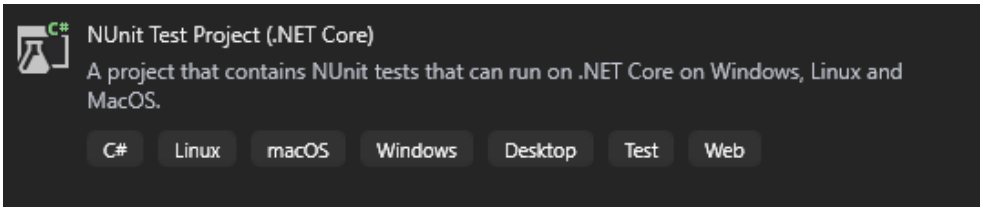
3. 

4. Now that you've create your project you can open the default class "Program" and copy/paste the "Main" code.
5. Next, you'll need to add a new class (by right clicking on the project, then "add"). Name this class "Calculator".
6. Copy/Paste the Calculator code into this new class.
7. At this point you should be able to run this simple console calculator by pressing the **F5 key**.

## Adding the Units

8. Now that we have our basic application, we plan to make enhancements to it. However, we want to ensure that we maintain functionality—that we don't break anything. With this goal in mind, we're going to setup some Unit tests. The convention for adding in Unit Tests is to create a new project within your Solution "3101_Calculator"—right click on the solution and select "Add" then "New Project".

   For our Unit tests we're going to be using the most well established NUnit. In the "Add a new project" menu select a new NUnit Test Project (.Net Core):




9. Name this project "ICT3101_Calculator.UnitTests".

10. In this new project we're given a default class called "UnitTest1", right-click to rename this to "CalculatorTests".

11. Now, copy/paste the "Calculator Unit tests code" into this class.

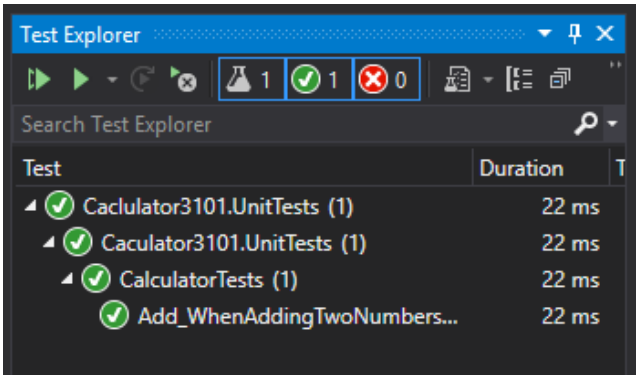
12. As you will notice in the Unit test, we follow the naming convention of "MethodNameWe'reTesting_ScenarioWe'reTesting_ExpectedBehaviourOrResult". We also use the NUnit Prefix of "[Test]" to denote a Unit test.

```
[Test]
public void Add_WhenAddingTwoNumbers_ResultEqualToSum()
```

    Also note we're using the "[SetUp]" prefix. This causes the following function to get called before any of the test in this class executes—the convention is to name this method "Setup".

```
[SetUp]
public void Setup()
```

13. Now to check that this runs okay. Open "Test Explorer" (Test-Windows-Test Explorer) or just right-click on your UnitTests project, and the select "Run Tests". You should get something like the following.



# Unit tests: basic lessons

## Lesson 1: Characteristics of a good unit test

- **Fast**. It is not uncommon for mature projects to have thousands of unit tests. Unit tests should take very little time to run. Milliseconds.
- **Isolated**. Unit tests are standalone, can be run in isolation, and have no dependencies on any outside factors such as a file system or database.
- **Repeatable**. Running a Unit test should be consistent with its results, that is, it always returns the same result if you do not change anything in between runs.
- **Self-Checking**. The test should be able to automatically detect if it passed or failed without any human interaction.
- **Timely**. A Unit test should not take a disproportionately long time to write compared to the code being tested. If you find testing the code taking a large amount of time compared to writing the code, consider a design that is more testable.

## Lesson 2: Best practices: Naming your tests

The name of your test should consist of three parts:

- The name of the method being tested.
- The scenario under which it's being tested.
- The expected behavior when the scenario is invoked.

