INTRODUCTION TO UNIT TESTING

In C#, you can think of a unit as a method. You thus write a unit test by writing something that tests a method.

For example, let's say that we had the aforementioned Calculator class and that it contained an Add(int, int) method. Let's say that you want to write some code to test that method.

```
public class CalculatorTester
{    public void TestAdd()
   {     var calculator = new Calculator();
       if (calculator.Add(2, 2) == 4)
           Console.WriteLine("Success");
       else
           Console.WriteLine("Failure");
   }
}
```

Let's take a look now at what some code written for an actual unit test framework (MS Test) looks like.

```
[TestClass]
public class CalculatorTests
{
   [TestMethod]
   public void TestMethod1()
   {
       var calculator = new Calculator();

       Assert.AreEqual(4, calculator.Add(2, 2));
   }
}
```

Notice the attributes, TestClass and TestMethod. Those exist simply to tell the unit test framework to pay attention to them when executing the unit test suite. When you want to get results, you invoke the unit test runner, and it executes all methods decorated like this, compiling the results into a visually pleasing report that you can view.

Let's take a look at your top 3 unit test framework options for C#.

MSTest/Visual Studio

MSTest was actually the name of a command line tool for executing tests,
MSTest ships with Visual Studio, so you have it right out of the box, in your IDE, without doing anything. With MSTest, getting that setup is as easy as File->New Project. Then, when you write a test, you can right click on it and execute, having your result displayed in the IDE

One of the most frequent knocks on MSTest is that of performance. People find the experience can be sluggish. On top of that, many people struggle with interoperability. After all, Microsoft makes it so you can integrate with other Microsoft/Visual Studio stuff, so making it work with third party things would probably not rate as a priority.

NUnit

NUnit started out as a port from Java's JUnit, but the authors eventually redid it to be more C# idiomatic.

NUnit also has the distinction of interoperating nicely with other tools, such as non-Microsoft build platforms and custom test runners. On top of that, NUnit also has a reputation for fast testing running, and it has some nice additional features as well, including test annotations allowing easy specification of multiple inputs to a given test.

The main downside here is that it doesn't integrate into Visual Studio the way that MSTest does. Using it means doing extra work, and installing extra tools, regardless of how easy those tools' authors may make the process.

xUnit.NET

xUnit has a relatively innovative for users to reason about their tests, dividing tests into "facts" and "theories" to distinguish between "always true" and "true for the right data," respectively.

xUnit earns points for creating extremely intuitive terminology and ways to reason about the language of tests. On top of that, the tool has a reputation for excellent extensibility. Another awesome feature of xUnit is actually not a feature of the software, but a feature of the authors. They have a reputation for commitment, responsiveness, and evangelism.

Implement a unit test

Demonstrate unit testing with a hypothetical and fairly trivial calculator class.

```
public class Calculator
{
   public int Add(int x, int y)
   {
      return x + y;
   }
}
```

We have a single class, Calculator, in a class library project.
Let's add a new console project and give it a reference to the project containing calculator, like so.

```
class Program
{
   static void Main(string[] args)
   {
      var calculator = new Calculator();

      int result = calculator.Add(5, 6);

      if (result != 11)
         throw new InvalidOperationException();
   }
}
```

Adding a method and call for each test will prove laborious, and tracking the output will prove unwieldy.

Use MSTest/Visual Studio

Delete the CalculatorTester console project

Right-click on the solution to add a project, and choose the "Unit Test Project Template" after selecting "Test."

