# Lab: Unit Testing Methods

Tasks for exercise in class and for homework to the course  ["Programming Fundamentals and Unit Testing" @ SoftUni](https://softuni.bg/trainings/4256/programming-fundamentals-and-unit-testing-september-2023).

# Unit Test: Calculator

Look at the **provided skeleton** and examine the Calculate class that we are testing:

A screenshot of a computer

Description automatically generated 🡪

A screenshot of a computer code

Description automatically generated

Then look at the **unit tests** for the above class:

A screenshot of a computer

Description automatically generated

Look around the CalculateTests **class** to see how **unit tests are written**. This is how our unit tests look like:

A computer screen shot of a program

Description automatically generated

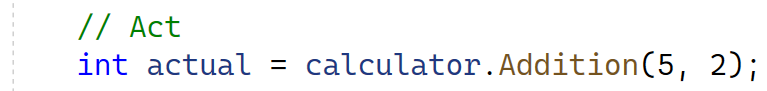
Here is how the Test\_Addition() test works.

First, we **create an instance** of the class we are **going to test.** This is part of our "Arrange" section:

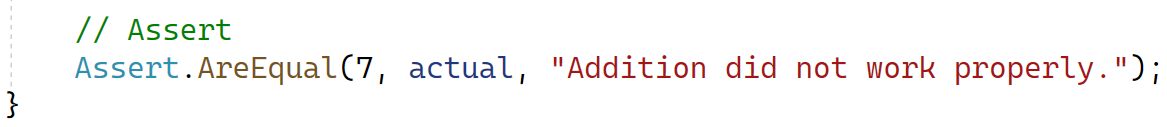
A white background with black text

Description automatically generated

Next, as part of our "Act" section we call the Addition() method from our **instance** passing in the **required arguments** and **storing the result** in an integer.



At the end for our "Assert" section we are going to call Assert.AreEqual() and pass in an **expected value** we want to see, the **actual value** that the method returned and a **failure message** in case the test does not pass.



Now try to **create your own unit test** in the same class following the **existing example** to test the Subtraction() **method** from the same class.

A screenshot of a computer program

Description automatically generated

## How to Run Tests

To **run your tests**, navigate from the navbar: [Test] **🡪** [TestExplorer]or just press [Ctrl] + [E], [T]. From there your "Test Explorer" is going to open, press the **two green triangles** to "Run All Tests in View" ([Ctrl] + [R], [V]) or the **single green triangle** to run **the selected test** ([Ctrl] + [R], [T]).

A screenshot of a computer

Description automatically generated 🡪 A screenshot of a computer

Description automatically generated

When you are ready this is **how the tests** **should look like:**

A screenshot of a computer

Description automatically generated

# Unit Test: String Reversal

Next you will test a method which **accepts a string** and **returns** the string but **reversed**.

The method is found in the StringReverse.cs file:

A computer code with text

Description automatically generated

You are given again a **test** **file** StringReverseTests.cs which contains **3 tests**. One of them has been **finished** **partially**:

A computer code with black and red text

Description automatically generated

Finish the first test and using the **names of the other two** finish writing the unit tests:

A close-up of a white background

Description automatically generated

When you are ready this is **how the tests** **should look like:**

A close-up of a computer code

Description automatically generated

# Unit Test: Triangle

Moving right along, a method that prints a **pyramid of numbers** with a given **size** like so:

|  |  |
| --- | --- |
| Size 3 | Size 4 |
| 1  1 2  1 2 3  1 2  1 | 1  1 2  1 2 3  1 2 3 4  1 2 3  1 2  1 |

Found in the Triangle.cs file:

A screen shot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

For the test file again, you are given **the test names**, based on them you should be able to write all the **necessary tests**:

A screenshot of a computer program

Description automatically generated

## Hints

Use Environment.NewLine when formatting your **expected string**.

Make sure your tests pass:

A screenshot of a computer

Description automatically generated

# Unit Test: Factorial

You are given a method which **calculates factorial** of a given number in the Factorial.cs class:

A screenshot of a computer program

Description automatically generated

The drill is the same, implement the unit tests **based on name**:

A screenshot of a computer program

Description automatically generated

The new concept is you need to implement a test that check if the ArgumentOutOfRange **exception** is **throwing** correctly in case of a **negative number**.

Make sure your tests pass:

A screenshot of a computer

Description automatically generated

# Unit Test: Grades

We have a **method** that **receives** a **grade** between **2.00** and **6.00** and prints the **corresponding** **grade** as a string.

* 2.00 … 2.99 ➡ "Fail"
* 3.00 … 3.49 ➡ "Average"
* 3.50 … 4.49 ➡ "Good"
* 4.50 … 5.49 ➡ "Very good"
* … 6.00 ➡ "Excellent"
* Other ➡ "Invalid!"

A screenshot of a computer code

Description automatically generated

When writing the unit tests try to use **multi data tests**. Let's see how, by doing it on our calculator tests:

A screenshot of a computer code

Description automatically generated

A test is just a **method** which means we can pass **parameters** on it. By doing this we can **call** the same test multiple times if the test needs to be executed for different **test cases**.

Having this knowledge, you are given only **2 tests** for the GradeAsWords() **method:**

A screen shot of a computer program

Description automatically generated

One test checks the **normal functionality of the method**, while the second **deals with edge cases**. Both tests accept the **grade** and **expected string** as arguments. Your task is using [TestCase()] write as much test cases as you **see fit**, to cover as **much cases as possible**.

We managed to get 18 test cases in total:

A screenshot of a computer

Description automatically generated

Finally make sure all your tests run:

A screenshot of a computer

Description automatically generated