### Why?

- Naming standards are important because they explicitly express intent of a test in a way that is commonly understood.

## Lesson 3: Arranging your tests: "Arrange, Act, Assert" is a standard pattern when writing Unit tests. As this name implies, it consists of three main actions:

- ***Arrange*** your objects, creating and setting them up as necessary.
- ***Act*** on an object.
- ***Assert*** that something is as expected.

### Why?

- Clearly separates what is being tested from the ***arrange*** and ***assert*** steps.
- Less chance to intermix assertions with "**Act**" code.

### Example:

```
[Test]
//(We want our code to return 0 when an empty string is added.)
public void Add_EmptyString_ReturnsZero()
{
    // Arrange
    var stringCalculator = new StringCalculator();

    // Act
    var actual = stringCalculator.Add("");

    // Assert
    Assert.Equal(0, actual);
}
```

## Exploring with Units

14. Add Unit tests to cover all the methods: Add, Subtract, Multiply and Divide.

    a. Why not try out some edge cases—don't use max int/double though…unless you've a lot of time to spare. E.g., what happens when you pass in zeros—specifically into the divide function?

    b. Use the results from your new tests to refactor your methods. How does this change the existing functions/does it?

***Note:*** *remember that we're writing tests to ensure correct behaviour, and we don't need to deal with test cases that are already dealt with, E.g., passing a String into a function that takes an Integer is already accounted for.*

15. Let's assume that you now wish for your Divide function to throw an exception should there be a 0 input for either value.

    a. Given the following Unit test, edit your Division function to handle this.

```
[Test]
public void Divide_WithNegativeInputs_ResultThrowArgumentException()
{
   Assert.That(() => _calculator.Divide (0, 0),Throws.ArgumentException);
 }
```

    b. Furthermore, you want to handle two other variations in this test. This can be done using a "Parameterized test". With NUnit we use the following notation. Add this to your test and alter accordingly. Ensure that the correct exception is thrown in each case.

```
[Test]
[TestCase(0, 0)]
[TestCase(0, 10)]
[TestCase(10, 0)]
public   void   Divide_WithZerosAsInputs_ResultThrowArgumentException(int   a,   int   b)
…
```

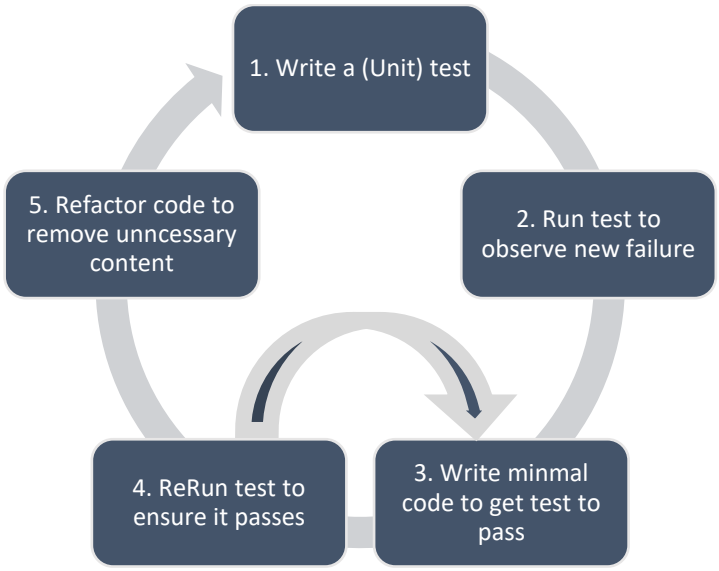
16. Now add a new function for calculating a Factorial and write the corresponding Unit tests. (The Whitebox Testing lectures have an example of this function that you can use—if you're struggling.)

    a. How many tests do you think are necessary here—how do you decide? Ensure you throw ArgumentExceptions for negative inputs.

    b. Add this into the application as an option ('f' for factorial of first number entered).

# What is TDD?

Test Driven Development (TDD) is a methodology of software development that was created to increase the quality of software systems. The core idea is to write tests before writing the program code. This section provides a breakdown of the key principles.