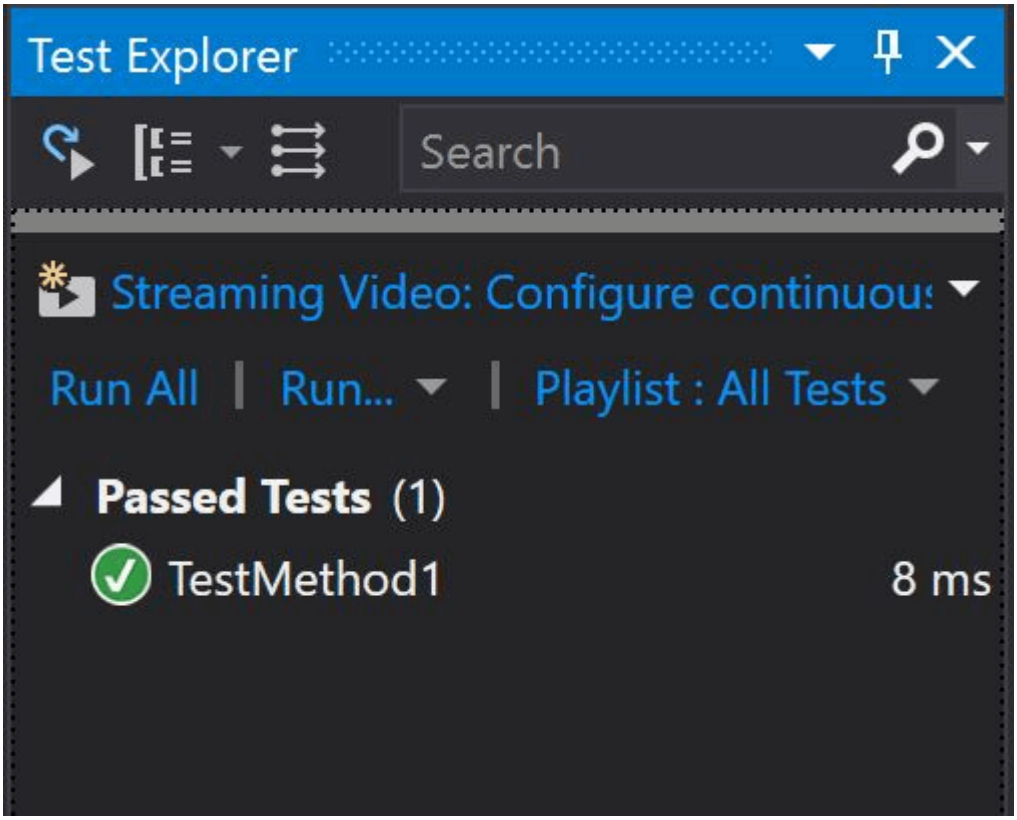
Add the project reference to Calculator again, and then take a look at the class it created for you, called "UnitTest1."

```
[TestClass]
public class UnitTest1
{
   [TestMethod]
   public void TestMethod1()
   {
      var calculator = new Calculator();
```

```
        int result = calculator.Add(4, 3);

        Assert.AreEqual(7, result);
    }
}
```

Now, we've got an actual unit test! Click Ctrl-R, T to run it, and look what happens.



Anatomy of a Unit Test

Let's do a bit of renaming.

```
[TestClass]
public class CalculatorTests
{
    [TestMethod]
    public void Adding_4_And_3_Should_Return_7()
    {
        var calculator = new Calculator();

        int result = calculator.Add(4, 3);
```

```
        Assert.AreEqual(7, result);
    }
}
```

First off, we have a class called “CalculatorTests,” indicating that it will contain tests for the calculator class. (Other ways to conceptually organize your tests exist but consider that an advanced topic.) It gets an attribute called “TestClass” to tell Visual Studio’s default test runner and framework, MSTest, that this class contains unit tests.

Then, we have a method now called “Adding_4_And_3_Should_Return_7.”

We want to name our test methods in a very descriptive way that indicates our hypothesis as to what inputs should create what outputs. Notice the TestMethod attribute above this method. This tells MSTest to consider this a test method. If you removed the attribute and re-ran the unit tests in the codebase, MSTest would ignore this method. You need to decorate any test classes and test methods this way to make MSTest execute them.

Finally, consider the static method Assert.AreEqual. Microsoft supplies a UnitTesting namespace with this Assert class in it. You use this class’s various methods as the final piece in the MSTest puzzle. Assertion passes and fails determine whether the test passes or fails as seen in the test runner. (As an aside, if you generate an unhandled exception in the test, that constitutes a failure, and if you never assert anything, that constitutes a pass.)

Unit Testing Best Practices

Arrange, Act, Assert

The logical components of a good unit test.

Consider your test as a hypothesis and your test run as an experiment. In this case, we hypothesize that the add method will return 7 with inputs of 4 and 3.

To pull off this experiment, first, we *arrange* everything we need to run the experiment. In this case, very little needs to happen. We simply instantiate a calculator object. In other, more complex cases, you may need to seed an object with some variable values or call a particular constructor.

With the arranging in place, we *act*. In this case, we invoke the add method and capture the result. The “act” represents the star of the unit testing show

Finally, we *assert.* The invocation of the Assert class probably gave that one away. But the *assert* concept in the unit test represents a general category of action that you cannot omit

and have a unit test. It asserts the hypothesis itself. Asserting something represents the essence of testing.

One Assert Per Test Method

You should shoot for one assert per test method. Each test forms a hypothesis and asserts it.

Avoid Test Interdependence

Each test should handle its own setup and tear down. You, therefore, cannot count on the test suite or the class that you're testing to maintain state in between tests.