## TDD Principles

Is a software development process: not a testing technique. However, it's reliant on testing tools.

1. **Write tests first**
    a. Tests determine what code is to be written.
    b. Testing is done in a fine-grained fashion.

2. **Run tests to see the failure**
    a. This can inform you of the difference between the expected and actual result. Hence, allowing for a better understanding of any adjustments needed.

3. **Write minimal code that get the test to pass**—making more future-proof changes only if they come to mind.

4. **Rerun the tests to ensure the changes are effective**
    a. If the test fails again, then continue to make minimal changes until it passes.
    b. Else the test passes, then move on to refactoring the code.

5. **Refactor**: I.E., Rewrite already-working code to:
    - Eliminate duplicate code.
    - Simplify testing.
    - Improve future use (and understanding ideally) of code.
    - Follow accepted software engineering principles.

**Summary**: Always start with a failing test. Quickly write the simplest code needed to pass the test. Refactor and repeat as needed—to meet requirements. Test everything that could possibly break.

## Practicing TDD

17. You now plan to use another testing practice, involving Unit Testing, this is TDD (as above). Tests are developed first, they fail, and then the code is refactored to pass these tests.

    a. Write a function for calculating the area of a triangle using TDD (given height and length). **Test cases first**.

    b. Write a function for calculating the area of a circle (given only the radius as the first argument). Again, **test cases first**.

18. The following Unit tests are for functions that use your Factorial, Division, Subtraction, and Multiplication functions. Given the sets of Unit tests for 'a.' and 'b.', write the corresponding function code for each set that passes all tests. What are the following functions—what are they called?

    *[Hint a. involves the following function uses: 2 Factorials, 1 Divide, and 1 Subtract.]*

    a. UnknownFunctionA's Unit tests:

```
[Test]
public void UnknownFunctionA_WhenGivenTest0_Result()
{
    // Act
    double result = _calculator.UnknownFunctionA(5, 5);
    // Assert
    Assert.That(result, Is.EqualTo(120));
}

[Test]
public void UnknownFunctionA_WhenGivenTest1_Result()
{
    // Act
    double result = _calculator.UnknownFunctionA(5, 4);
    // Assert
    Assert.That(result, Is.EqualTo(120));
}

[Test]
public void UnknownFunctionA_WhenGivenTest2_Result()
{
    // Act
    double result = _calculator.UnknownFunctionA(5, 3);
    // Assert
    Assert.That(result, Is.EqualTo(60));
}

[Test]
public void UnknownFunctionA_WhenGivenTest3_ResultThrowArgumnetException()
{
    // Act
    // Assert
    Assert.That(() => _calculator.UnknownFunctionA(-4, 5), Throws.ArgumentException);
}

[Test]
public void UnknownFunctionA_WhenGivenTest4_ResultThrowArgumnetException()
{
    // Act
    // Assert
    Assert.That(() => _calculator.UnknownFunctionA(4, 5), Throws.ArgumentException);
}
```

*[Hint b. involves the following function uses: 3 Factorials, 1 Divide, 1 Multiply, and 1 Subtract.]*

b. UnknownFunctionB's Unit tests:

```
[Test]
public void UnknownFunctionB_WhenGivenTest0_Result()
{
    // Act
    double result = _calculator.UnknownFunctionB(5, 5);
    // Assert
    Assert.That(result, Is.EqualTo(1));
}

[Test]
public void UnknownFunctionB_WhenGivenTest1_Result()
{
    // Act
    double result = _calculator.UnknownFunctionB(5, 4);
    // Assert
    Assert.That(result, Is.EqualTo(5));
}

[Test]
public void UnknownFunctionB_WhenGivenTest2_Result()
{
    // Act
    double result = _calculator.UnknownFunctionB(5, 3);
    // Assert
    Assert.That(result, Is.EqualTo(10));
}

[Test]
public void UnknownFunctionB_WhenGivenTest3_ResultThrowArgumnetException()
{
    // Act
    // Assert
    Assert.That(() => _calculator.UnknownFunctionB(-4,5), Throws.ArgumentException);
}

[Test]
public void UnknownFunctionB_WhenGivenTest4_ResultThrowArgumnetException()
{
    // Act
    // Assert
    Assert.That(() => _calculator.UnknownFunctionB(4,5), Throws.ArgumentException);
 }
```

*—The end of questions—*

CODE APPENDICES

## "Main" code

```
static void Main(string[] args)
        {
            bool endApp = false;
            Calculator _calculator = new Calculator();
            // Display title as the C# console calculator app.
            Console.WriteLine("Console Calculator in C#\r");
            Console.WriteLine("------------------------\n");

            while (!endApp)
            {
                // Declare variables and set to empty.
                string numInput1 = "";
                string numInput2 = "";
                double result = 0;

                // Ask the user to type the first number.
                Console.Write("Type a number, and then press Enter: ");
                numInput1 = Console.ReadLine();

                double cleanNum1 = 0;
                while (!double.TryParse(numInput1, out cleanNum1))
                {
                    Console.Write("This is not valid input. Please enter an integer value: ");
                    numInput1 = Console.ReadLine();
                }

                // Ask the user to type the second number.
                Console.Write("Type another number, and then press Enter: ");
                numInput2 = Console.ReadLine();

                double cleanNum2 = 0;
                while (!double.TryParse(numInput2, out cleanNum2))
                {
                    Console.Write("This is not valid input. Please enter an integer value: ");
                    numInput2 = Console.ReadLine();
                }

                // Ask the user to choose an operator.
                Console.WriteLine("Choose an operator from the following list:");
                Console.WriteLine("\ta - Add");
                Console.WriteLine("\ts - Subtract");
                Console.WriteLine("\tm - Multiply");
                Console.WriteLine("\td - Divide");
                Console.Write("Your option? ");

                string op = Console.ReadLine();

                try
                {
                    result = _calculator.DoOperation(cleanNum1, cleanNum2, op);
                    if (double.IsNaN(result))
                    {
                        Console.WriteLine("This operation will result in a mathematical error.\n");
                    }
                    else Console.WriteLine("Your result: {0:0.##}\n", result);
                }
                catch (Exception e)
                {
                    Console.WriteLine("Oh no! An exception occurred trying math.\n - Details: " + e.Message);
                }

                Console.WriteLine("------------------------\n");

                // Wait for the user to respond before closing.
                Console.Write("Press 'q' and Enter to quit the app, or press any other key and Enter to continue: ");
                if (Console.ReadLine() == "q") endApp = true;

                Console.WriteLine("\n"); // Friendly linespacing.
            }
            return;
        }
```

## Calculator code

```
public class Calculator
    {
        public Calculator() { }
        public double DoOperation(double num1, double num2, string op)
        {
            double result = double.NaN; // Default value
            // Use a switch statement to do the math.
            switch (op)
            {
                case "a":
                    result = Add(num1, num2);
                    break;
                case "s":
                    result = Subtract(num1, num2);
                    break;
                case "m":
                    result = Multiply(num1, num2);
                    break;
                case "d":
                    // Ask the user to enter a non-zero divisor.
                    result = Divide(num1, num2);
                    break;
                // Return text for an incorrect option entry.
                default:
                    break;
            }
            return result;
        }

        public double Add(double num1, double num2)
        {
            return (num1 + num2);
        }

        public double Subtract(double num1, double num2)
        {
            return (num1 - num2);
        }

        public double Multiply(double num1, double num2)
        {
            return (num1 * num2);
        }

        public double Divide(double num1, double num2)
        {
            return (num1 / num2);
        }

    }
```

## Calculator Unit test code

```
namespace ICT3101_Caculator.UnitTests
{
    public class CalculatorTests
    {
        private Calculator _calculator;

    [SetUp]
        public void Setup()
        {
            // Arrange
            _calculator = new Calculator();
        }

        [Test]
        public void Add_WhenAddingTwoNumbers_ResultEqualToSum()
        {
            // Act
            double result = _calculator.Add(10, 20);
            // Assert
            Assert.That(result, Is.EqualTo(30));
        }
    }
}
```

## NOTES:

Basic principles of a Unit Test is based on: https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices
Basic Calculator code is based on: https://docs.microsoft.com/en-us/visualstudio/get-started/csharp/tutorial-console?view=vs-2019